This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Atmojo, Udayanto Dwi; Vyatkin, Valeriy; Salcic, Zoran
On Achieving Reliable Communication in IEC 61499

Published in:
Proceedings of the 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation, ETFA 2018

DOI:
[10.1109/ETFA.2018.8502487](https://doi.org/10.1109/ETFA.2018.8502487)

Published: 22/10/2018

Document Version
Peer reviewed version

Please cite the original version:
Atmojo, U. D., Vyatkin, V., & Salcic, Z. (2018). On Achieving Reliable Communication in IEC 61499. In *Proceedings of the 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation, ETFA 2018* (Vol. 2018-September, pp. 147-154). [8502487] (Proceedings IEEE International Conference on Emerging Technologies and Factory Automation). IEEE.
<https://doi.org/10.1109/ETFA.2018.8502487>

This is the accepted version of the original article published by IEEE.

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

On Achieving Reliable Communication in IEC 61499

Udayanto Dwi Atmojo¹, Valeriy Vyatkin^{1,2}, Zoran Salcic³

¹Dept. of Electrical Engineering and Automation, Aalto University, Finland

²Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden

³Dept. of Electrical and Computer Engineering, The University of Auckland, New Zealand

Email: udayanto.atmojo(at)ieee.org; vyatkin(at)ieee.org; z.salcic(at)auckland.ac.nz

Abstract—This paper proposes a novel extension for communication in the IEC 61499 standard. Inspired by the features found in the formal programming language SystemJ, the extension supports reliable and guaranteed communication in distributed execution of function block application(s)/program(s). The extension utilizes mechanisms agnostic on underlying network protocols and are based on formal semantics that guarantee data delivery. The use of proposed extension, called channel, is demonstrated in an industrial automation-type example.

Keywords— IEC 61499; reliable communication; blocking; channel; formal semantics; distributed control

I. INTRODUCTION

In the industrial automation domain, there is increasing interest into and also migration from the legacy IEC 61131-3 [1] to the newer IEC 61499 [2] standard. Unlike its predecessor, the IEC 61499 enables easier development of distributed systems, where software control logics, implemented in the form of function blocks (FBs), can be distributed into multiple hardware controllers and composed to form an IEC 61499 application (program). In IEC 61499, communication between distributed function blocks is supported by the software runtime of the IEC 61499 tools, e.g., FBDK [3], 4DIAC [4], or NXTStudio [5]. These tools leverage the so-called Communication Service Interface Function Blocks (CSIFBs) to implement physical communication interfaces, allowing FBs on multiple hardware controllers to communicate with each other. Among the most typically used CSIFBs are the CLIENT/SERVER and PUBLISH/SUBSCRIBE, which are the generic communication patterns in the IEC 61499 standard.

In the aspect of communication in IEC 61499, typical research works revolve around mapping the above mentioned patterns to various communication protocols, for example, eXtensible Messaging and Presence Protocol (XMPP) [6], Common Industrial Protocol (CIP) [7], Ethernet/IP [8] and Profibus DP [9]. Other efforts emphasize on improving or extending existing communication provision, for example, modified version of OPC-UA [10] and Real Time Publish Subscribe (RTPS) [11]. Some IEC 61499 software tools have also introduced additional communication provisions complying with the IEC 61499 standard. For example, the OPC-UA [12] has also been made available in some IEC 61499 software tools. Also, several communication protocols, e.g., openPowerlink, MQTT, and Modbus, have been made available in 4DIAC. Other attempts utilize different approaches

on a higher level, for example, in [13] where a service-oriented architecture-based interaction is introduced in IEC 61499 to provide interoperability between different hardware controllers and IEC 61499 software implementations. However, the aforementioned attempts are ill-focused on enabling reliable, dependable communication, which the IEC 61499 standard is currently lacking. When higher degree of reliability in communication is needed, programmers often need to resort to low-level communication primitives and implement synchronization mechanisms themselves, which can be a nightmare especially when dealing with large-scale, complex distributed systems.

This paper takes a different approach from the above state of the art by proposing a new extension that introduces reliable communication in the IEC 61499 standard. The extension adopts the abstracted communication mechanism called channel used in the formal language SystemJ [14]. The channel mechanism is based on formal semantics which guarantees exchange of data between sending and receiving partners at the application level, regardless of the underlying network communication protocols/interfaces being used. The channel is included into IEC 61499 through the new, standard function blocks that can be used by the programmers at the application level. The proposed extension is showcased in an industrial automation-type example to demonstrate its use by programmers in straightforward manner, without the need to resort to low-level synchronization or communication primitives. In addition, this paper gives another perspective in answering the question raised in [15] of whether IEC 61499 and SystemJ are “competitors” or “cousins”.

