Messaging Interaction Patterns for a Service Bus Concept of PLC-Software

Virendra Ashiwal*, Alois Zoitl*†, *Member, IEEE*
†*CDL VaSiCS*
**LIT CPS Lab, Johannes Kepler University Linz*
Linz, Austria
{virendra.ashiwal, alois.zoitl}@jku.at

***Abstract*—The software architecture of PLC-software has a direct influence on its adaptability, modularity, and flexibility. The software component (Program organization unit, POU) is the smallest unit of PLC-software that is responsible for providing certain machine functionality in an overall production system. The way these software components interact with each other has a significant role in the overall PLC-software architecture. To address current market demands, software components need to be atomic, modular, and loosely coupled. PLC-Service bus-based architecture enables such PLC-software. We are using this architecture as a basis.**

In this paper, we are evaluating available messaging interaction patterns from the enterprise IT domain. Our evaluation is based on requirements defined by the PLC-Service bus architecture. Therewith, we identified a list of interaction patterns that can be used to design interactions between software components in a PLC-Service bus environment. We use a reference production cell as a running example. To evaluate our findings, we first explain what the PLC-Service bus concept would look like for our running example. Later, we used four practical process scenarios from our running example to evaluate the applicability of identified messaging integration patterns for the PLC-Service bus.

***Index Terms*—Factory Automation, Control software, Middleware, PLC service Bus, PLC software, Software components, Interaction pattern, POU**

I. Introduction

Increased number of product variants, batch size one, shorter product life cycle, and volatile demands are the current challenges of production systems. To address these challenges, production systems need to be adjusted and adapted more frequently than ever before.

Hardware mechatronics components such as sensors and actuators are designed to used throughout their lifetime, mostly for several years. Only at the time of development, changes in this kind of hardware can be planned. On the other side, software can be updated more often than its hardware parts which brings the need for designing software so that any future requirements can be accommodated with a minimum downtime [1].

A Programmable Logic Controller (PLC) is a crucial part of the production system. The software of the PLC is comprised of many software components which are responsible for a particular functionality of a production system. With the increasing machine functionality, software systems are getting more complex and interdependent [2].

According to [1], PLC-software affects the quality and functionality of the overall production cell to a large extent, as more and more software components are implemented, improved, and replaced during the life cycle of production cell. Quality criteria of PLC-software such as modularity, flexibility and adaptability, highly depend on the software components and their connections as defined by software architecture.

In the current state of PLC-software, software components are communicating either directly or indirectly via global variables. In both cases, the software components are tightly coupled. Therefore, the resulting PLC-software would be less flexible, less adaptable, and requires high maintenance cost. It also have low portability and low interoperability. They also complicate adopting new trends in the software industry. Due to the aforementioned reasons, a new architecture concept for PLC-software based on the service bus was suggested. In this concept, software components of PLC-software communicate with each other via a service bus. The resulting software components will be more modular, atomic, and loosely coupled. The authors also list requirements of such an architecture, which needed to be fulfilled in order to be accepted in the industrial domain [3].

The PLC-Service bus architecture explained in our previous paper [3] is used as a basis for this paper. This paper will focus on analyzing available integration patterns from the enterprise IT level and using them for the PLC-Service bus concept. In the next section, we will discuss various PLC-software design patterns. Based on requirements defined for the PLC-Service bus, identified Messaging Integration patterns which are applicable for PLC-software are described in Section III. Then a running example is presented in Section IV. In Section V, typical machine message integration scenarios are explained by applying the integration patterns from Section III to our running example. Section VI discusses the outcome of the scenarios with their suitable integration patterns. Finally, with future work paper is concluded in Section VII.

II. Software design patterns in Industrial Automation domain

In software engineering, a design pattern is a common practice to overcome repeated problems that occur while designing any software product [4]. In [5], the author admits that there is a gap between mainstream software industries and the way

software is developed in the industrial automation domain. The author provides an overview of the current practice of software development in PLC-software and tries to bridge the gap to the available mainstream software world.

Zoitl et al. [6] outline structuring principles and design guidelines for automation programs that are purely based on hierarchical control architectures. In [7], the authors propose a design pattern for decomposing or aggregating automation systems and apply it to the VDMA demonstrator following a hierarchical architecture. Inspired by design patterns from object-orientated design, Patil et al. [4] proposed application design patterns for IEC 61499 in four categories: structural, architectural, compositional, and behavioral. For distributed automation applications, authors in [8] proposed a component-based design pattern in form of a set of rules to improve the reusability of programs and reduce time requires for adaptation.

Based on the enterprise service bus (ESB), we described a new PLC-software architecture concept in [3] which had been not explored earlier for PLC-software. Service bus or middleware concepts are not new to the industrial IT domain and have been used in various European projects such as ARUM [9], LISA [10], SOCRADES [11], PERFoRM [12] and BaSys 4.0 [13]. They all considered PLC-software as a single software component in the overall automation architecture and no one considered such a service bus for exploring the interaction between software components of PLC-software itself.