This article is organized as follows. Section II briefly introduces the formal language SystemJ and the feature in the language adopted into the extension proposed in this article. Section III describes how the communication provision from SystemJ is brought and mapped into IEC 61499. Section IV showcases the use of the proposed extension. Finally, Section V concludes the article and presents potential future works.

II. THE FORMAL LANGUAGE SYSTEMJ

Before going into the detailed discussion regarding the proposed communication provision, it’s imperative that readers are given some knowledge regarding the formal language SystemJ in order to understand the provision proposed in this article.

SystemJ is a formal system-level programming language which complies with the Globally Asynchronous Locally

Synchronous (GALS) model of computation [16]. Being based on formal semantics that allow for formal verification of software function, the language utilizes Java for data computation and reactive statements for control-dominated computation. In SystemJ, software logics are encapsulated as concurrent entities, where they are asynchronous at the top-level, referred as *clock domains* (CDs), with individual clock domains potentially consist of collective of software entities advancing in lock-step (synchronously), referred as *reactions*. SystemJ incorporates the notion of logical time represented by *ticks*.

A. SystemJ : Brief Overview

Being a system-level language, SystemJ utilizes abstracted communication mechanisms, namely signals and channels, to facilitate interactions between concurrent software entities and with the environment. The use of abstract communication mechanisms in SystemJ alleviates the need to deal with low-level implementation details of physical interfaces, e.g., network protocols and physical I/O hardware (sensors, actuators, and controllers). Signals and channels can be mapped into various physical mechanisms, where the Java-based SystemJ Runtime Support (RTS) describes their implementation.

In SystemJ, a group of CDs may be handled by the same RTS instance, which belong to the same SystemJ *subsystem*. At least one subsystem forms a SystemJ *system*. CDs that belong to different subsystems interact with each other through channels, with the data exchanged physically through different physical media called *links*. Fig.1 shows a graphical illustration of a SystemJ *system* which comprises 4 subsystems (SS1-SS4) interconnected via links and distributed on three computing machines (M1-M3).

Fig. 1. Illustration of SystemJ system.

As the only communication mechanisms in SystemJ, signals and channels have different semantics. Signals are used to interact between reactions in the same CD and with the environment. Once emitted, they are visible to all reactions in the CD (broadcasted). On the other hand, channels are used exclusively for communication between reactions in different CDs. Unlike signals, channels provide simplex point-to-point communication (through *send* and *receive* statements in SystemJ) with inherent full handshake (called *rendezvous*)

which guarantees the delivery of message. Being unidirectional communication mechanism, channel has input or output port. Input port belongs to the receiver and is connected to an output port of the sender, while output port belongs to the sender and is connected to an input port of the receiver. Due to the blocking nature of the channel, communication occurs only when both sides (the pair) are ready to exchange data. Otherwise, one side will wait until its partner is ready. An ongoing or uninitiated rendezvous can be preempted at certain point in time, at a tick boundary. Channel communication can involve data exchange or, when no data is exchanged, it is used only for synchronization between the sender and receiver. If data exchange is involved, it is guaranteed before the sender and receiver can continue with their execution. This guarantee inspired the introduction of a similar mechanism into the IEC 61499 for communication.

B. SystemJ Channel Semantics

Here we describe the semantics of SystemJ channel in sufficient level for the purpose of this article. More details on SystemJ channel semantics can be found in [14]. Channel rendezvous is implemented using a set of statuses and value buffers as follows.

1. A write-sent status (w_s). This status is attached to the output channel of the sending CD and indicates that the sending CD is ready to initiate rendezvous and send any data (if any data is included).
2. A read-received status (r_r). This status is attached to the input channel of the receiving CD and contains a copy of the write-sent status of the output channel of the sending CD. This status notifies the receiving CD that the sending CD is ready to initiate rendezvous.
3. A read-sent status (r_s). This status is attached to the input channel of the receiving CD. This status acts as an acknowledgment from the receiving CD and is sent to the sending CD to inform that the receiving CD is ready for rendezvous.
4. A write-received status (w_r). This status is attached to the output channel of the sending CD and contains the copy of the read-sent status from the receiving CD.
5. Preemption statuses of the sending (p_s) and the receiving (p_r) side. These preemption statuses indicate that preemption occurs on the sending or receiving CD.
6. Data buffer of the output channel of the sending CD (v_s), which stores the data (if any) to be transmitted to the receiving CD.
7. Data buffer of the input channel of the receiving CD (v_r), which is used to store received data (if any) from the sending CD.

Channel communication in SystemJ follows a set of rules based on the formal semantics described in [14]. These rules can be summarized into two algorithms shown in Listing 1 (for sending via output channel port) and Listing 2 (for receiving via input channel port), respectively (both are rewritten from [14]).

Listing 1. The algorithm of the *send* statement.

```

1  abort (immediate pr>ps)
2  {
3      ws++;
4      vs=data;
5      abort (immediate ws=wr) {
6          abort(immediate Ips) {
7              while(true) {
8                  pause;
9              }
10         }
11     }
12     do{
13         reset (ws, wr, vs) ;
14         ps=pr;
15     }
16 }
17 do{
18     reset (ws, wr, vs) ;
19     ps=pr;
20 }

```

Both algorithms make use of SystemJ statements *abort* and *pause*. In SystemJ, the *abort* statement implements strong preemption which preempts execution of software logic with explicit scope, while *pause* consumes (terminates the current) tick. In case of output channel, when control point reaches a *send* statement (the transmission from an output channel is initiated), the algorithm in line 3-15 proceeds if the receiving side is not preempted, otherwise line 17-20 is executed. Meanwhile, for input channel, the algorithm in line 2-16 of Listing 2 proceeds if the sending side is not preempted (line 1 of Listing 2), otherwise line 17-20 of Listing 2 is executed.

Continuing on Listing 1, once the control point hits the *send* statement in a SystemJ program, provided that the receiving side is not preempted, the write-sent status is incremented (line 3), the value buffer of the output channel is filled with data (if any) (line 4), and then the channel is blocked (line 7-9) until the receiving side is ready to receive (indicated by the condition $w_s=w_r$) or if preempting signal (i.e., a subset of the set of preempting signals Ip_s) has been emitted and becomes present. Once the receiving side is ready to receive or if preempting signal is present, the blocking terminates. If preempting signal is present (line 6), the output channel statuses and buffer are reset and the preemption status of the receiving side is copied to the preemption status of the sending side (line 12-15).

Listing 2. The algorithm of the *receive* statement.

```

1  abort (immediate ps > pr) {
2      abort (immediate rr > rs) {
3          while (true) {
4              pause;
5          }
6      }
7      abort (immediate Ips) {
8          rs++;
9          pause ;
10         vr = data;
11     }
12     do{
13         reset (rs, rr, vr) ;
14         pr = ps;
15     }
16 }
17 do {

```

```

18     reset (rs, rr, vr) ;
19     pr = ps;
20 }

```

Referring to Listing 2, assuming that the sending side is not preempted, the channel is blocked until the sending side initiates transmission and is ready to rendezvous (indicated by the condition $r_r > r_s$) (line 2). Once this condition is achieved, r_s is incremented, indicating an acknowledgement to the sending side that the receiving side is ready to receive data, provided that preemption signal is not present (line 7-8). The algorithm consumes one tick, then updates its buffer with the transmitted data (line 9-10). Finally, the rendezvous is completed. In case preemption signal is present (line 7), the input channel statuses and data buffer are reset and the preemption status of the sending side is copied to the preemption status of the receiving side (line 13-14). The reset function in Listing 1 and Listing 2 can be in Listing 3.

Listing 3. Algorithm of reset function

```

ws , wr = 0;
rr , rs = 0;
vr , vs = null; //emptied
ps = ps + 1;
pr = pr + 1;

```

III. INTEGRATION OF SYSTEMJ CHANNELS TO IEC 61499

The features of channel communication in SystemJ, e.g., built-in formal semantics to achieve reliable communication, are an additional communication mechanism that can be used in the IEC 61499 standard to provide reliable and guaranteed communication. This article presents how the SystemJ channels are included and implemented in IEC 61499 by mapping the algorithms in Listing 1, Listing 2, and Listing 3, into IEC 61499 function blocks that can be used by the application developers. The implementation takes into account on how the SystemJ RTS handles the channels and uses it as a reference for implementation of channels in IEC 61499 by including the elements of SystemJ RTS into function blocks.