III. Integration Patterns for a Service Bus Concept of PLC-Software

How flexible, reusable, and adaptive a software product can be, unquestionably depends on the interactions between software components. Interaction patterns are widely accepted solutions that would help to achieve loosely coupled interaction between software components. They are well accepted in mainstream enterprise and software industries [14]. Until now, very few of them are common practice and used in PLC-software. Point-to-point communication via global variables [2] and, since the last decade, Pub-Sub via OPC UA [15] are the predominant interaction patterns in PLC-software. In the next subsection, we will discuss criteria for identifying messaging interaction patterns, and then resulting patterns will be listed in the following subsections.

A. Relevant Patterns from Enterprise IT Domain

In our previous paper, we identified a list of requirements that a PLC-Service bus should fulfill. Four of the groups (Requirements of data exchange between software components, message transformation, data integration, and infrastructure modules) are mainly responsible for interactions between software components. Therefore, we used the requirements listed in these four groups to identify message interaction patterns from the enterprise IT domain.

Gregor et al. outlined enterprise messaging interaction patterns in their book [14]. These patterns are well-acknowledged in the enterprise IT domain. They are grouped as message construction, message endpoint, messaging channel, message routing, message transformation, and system management. They are shown in Fig. 1. We analyzed the applicability of these patterns in the PLC-software domain with the aforementioned group of requirements.

All five groups of integration patterns are relevant for automation software. Each group covers a distinct aspect of the communication using a PLC-Service Bus: A sending component has to construct a message before sending it over the Service Bus. Then, an endpoint is needed that will let such software components connect to the PLC-Service bus. There should be a message carrier that transports the message, which is the PLC-Service bus itself. In this architecture, software components are loosely coupled because sender and receiver do not know each other. Therefore, a routing concept is required. Once the message gets routed to the receiver side of the software component, there is a need for message transformation. Message transformation will make sure that the receiver could interpret the original message content. In upcoming subsections, we will discuss identified patterns for the aforementioned five groups.

B. Message Construction

When two software components want to exchange data, they need to wrap the data in message and then transport it over the service bus. Creating such a message depends on: intention of interaction, need of a response, amount of data volume. Patterns for possible message creation in a PLC-software environment are listed as follows:

1) Command Message: When a PLC software component would like to invoke a function or method at another PLC software component at the receiver end, it can use a command message pattern. A Command message is transmitted over the service bus to the receiver end and invokes the function/method locally.

2) Document Message: When a PLC software component needs data from another PLC software component in order to provide certain machine functionality, this data can be transmitted as a document message. This kind of message is read-only in order to have consistency even if there are multiple receivers inside the PLC-software.

3) Event Message: Event Messages are asynchronous notifications. They are the best way to inform other PLC-software components about the action which has been performed so the receiver software component can use it to execute its program.

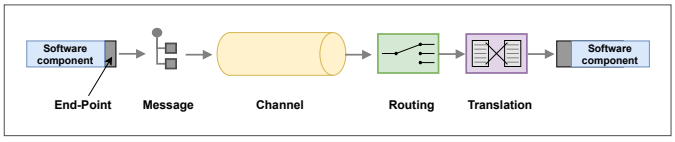

Fig. 1. Messaging integration patterns for PLC-Service bus concept [14]

4) Hybrid Message: A hybrid message is a combination of the aforementioned messages patterns. Here three type of hybrid messages are described as follow:

a) Push Model: Here, the PLC-software components push an event (Event Message) together with data (Document Message). It is more efficient if all receivers want to have both, otherwise it will create unnecessary overhead.

b) Pull Model: When receiving an Event notification, a PLC-software component may require additional information (e.g., data, values). It should then consider the Pull model. Here, the message will be a combination of an event, a command, and a document type. With the Event Message, the receiver will send a request message (Command Message) to get more details. In the reply, it will receive a Document Message with the requested value or data.

c) Correlation Identifier: With asynchronous communication, it is often hard to know the original request message of the received reply (result) message. Therefore, using a correlation identifier would facilitate tracking back each received reply back to its origin request message.

5) Split with sequence: When one software component requires huge data from another software component which does not fit in the single message, it would be easy to divide it in smaller messages with a sequence number and to transmit them via middleware. With sequence numbers, it would be easy to aggregate all messages in their original order.

6) Time-constrained message: For real-time, and time-constrained communication, the message needs to be delivered within a time limit for certain functionality of the production system. Such messages can be specified with an expiration time so that the receiver will know whether it is valid.

7) Message Format Indication: With this pattern, the sender can inform the receiver about the type of message format. With that, the receiver knows which format to use for interpreting the message's contents from several possible predefined known message formats.