A. Implementing SystemJ Channel in IEC 61499

Here, NXTStudio is used as the software tool for implementation. The implementation of channel in IEC 61499 consists of two basic function blocks, namely InputChannelInstance, which corresponds to the function block that implements the algorithm in Listing 2, and OutputChannelInstance, which corresponds to the function block that implements the algorithm in Listing 1. The InputChannelInstance and OutputChannelInstance function blocks are shown in Fig.2.

With regard to their input data ports, both FBs receive the communication statuses from their respective partners (Part_w_s, Part_w_r, Part_Preempted, corresponding to w_s , w_r , and p_s statuses sent by the OutputChannelInstance and received by InputChannelInstance and Part_r_s, Part_r_r, and Part_Preempted, corresponding to r_s , r_r , and p_r statuses sent by the InputChannelInstance and received by OutputChannelInstance). The InputChannelInstance function block receives data transmitted by the sender (DataIn), while the OutputChannelInstance function block can buffer (channel)

data which will be transmitted to the receiving side (if any) based on the HasData (of data type BOOL) value. Similar to SystemJ, it is possible not to include any data (to be transmitted from sending to receiving side) when using the OutputChannelInstance and InputChannelInstance function blocks. Both function blocks output their respective channel communication statuses (and data to be transmitted, if any in case of OutputChannelInstance) which then are transmitted to their partner. For InputChannelInstance, it outputs the received (channel) data if any and generates the ChannelReceived event when rendezvous is completed.

Fig. 2. InputChannelInstance (top) and OutputChannelInstance (bottom) function blocks

The Execution Control Chart (ECC) which governs the behaviour of InputChannelInstance function block is shown in Fig. 3. Referring to Fig. 3, when rendezvous is not yet initiated, the FB can receive updated channel statuses of its partner and store a copy in internal variables. The attempt to perform rendezvous by input channel is initiated after receiving the REQ event. The 'WithCSIFB' data input needs to be provided when triggering the REQ event, TRUE when CSIFB is used (in multiple distributed IEC 61499 programs setting) or FALSE (in single IEC 61499 program setting). Upon receiving REQ, the FB moves to the *GetWithCSIFBnUpdSt* state to obtain the value of the 'WithCSIFB' data input and updates the partner with the channel communication statuses by transmitting the *update_statuses_to_partner* event and the associated data outputs, then the FB moves to the *BLOCKED* state. If the partner is not ready to initiate rendezvous, the InputChannelInstance FB remains in the *BLOCKED* state while also able to receive updates of the partner's statuses and save a copy to store internally. This behaviour consumes physical time and emulates the "while(true){pause;}" in line 3-5 of Listing 2. When the statuses indicate that the partner is ready to rendezvous (indicated by the " $r_r > r_s$ ", which corresponds to line 2 of Listing 2), the function block moves to the *ReceiveSync* state, where it increments r_s (line 8 of Listing 2), (corresponding to an acknowledgement that it has synchronized with the partner) and then transmits its (updated) statuses to the partner. In multiple distributed IEC 61499 programs setting (if 'WithCSIFB' data input is set to 'TRUE'), it needs to synchronize with the physical communication functionality, so it will wait to receive event through *Confirmed_Transmission* which needs to be generated by the

physical communication functionality indicating that it has successfully transmitted the channel statuses. Finally, the function block outputs the received data (if any), resets the channel statuses and data buffer, and generates *ChannelReceived* event indicating that the channel communication is complete.

The ECC which governs the behaviour of OutputChannelInstance function block is shown in Fig.4. Referring to Fig.4, the function block initiates rendezvous when REQ event is received. Also, the 'WithCSIFB' data input needs to be provided when triggering the REQ event, TRUE when CSIFB is used (e.g., in multiple distributed IEC 61499 programs setting) or FALSE otherwise. Depending on whether the channel communication will include (channel) data, the function block goes to *StartSend_WithData* or *StartSend_NoData*. In these states, apart from obtaining the value of the 'WithCSIFB' data input, the w_s is incremented and (channel) data is stored in the data buffer to be transmitted to the receiving side. Also, the function block generates the *update_statuses_to_partner* or *update_statuses_data_to_partner* event along with its communication statuses and data to be transferred to the receiving side. Then, in multiple distributed IEC 61499 programs setting (if 'WithCSIFB' data input is set to 'TRUE'), the FB needs to synchronize with the physical communication functionality to complete (*WaitTransmissionConfirmation*), while concurrently also being able to receive updates of the partner's statuses and save a copy internally. Next, the FB waits for the partner to be ready to rendezvous when it reaches the *BLOCKED* state. This behaviour consumes physical time and emulates the "while(true){pause;}" in line 7-9 of Listing 1, where the output channel consumes tick until the partner is ready to rendezvous. The partner is ready to rendezvous when the function block receives statuses such that $w_s = w_r$, after which the function block moves to the next state where it will reset its statuses and data buffer (housekeeping), and generate the *ChannelsIsSent* to indicate that channel communication is complete.