C. Message Channels

Message channels create a loosely coupled connection between sender and receiver software components. Patterns that support creating such connections are listed as follows:

1) Publish-subscribe Channel: With the Publish-Subscribe channel, the PLC-software component can announce an event as a result of a change in its state. PLC-software components which are interested in these events can subscribe to them. With selective consumer patterns, a receiver can even filter out all unnecessary published messages.

2) Data Type Channel: To process a message at the receiving PLC-software component, it has to know the type of the message and its structure. Having a dedicated channel for each data type would reduce the message identification burden from the receiver side. For example, for handling alarm data and user management data there should be separate data type channels.

3) Invalid Channel: When a receiver can no longer interpret a message, this messages should go to the invalid channel.

4) Dead Channel: When a message can no longer be delivered, it should go to a dead channel. Reasons for such messages could be when a message expires, if a message can not pass selective consumer criteria, or a receiver is detached after a message is generated.

5) Crash-Proof Delivery: Here, messages with crash-proof delivery will be persisted first at the PLC's memory before they are delivered to the respective software component. It is more useful when communication is between two PLC software components via their PLC-Service bus. This pattern is mostly useful for saving machine settings and parameters.

D. Message Endpoints

A service bus and software components are two separate sets of software. Software components provide functionality to the modular production system, whereas the service bus (Message channel) manages to transport messages for communication. Messaging endpoint encapsulates the implementation part of the interaction of software components to the PLC-Service bus. With that, a software component does not have to care about message format, message channels, and any other details of communicating with other software components via the service bus [14]. A list of the patterns that support messsage endpoints are listed as follows:

1) Selective Consumer: When a receiver would like to receive only selected messages out of all delivered messages from the message channel, it can set suitable criteria/filters for incoming messages.

2) Durable Subscriber: For the publish-subscribe communication pattern, the subscription ends when the connection between software components is disconnected. As a result, published messages would not be received by a disconnected (inactive) software component. If an inactive subscriber still would like to receive all or some specific published messages as soon as it connects back, a durable subscriber pattern would be useful.

3) Idempotent Receiver: When a single message is delivered more than once, then the receiver-side software component should be able to identify it as a duplicate message.

4) Service Activator: A service activator is needed when a service requires synchronous communication, which would however block the resource of the bus. A Service activator will enable synchronous communication with the service while maintaining asynchronous communication with the PLC-Service bus. A reply from the service will be sent to the service bus as soon as it is ready.

5) Message Dispatcher: It works as a match-maker. For each incoming message, it matches available receiver's specialties. Based on this matching, it routes the message. It is useful in a scenario where multiple PLC-software components are available to process a message.

6) Channel Adapter: It works as a messaging client for the PLC-Service bus. It enables the communication between the PLC-Service bus and a non-native application/PLC-software component by converting messages to functions of the non-native application and vice versa.

7) Messaging Bridge: When two PLC-Service buses have to communicate with each other, they can use a special form of channel adapter called a messaging bridge.

E. Message Routing

Message routing is a process which uses rules and filters to deliver a message from a message channel to the receiver. Patterns for message routing are listed as follows:

1) Content Based Router: It checks the message content and based on that routes it to its destination. With this pattern, messages are decoupled from their destination. There is one drawback: Such router has to know the destination.

2) Dynamic Router: All the PLC-software components need to inform dynamic routers about the message type and format they are interested in at the beginning of the operation. Now, based on this information, the dynamic router configures the routing. All these input message types and formats should be persistent. Otherwise all information will be lost at the end of the power cycle.

3) Splitter: With this pattern, each large message can be split into individual messages and can be routed further for processing them.

4) Aggregator: This pattern is the opposite of the splitter. The Aggregator listens to the incoming messages and, based on their content, combines them into a large message. It is a stateful message router.

5) Resequencer: Based on the message content, messages are routed to different channels or destinations. Some of them get processed quickly, while others may take some time. Sometimes, for a particular machine process, all these messages have to be processed in a predefined order. Ensuring this order is the task of the resequencer.

6) Pipes and Filter: It is a messaging pattern to divide a large processing step into a sequence of smaller and independent processing steps, called filters. Filters are connected via pipes. Pipes can be inbound (through which a message is received by a filter) and outbound (through which a message is published by a filter).

7) Scatter-Gather: Scatter-Gather routes a message to multiple receivers and aggregates back the responses from those receivers to a single message.

8) Routing Slip: Pipes and filters are useful when an order of processing of the message is already fixed beforehand, and also each participant (Software component) knows about this sequence, which results in tightly coupled software components. Here, with a routing slip at a message, we detach the message from the software components and enable a loosely coupled scenario. Each routing slip contains a sequence of process steps. To create a routing slip, the sequence of process steps needs to be known beforehand.

9) Process Manager: Routing slip would not work in sequential process steps, when the sequence is highly dependent on the message content, which can dynamically change onfly. The process manager is a dynamic, centralized router and works as an orchestrator. Here, there is more flexibility with respect to routing but at the same time the drawback of being centralized.

10) Command and Query Responsibility Segregation (CQRS): It is a well-known pattern for data storage at the enterprise level. The PLC-Service bus can also benefit from it. With CQRS, we can separate write and read operations via command and query. Two separate datatype channels for each read and write operation to the PLC database are required.

F. Message Transformation

There is a possibility that message constructed by one software component does not have the same semantics, and syntax as the receiver side software component. Therefore, we need transformation patterns that can translate the incoming message in the format which is understandable by receiver. These patterns are listed as follow:

1) Message Translator: Message translator translates a message with respect to the required format of the receiver end software component. It is useful in case of adding a new software component to the existing PLC-software when the newly created software component is with a different data structure and message format. It also gives the freedom of choosing different technology to PLC programmers (based on new software trends and available libraries) for creating a new PLC-software component.

2) Content Enricher: The Content enricher adds an additional required data field for a message. This data is essential for a receiver-end PLC-software component. The content enricher has the ability to look up missing information or compute it from the available data.

3) Content Filter: It is the opposite of the content enricher. It removes unwanted information or data from a message. It also helps to address security concerns as it can remove data that a receiver is not authorized for. In addition to security, it also helps to reduce network traffic and to improve easy message handling.

4) Claim Check: The content filter removes unwanted and unauthorized data permanently. If removal of such data is just for a temporary process, and the data is needed again, then we need to use the claim check pattern. Here, the unauthorized data will be saved in the database with a key that will allow to retreive the message back and rejoin it in its original format.

5) Normalizer: It is a combination of message routers with message translators which is useful when the PLC-Service bus has to communicate with multiple non-native application/software components via a message. With a couple of non-native software components, it is not necessary to have normalizer patterns, and can be done via a message translator. The increase of such non-native software components increases complexity to maintain their respective translator. In this case, a router can be useful as it can automatically direct each incoming message to its respective message translator and translate an incoming message to the message format of the PLC-Service bus.

6) Canonical Data Model: Using a PLC-Service bus requires having a common data model. With this, all the messages will be converted unless they are in the common format at their origin.

IV. Running Example

In this section, we will first describe the VDMA R+A OPC UA Demonstrator [7] and later on, we will apply the PLC-Service bus scenario.

A. VDMA R+A OPC UA Demonstrator

The demonstrator as shown in Fig. 2 was presented at Automatica-2018 to show interoperability, skill-based engineering, and consistent industrial control via OPC UA. It shows a complete production cell architecture with a hierarchical structure with stations, sub-stations, and components. The authors in [7] have described this demonstrator along with its hierarchical structure.

The demonstrator is a production cell and offers the highest level skills, which produce a fidget-spinner as a final product. The next layer of the hierarchical structure consists of six assembly stations, which are: rotary Indexing table (S1), Prepare body (S2), Capping station (S3), Laser engraving (S4), Quality check (S5), and Handover (S6). Within each station, there are several sub-stations. Each of the skills at these stations and sub-stations is application-specific. At each hierarchical layer, the demonstrator has its own controller and offers skills via its own OPC UA Server. These skills are implemented according to the standards IEC 61131-3, and IEC 61499. The hierarchical structure of one of the station (Capping station) [16] is shown in Fig. 3.

For our use case scenario, we will consider S3 (Capping station), S4 (Laser engraving), and S5 (Quality check). The product is transported using an external rotary indexing table from one station to another one. At S3, the desired coloured cap is placed at both sides of the central ball bearing of the fidget-spinner. To do this, the camera unit first checks if the desired cap is available. If it is not available, new caps will drop on the vibrator from the feeder sub-station. Next, it checks the orientation of the desired cap and if it is not in the right orientation, the vibrator will start to flip it. When a

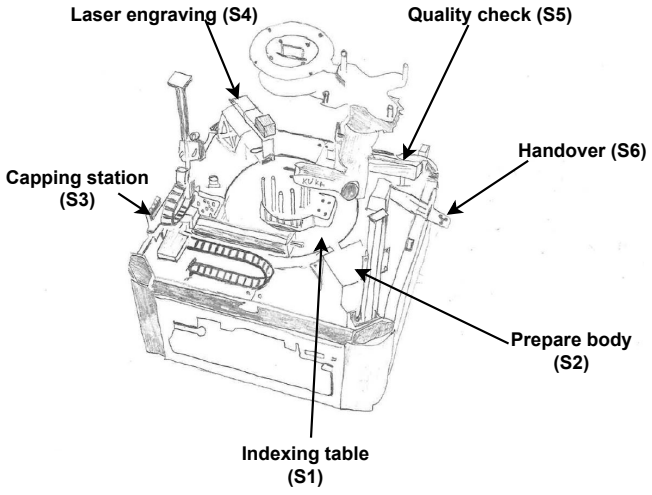

Fig. 2. VDMA R+A OPC UA Demonstrator [16]