The InputChannelInstance and OutputChannelInstance FBs can be wrapped into composite FBs, which are composed of both FBs connected with adapter FB. In Fig.5, two composite FBs, *AdptdInChan* and *AdptdOChan*, use adapter FB namely *UPDChanAdp*, which connects the InputChannelInstance and OutputChannelInstance FBs and other FBs, e.g., CSIFBs in multiple distributed IEC 61499 programs setting or other channel FBs (partner) in single IEC 61499 program setting. The adapter FB groups sets of events and data into 'buses', where in *AdptdInChan* and *AdptdOChan*, these buses are namely *RCV_TRNS* and *SND_TRNS*.

B. Comparisons between SystemJ and IEC 61499 Channel

The semantics of IEC 61499 adaptation of channel is to some extent different from one of SystemJ. For example, the notion of tick is not present in IEC 61499. At this stage, this is resolved by having the channel FBs to emulate the behaviour which involves the pause statement, e.g., line 3-5 of Listing 2 and line 7-9 of Listing 1 by having the *BLOCKED* state of both channel FBs with behaviour as elaborated in Section IV.A. The pause statement is inserted in line 9 of Listing 2 since in SystemJ, the r_s of input channel is incremented first before the

Fig. 3. The ECC of InputChannelInstance function block

Fig. 4. The ECC of OutputChannelInstance function block

Fig. 5. AdptdInChan FB and its composition (top), AdptdOChan FB and its composition (bottom).

input channel statuses are transmitted to the partner by the RTS, which executes at the end of the tick. On the other hand, in IEC 61499 channel implementation, this behaviour is emulated by having the channel statuses transmitted to the opposite party once r_s is incremented. If CSIFB is involved, the channel FB then waits for the CSIFB to successfully transmit the channel statuses before the channel FB proceeds with the next operation.

Also, in the IEC 61499 channel implementation, preemption is enabled only through providing event to the Preempt input event. Unlike SystemJ where the compiler checks for the code and detects channel communication deadlock, the IEC 61499 channel implementation lacks similar feature, making it more prone to programmer-induced deadlocks compared to SystemJ. In this case, preemption comes in handy by allowing the termination of ongoing rendezvous, especially in the IEC 61499 channel implementation where at this stage, there is no provisions to detect any programmer-induced deadlocks during design time.

Also, in channel FBs, preemptions of ongoing or uninitiated rendezvous can be triggered through the Preempt event input

which is available on the channel FBs. Preemption which is triggered by either the sending or receiving side will trigger preemption on their respective partner, as shown in Listing 1 and Listing 2. Referring to Fig.4 and Fig.5, InputChannelInstance and OutputChannelInstance FB go to the Preempt state upon receiving Preempt event or if the partner's preemption status is higher than zero (as indicated by the state transition when the condition of "Part_preempted_localcopy > 0" in Fig.3 and Fig.4). When these function blocks are at the Preempt state, the preemption status of the corresponding channel FB (InputChannelInstance or OutputChannelInstance) is incremented and the copy of this status' value is transmitted to the partner, so the partner can react accordingly. In this state, communication statuses and data buffer are reset, and the function block returns to the initial (START) state. If the partner's side receives trigger for preemption, the partner preemption status is then transmitted to the partner which then can react to it accordingly. In the function block ECC, the received partner's preemption status is copied into the Part_preempted_localcopy variable. If communication needs to be reestablished, the appropriate trigger needs to be provided again to the InputChannelInstance of the receiving side or OutputChannelInstance of the sending side through the REQ event input.