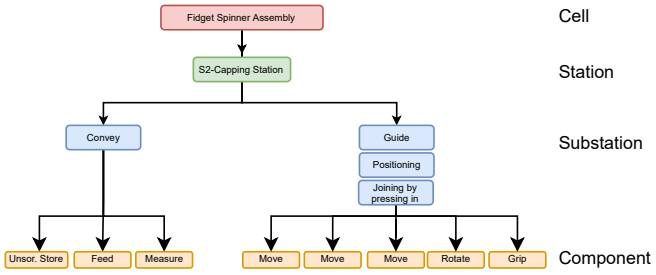

Fig. 3. Hierarchical classification of the Capping station (S3) [16]

correct oriented cap is found at vibrator, it will be placed at the both side of the product (fidget-spinner). After that the fidget-spinner moves to S4, where a personalized text on one side of the cap and the VDMA-logo on the other side of the cap will be engraved. At S5, it checks the quality of the product by a camera, whether the colour of the caps, and texts on them are the same as the desired one. If the product passes the quality check, it will be go the Handover station (S6) [7].

B. VDMA Demonstrator with the PLC-Service bus

If we apply the PLC-Service bus architecture to S3, S4, and S5 of the VDMA demonstrator, the result can be seen in Fig. 4. It shows a visual hypothesis of the VDMA demonstrator. We encapsulated services related to one mechatronic component into a software component. In the case of S3, there is a component *S3-Camera* for a camera that finds a cap at the vibrator. Then, we have a component for filling *S3-Bunker* which adds more caps if there is no cap available at the vibrator. There is a component for the vibrator itself, *S3-Vibrator*, which provides two services: flip, and separate. Services from the gripper (open and close) are encapsulated in the software component *S3-Gripper*. For the movement of the cap from the vibrator to the fidget-spinner there is a software component called *S3-Axis*. All software components from station 3 have a prefix of *S3-*, and alike for S4 and S5: they have a prefix of *S4-* and *S5-*, respectively. There is also a software component *Product*, which has information as shown in Fig. 6. They are product ID, name, current position, the product configuration coming from the customer such as colour and engraving details. Infrastructure components which are common in all machines, such as *Alarm Server*, *Report* and *Trend Server* and 3rd party software components such as *OEE (Overall equipment effectiveness) calculation*, *Data Analytics* can also be seen in Fig. 4. Here, all components will be at the same level and without any hierarchical structure as previously mentioned by [7].

V. Applying Integration patterns at VDMA Demonstrator with PLC-Service bus architecture

Product-driven machine station initialization, integrating non-PLC software components, handling of time expired and not interpretable data, transportation of huge amount of data, and accessing the database are typical machine processes scenarios for the PLC-software. We are using these scenarios for