Another difference in channel semantics between SystemJ and IEC 61499 is that, in SystemJ, a particular channel (input or output) can only be used by one particular CD and this property is checked and retained by the SystemJ compiler. However, such checking is not available in the IEC 61499 adaptation of channel. At this stage, there is no restricting mechanisms which would prevent more than one FBs to use the same channel FB. While sharing channel FBs for multiple requesting parties (i.e., FBs) can have its own merits, one complication due to potentially shared channel function blocks is to ensure fairness in accessing the shared channel function blocks. For instance, in extreme cases, one particular function block can use the channel function block and initiate a rendezvous which takes (unlimited) long time to complete, thus other requesting function blocks would be deprived access to the channel function block (*starvation*).

To completely avoid the potential issue of starvation, one could simply choose to allocate one channel FB exclusively to one party (FB) that will use it. Mitigating starvation in shared channel FBs will not be discussed in this paper and will be one of potential future works. Still, in shared channel FB, it is crucial that the same channel function block should only be accessed (used) by one party at a time.

To demonstrate how such access control can be included in AdptdInChan & AdptdOChan FBs, this paper presents an example on how an access control mechanism can be introduced to resolve access to potentially shared channel FBs. Both AdptdInChan and AdptdOChan are encapsulated into two composite function blocks named PermAdptdInChan and PermAdptdOChan, respectively. These composite function blocks include a built-in access control mechanism which permits only one function block (defined by the identifier/name of the function block and the identifier of the IEC 61499 application/program the function block belongs) to use the AdptdInChan and AdptdOChan function block at a time. A

snapshot showing the access control mechanism inside the composite function block is shown in Fig.6, which however shows only the AdptdInChan FB inside the PermAdptdInChan FB. The PermAdptdOChan FB has identical composition, except that it has the AdptdOChan FB instead of AdptdInChan FB.

Fig. 6. PermAdptdInChan composite function block

The PermAdptdInChan and PermAdptdOChan have the ChannelPermission FB which is responsible for granting access to the requesting party on a first-come first-served basis. Upon receiving REQ event, if the AdptdInChan or AdptdOChan doesn't have ongoing rendezvous, the ChannelPermission generates the ChannelFree event as a notification that the request to initiate rendezvous (i.e., communication) is accepted and the AdptdInChan or AdptdOChan immediately starts the rendezvous process. Otherwise, the ChannelInUse event is generated and the requester should make another attempt in future. It is also possible to simply query if any ongoing rendezvous applies through the Query event input. In this case, if no ongoing rendezvous occurs when the query is made, the ChannelPermission FB still generates the ChannelFree event, but rendezvous is not initiated.

Based on both PermAdptdOChan and PermAdptdInChan FBs or the AdptdOChan and AdptdInChan FBs (with some extra effort by programmers to add access control mechanism on their own), users only need to connect these function blocks to their preferred communication network function blocks, e.g., the ones considered in the Section II, therefore introducing the extra guarantee as defined by SystemJ formal semantics.

IV. USE CASE EXAMPLE

To demonstrate how the channel function blocks are used, this section shows a use case example from automation domain which utilizes the channel function blocks. This article chooses the dual-cylinder example used in [15], extended with additional scenarios of enabling temporary suspension and resumption of the cylinder movement by human operators when needed, e.g., if a faulty condition has been detected. While it is possible to extend the scenario further to include more cylinders to showcase more complex scenarios, the dual cylinder example is already sufficient for demonstrating the use of the channel function blocks while still being relatively easy to illustrate in this article.

In brief, this example consists of two cylinders, one horizontal and one vertical, which move in linear motion. Fig.7 shows a graphical illustration of the example (left, taken from

[15]) with the Human Machine Interface (HMI) designed using NXTStudio (right). In the example, when a workpiece has been placed in front of the pusher and detected by the workpiece arrival sensor, the cylinder must actuate to transport the workpiece to the destination sink, and then return to its original state. Such cylinder system needs to have coordinating mechanism to prevent possible collisions between the pushers. To achieve this, a token-based coordination is introduced, where the cylinder only actuates when a workpiece is detected by the workpiece sensor and it has the token. A cylinder gives the token to the other cylinder after it has finished its operation of moving the workpiece (i.e., the pusher extends and then comes back to the initial position) or if no workpiece has been placed and detected by the workpiece sensor. The command to suspend and resume the motion of the pusher can be given by the operator through pushing the two grey-coloured buttons on the HMI. The workpiece sensors are emulated through the cyan-coloured buttons, each corresponds to the sensors in front of the horizontal and vertical pushers, respectively.