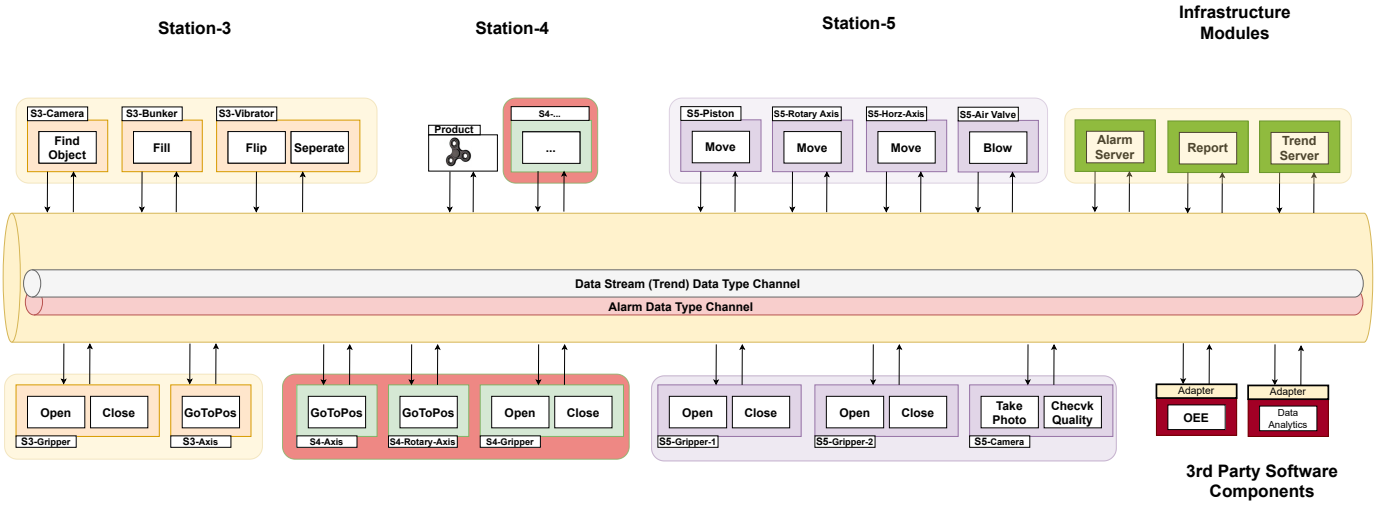

Fig. 4. PLC-Service bus for the VDMA Demonstrator with S3, S4, and S5

Fig. 4 and they are not exactly similar to the original processes of the VDMA demonstrator. Based on these scenarios, we first identify interactions between software components, and later on map these interactions to the suitable message interaction patterns from Section III.

A. Scenario 1: Product driven machine station initialization

To provide overall machine functionality, PLC software components need to interact. Overall flexibility, modularity, and adaptability will increase if these software components are loosely coupled. In this scenario, we are considering software components of S3, S4, and Product. S3 and S4 contain various mechatronic components. The functionality of these mechatronics components is encapsulated in their respective software components and placed at the same level in the PLC-Service bus. This arrangement is shown in Fig. 5.

The *Product* (fidget-spinner) carries information about its current position, the desired colour that is entered by the customer. In this subsection, we show the initialization of a station S3 and, S4, when a product (fidget-spinner) reaches at these stations. We can see these interactions in Fig. 5 between *Product*, *S4-Piston*, *S3-Camera* and, *S3-Bunker*. Here *Product* send its data via message in the PLC-Service bus. This message contains attributes such as name, ID, *colour*, *current_Pos* and Operating_Time. *S3-Bunker* and *S4-Piston* is only interested in *current_Pos*. Based on values of these attributes, the respective station starts to work. In case of S3, *S3-Bunker* and *S3-Camera* and in case of S4, *S4-Piston* initiates first. After initialization of S3, *S3-Camera* checks if the desired coloured cap is available at *S3-Vibrator*, and if it is not available, then *S3-Camera* asks *S3-Bunker* to add more caps in the *S3-Vibrator*. The processes from aforementioned scenario are as follows:

1) Process-1 (P1): S3 and S4 start to work when the fidget-spinner reaches these stations.

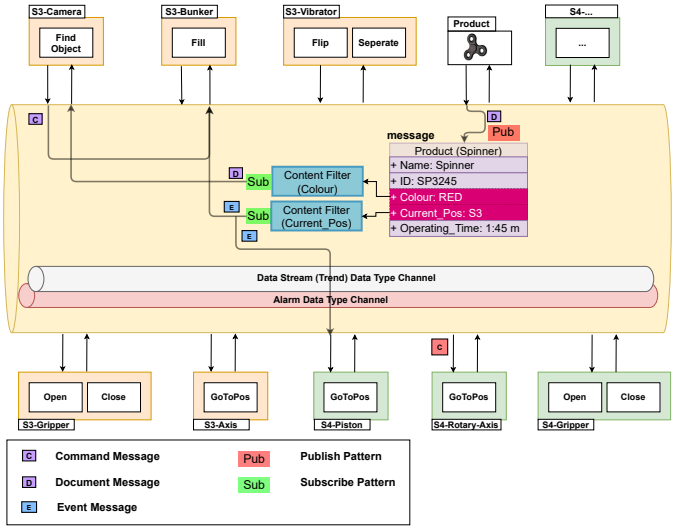

Fig. 5. Messaging integration patterns for Product driven machine station initialization and to prevent to overwhelmed with unnecessary messages

2) Process-2 (P2): The bunker feeds more caps in the vibrator when the camera detects that a cap in the desired colour is no longer available.
3) Process-3 (P3): Bunker and piston software components avoid unnecessary messages from PLC-software.

Interaction patterns from Section III to address the aforementioned processes is shown in Fig. 5. For P1, *S3-Bunker* and *S4-Piston* use the publish-subscribe pattern to subscribe to *current_Pos* from Structured Type data of *Product*. For P2, first *S3-Camera* uses publish-subscribe pattern to subscribe to the colour of the desired cap and if a cap is not available, it will use a Command Message pattern to start *Fill* service at *S3-Bunker*. For P3, *S3-Bunker* and *S4-Piston* use the Content Filter pattern to filter out unnecessary data.

Fig. 5 shows what kind of message one should use, how loosely coupled interaction can be achieved by publish-

subscribe pattern, and how to prevent a software component from being overwhelmed with unnecessary messages by the Content filter pattern.

B. Scenario 2: Integrating OEE and Data Analytics software components at PLC-Software

We are analyzing here how non-PLC software components can be integrated into PLC-Software. In Fig. 6, OEE and Data Analytics are software components, which are not programmed in PLC languages. They are very important to provide key performance indicator (KPI) to the customer. In this case, there are interactions between OEE, Data Analytics, and the PLC-Service bus itself.

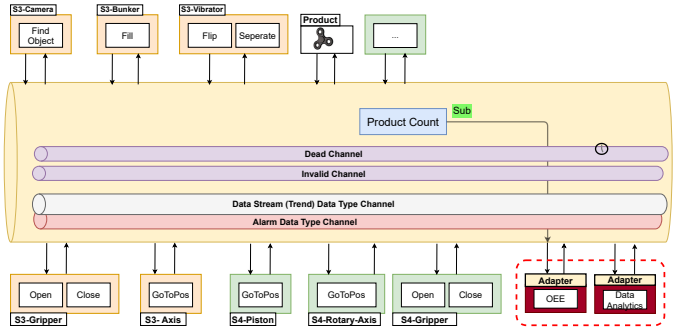

Fig. 6. Messaging integration patterns for integrating non native software component to the PLC software and to handle expired + invalied messages

A Channel adapter (adapter) pattern is needed in this scenario. This is shown in Fig. 6. The adapter works as a mediator between the service bus and the non-native software components. It will convert the PLC-Service bus messages into the functions of the respective software components and vice-versa.

C. Scenario 3: Handling of expired and not interpreted messages

In the PLC-Software, there is a possibility that a time-critical message is not delivered in its predefined duration. There is also a possibility that the receiver fails to interpret a message. It is not a good solution to let these messages stay in the channel. In Section III, we have identified two patterns to resolve these situations. These patterns can be seen in Fig. 6 as Dead Channel and Invalid Channel. All expired messages will be sent to the Dead Channel and all messages which failed to be interpreted will be sent to the Invalid Channel.

D. Scenario 4: Historical data access from database

Data Analytics software component needs access to historical data in order to provide some value added services and KPI of the production. Some historical data can be too large for a single message to carry. On the other side, if the machine is in production, then the PLC writes data to the database. Sometimes reading and writing to the same database can be hard to handle. This scenario can be described in these two processes as seen in Fig. 7:

1) Process-1 (P1): Here, large messages are transported from the database to the software component of data analytics.
2) Process-2 (P2): Accessing the database separately based on either reading or writing.

For P1, When reading data from a database with high volume, the *Message Sequence* pattern can be used. Here, huge data can be divided into small messages with acceptable size. A sequence number is assigned to these messages, which is used on the receiver side to rearrange the original messages. For P2, using the CQRS pattern, we can create separate channels for reading and writing in the database. Doing this will separate concerns and improve the flexibility and maintainability of the database. CQRS also supports scalability.

We presented here, how a large message can be transported between software components, and how to use separate reading and writing data for PLC database.

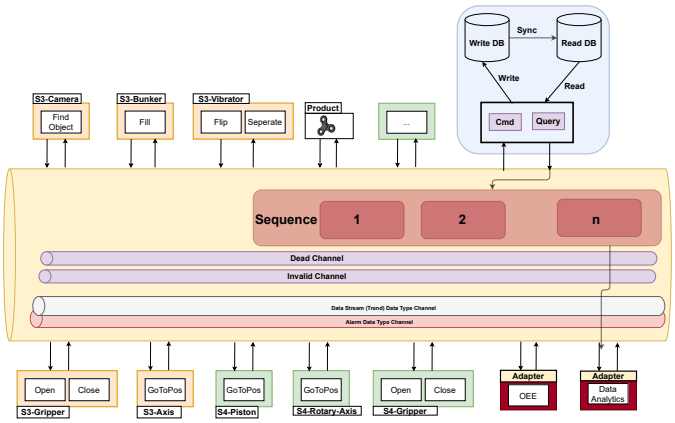

Fig. 7. Messaging integration patterns for accessing historical huge data

VI. Discussion

Messaging interaction patterns are widely recognized best practices that are available in the enterprise IT domain. Very few of these patterns are common practice for developing PLC-software. Based on the requirements from the PLC-Service bus, in this paper, we analyzed integration patterns from enterprise IT. The outcome is presented in Section III and can be utilized in the PLC-Service bus. We used the VDMA demonstrator as a running example. With this, we identified four typical machine process scenarios.