Fig. 7. Dual cylinder example and its HMI in NXTStudio.

The software controller and plant of the dual-cylinder system are run on two different IEC 61499 programs named APP1 and APP2, each corresponds to each cylinder and deployed on separate PLCs (emulated with NXTStudio SoftPLC). Reliable communication needs to be employed to ensure that the dual-cylinder system operates and behaves as required. For example, reliable communication has to be established to ensure that the token can be exchanged successfully between cylinders. Also, reliable communication is required to ensure that the command to suspend and resume the cylinder operation reaches the destination successfully. Hence, the use of the proposed channel function blocks fits perfectly to satisfy the above requirements.

For the purpose of demonstration, the two composite FBs are created, namely the InputChannelNxU and OutputChannelNxU. Since the use case example is in multiple distributed IEC 61499 programs setting, the aforementioned composite FBs are composed from PermAdptdOChan and PermAdptdInChan FBs connected with CSIFBs which implement physical communication (can also be seen as the equivalent of SystemJ links). Note that despite the use case example has multiple IEC 61499 programs setting, the channel FBs are also compatible for single IEC 61499 program setting to guarantee communication between FBs (and where the use of CSIFBs is not necessary), whether the connected FBs are deployed on the same or on distributed controller(s).

In this example, the UDP/IP protocol of the NXTStudio's NetIO FB is chosen as the physical communication method. Due to the use of permissioned channel FBs

(PermAdptdOChan and PermAdptdInChan FBs), the InputChannelNxU and OutputChannelNxU FBs require inputs which specify function block(s) that utilize them. In this example, each InputChannelNxU and OutputChannelNxU is used exclusively by one requesting FB thus the issue of starvation doesn't apply. Fig.8 shows how the OutputChannelNxU is used to pass the token from one cylinder to another and how the InputChannelNxU is used to accept the token which is passed from the other cylinder.

Fig. 8. OutputChannelNxU used to pass token to the other cylinder (top) and InputChannelNxU used to receive token from the other cylinder (bottom).

In the example, the cylinder controller in APP1 CylControl1 FB, which is a basic function block, passes the token to another by generating the PASSTKN event. This event is passed to the REQ event input of the OutputChannelNxU. The OutputChannelNxU requires network configuration to be defined in the IPPort_Pair_Server_Client data input. The UDP/IP network functionality of the NetIO FB inside the OutputChannelNxU FB is initialized once the InitiateLink event input is triggered. When the token has been sent successfully, the ChannelIsSent event is generated and then passed to the SYNCCOMPL event input to notify the CylControl1. Meanwhile, the InputChannelNxU waits to receive the token. Once the rendezvous is performed and completed, the ChannelReceivedEvt is generated by the InputChannelNxU and then passed to the PERM event input of the CylControl1 FB, where the token is successfully passed to the corresponding cylinder.

The token-exchange scenario is an example of how a channel is realized using the InputChannelNxU and OutputChannelNxU to achieve synchronization between cylinders. In addition, this example includes the scenario of forwarding the command to suspend and resume the cylinder motion from one IEC 61499 program to another in order to show how the channel function blocks can be used to exchange data. The command sent from the HMI is received by the CAT function blocks in APP1, which will be forwarded to APP2 using the channel FBs. Fig.9 shows how the suspend command is forwarded from APP1 using OutputChannelNxU and then received by InputChannelNxU in APP2.

Fig. 9. Using OutputChannelNxU for sending (top) and InputChannelNxU for receiving (bottom) when data is included.

There is a slight difference between the token-passing and the command forwarding scenarios in using the channel function blocks in this example. In the token-passing scenario, the CylControl1 synchronizes with the OutputChannelNxU, i.e., waits for the OutputChannelNxU to complete rendezvous and generates the ChannelSent, while the command forwarding scenario doesn't employ similar synchronization. In the latter, the requesting function block will not receive notifications whether the rendezvous is completed or preempted (e.g., by other FBs or due to preemption occurring at the partner's side). This also shows another difference between channel in SystemJ and the IEC 61499 adaptation of channel, where SystemJ reaction will be blocked when the control point hits a send and receive statement until the rendezvous is completed, while in the IEC 61499 adaptation of channel, only the channel function blocks are blocked when rendezvous occurs. To achieve synchronized communication similar to the token passing scenario, the requesting function blocks will need to synchronize with the channel function blocks through the ChannelSent or ChanPreempted event outputs.