For scenario 1, the publish-subscribe pattern is well suitable as it provides loosely coupled software components. With a content filter pattern, we can ensure that unnecessary messages can be avoided for a software component. Messages can be created based on the intention of the interaction such as: for just notifying, one can use Event message pattern, for initiating a service at the receiver side, one can create Command message pattern. For carrying data, one can use document message patterns. Choosing the right pattern would improve the interaction between software components. In scenario 2, how an external software component can be integrated with the

PLC-Service bus via a Channel adapter pattern is shown. This gives freedom to the application engineer to accommodate the best practices from the mainstream software domain. It is also possible to handle timely expired and invalid messages via dead and invalid datatype channels. Accessing historical data when they are huge in volume can be handled with Message sequence and CQRS patterns.

Due to the space limitation, it was not feasible to show all the identified patterns with respective machine processes. We believe that with these patterns, PLC-software qualities such as flexibility, modularity, and adaptability would improve.

VII. Conclusion and future work

Direct connection and indirect connections (global variables) are two common practices in PLC-software for the interaction of its software components. PLC-software with such practices results in tight coupling, is less modular, and less flexible. It also needs high maintenance efforts to accommodate new customer requirements.

The PLC-Service bus concept is an architecture for the PLC-software to enable flexible and adaptable production systems. It is based on the concept of an enterprise service bus, which was introduced around two decades ago to address the aforementioned similar issues. Software components in the PLC-Service bus architecture are loosely coupled and thus are more flexible and accommodate any new requirements.

Based on the requirements available for the PLC-Service bus concept, we evaluated available messaging integration patterns in the enterprise IT domain. These patterns are well established and acknowledged in their domain. Until now, very few of them are realized in PLC-software. The outcome of this evaluation is presented as a list of the suitable patterns. Furthermore, we used the VDMA demonstrator as a running example. With this, we showed how the software architecture of the running example would look like with a PLC-Service bus. We identified typical automation process steps from the running example and described them as scenarios. After that, we used previously identified messaging integration patterns to address these scenarios and evaluate their applicability.

In future work, there is a need to map more of these identified patterns to the PLC process scenarios. To verify our finding, a prototype based on some commonly used patterns should be realized which can be used to compare various patterns. Based on that it is further necessary to provide recommendations of patterns for each PLC process scenario.

References

[1] B. Vogel-Heuser, J. Fischer, S. Rösch, S. Feldmann, and S. Ulewicz, "Challenges for maintenance of PLC-software and its related hardware for automated production systems: Selected industrial case studies," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 362–371.

[2] J. Fuchs, S. Feldmann, C. Legat, and B. Vogel-Heuser, "Identification of design patterns for IEC 61131-3 in machine and plant manufacturing," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 6092–6097, 2014, 19th IFAC World Congress. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016425668

[3] V. Ashiwal, A. Zoitl, and M. Konnerth, "A service bus concept for modular and adaptable plc-software," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 22–29.

[4] S. Patil, D. Drozdov, and V. Vyatkin, "Adapting software design patterns to develop reusable IEC 61499 function block applications," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 2018, pp. 725–732.

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

[6] A. Zoitl and H. Prähofer, "Guidelines and patterns for building hierarchical automation solutions in the IEC 61499 modeling language," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 4, pp. 2387–2396, 2013.

[7] B. Brandenbourger and F. Durand, "Design pattern for decomposition or aggregation of automation systems into hierarchy levels," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 895–901.

[8] W. Dai and V. Vyatkin, "A component-based design pattern for improving reusability of automation programs," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, 2013, pp. 4328–4333.

[9] P. Leitão, J. Barbosa, M.-E. C. Papadopoulou, and I. S. Venieris, "Standardization in cyber-physical systems: The arum case," in *2015 IEEE International Conference on Industrial Technology (ICIT)*, 2015, pp. 2988–2993.

[10] A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, and B. Lennartson, "An event-driven manufacturing information system architecture," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 547–554, 2015, 15th IFAC Symposium on Information Control Problems in Manufacturing.

[11] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. M. S. d. Souza, and V. Trifa, "SOA-based integration of the internet of things in enterprise services," in *2009 IEEE International Conference on Web Services*, 2009, pp. 968–975.

[12] F. Gosewehr, J. Wermann, W. Borsych, and A. W. Colombo, "Specification and design of an industrial manufacturing middleware," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 1160–1166.

[13] E. Trunzer, A. Calà, P. Leitão, M. Gepp, J. Kinghorst, A. Lüder, H. Schauerte, M. Reifferscheid, and B. Vogel-Heuser, "System architectures for industrie 4.0 applications," *Production Engineering*, vol. 13, no. 3, pp. 247–257, Jun 2019.

[14] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[15] OPC Foundation , "OPC 10000 14 unified architecture part 14 pub sub introduction," https://reference.opcfoundation.org/v104/Core/docs/Part14/5.1/, accessed: 2021-05-14.

[16] P. Zimmermann, E. Axmann, B. Brandenbourger, K. Dorofeev, A. Mankowski, and P. Zanini, "Skill-based engineering and control on field-device-level with OPC ua," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 1101–1108.