V. CONCLUSIONS

This paper proposes adoption of the channel communication mechanism used in the formal language SystemJ into the IEC 61499. This allows for network protocol-agnostic reliable communication between concurrent software entities underpinned by the formal semantics of SystemJ. This paper looked into the channel semantics and developed new IEC 61499 function blocks, based on these semantics, which allow the use of channels in IEC 61499. An automation-domain use case has been presented to demonstrate how these function blocks can be utilized. Besides being used for communication within IEC 61499 program, the presence of channels opens the gate of exchange of data between IEC 61499 and SystemJ program entities, which potentially creates a new perspective of what kind of applications can be tackled by using the power of combined tools, essentially extending them to the cousins rather than competitors relationship [15]. This remains a major future work as it addresses important issues of developing and deploying complex systems (or even

systems of systems) in distributed settings. This paper also leaves open questions of utilisation of shared channels, as well as of the channel function blocks in the context of dynamic/reconfigurable automation systems.

ACKNOWLEDGMENT

The authors would like to thank nxtControl GmbH for providing free license of NXTStudio software and also for their technical support.

REFERENCES

- [1] K. H. John and M. Tiegelskamp, IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids: Springer Publishing Company, Incorporated, 2010.
- [2] V. Vyatkin, "The IEC 61499 standard and its semantics," IEEE Industrial Electronics Magazine, vol. 3, pp. 40-48, 2009.
- [3] HOLOBLOC. (2018, 15 April). FBDK 3- The Function Block Development Kit. Available: <http://www.holobloc.com/fbdk3/index.htm>
- [4] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sunder, A. Valentini, et al., "Framework for Distributed Industrial Automation and Control (4DIAC)," in Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on, 2008, pp. 283-288.
- [5] nxtControl. (2018, 15 April). nxtControl - nxtStudio. Available: <http://www.nxtcontrol.com/en/engineering/>
- [6] A. Veichtlbauer, M. Parfante, O. Langthaler, F. P. Andrn, and T. Strasser, "Evaluating XMPP communication in IEC 61499-based distributed energy applications," in 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), 2016, pp. 1-8.
- [7] F. Weehuizen and A. Zoitl, "Using the CIP Protocol with IEC 61499 Communication Function Blocks," in 2007 5th IEEE International Conference on Industrial Informatics, 2007, pp. 261-265.
- [8] W. Leonhardsberger and A. Zoitl, "Using EtherNet/IP with IEC 61499 Communication Function Blocks," in Holonic and Multi-Agent Systems for Manufacturing, Berlin, Heidelberg, 2011, pp. 39-49.
- [9] G. Morn, F. Prez, D. Orive, E. Estvez, and M. Marcos, "IEC61499 Service Interface Function Blocks to Access Decentralized Peripherals with Profibus DP," IFAC Proceedings Volumes, vol. 42, pp. 882-887, 2009/01/01/ 2009.
- [10] M. V. Garca, E. Irisarri, F. Prez, E. Estvez, and M. Marcos, "An Open CPPS Automation Architecture based on IEC-61499 over OPC-UA for flexible manufacturing in Oil&Gas Industry," IFAC-PapersOnLine, vol. 50, pp. 1231-1238, 2017/07/01/ 2017.
- [11] S. A. Sierla, J. P. Peltola, and K. O. Koskinen, "Real-time middleware for the requirements of distributed process control," in INDIN '05. 2005 3rd IEEE International Conference on Industrial Informatics, 2005., 2005, pp. 1-6.
- [12] S.-H. Leitner and W. Mahnke, "OPC UA-Service-oriented architecture for industrial applications," ABB Corporate Research Center, 2006.
- [13] W. Dai, V. Vyatkin, J. H. Christensen, and V. N. Dubinin, "Bridging Service-Oriented Architecture and IEC 61499 for Flexibility and Interoperability," IEEE Transactions on Industrial Informatics, vol. 11, pp. 771-781, 2015.
- [14] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," Computer Languages, Systems, & Structures, vol. 36, pp. 317-344, 2010.
- [15] R. Sinha, V. Vyatkin, Z. Salcic, and H. J. Park, "Competitors or Cousins? Studying the parallels between distributed programming languages SystemJ and IEC61499," in Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA), 2014, pp. 1-7.
- [16] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," PhD, Department of Computer Science, Stanford University, California, 1984.