Controlling concurrent events in IEC 61499 based systems on FPGAs

Martin Resetarits
Automation and Control Institute
TU Wien
Vienna, Austria
https://orcid.org/0000-0003-4788-2664

Martin Melik Merkumians*
Automation and Control Institute
TU Wien
Vienna, Austria
https://orcid.org/0000-0001-7461-1793

Georg Schitter
Automation and Control Institute
TU Wien
Vienna, Austria
https://orcid.org/0000-0002-8746-5892

***Abstract*—IEC 61499 is currently explored at a rapid pace, although only few studies were conducted on the role of FPGAs for automation systems based on IEC 61499. With the necessary time quantization for synchronous FPGA systems, a new problem arises – the phenomenon of simultaneous events. Additionally, fan-ins of events complicate the requirements for the system, as it must be ensured that all arriving events are considered. This work proposes an implementation for concurrent events for FPGAs in the context of IEC 61499. To overcome the described problem, the incoming events are split into several single bit signals and are stored in a ring buffer. The proposed approach is verified in simulations of typical situations, testing the behavior of the system with respect to simultaneous events.**

***Index Terms*—IEC 61499, automation, FPGA, VHDL, Events**

I. Introduction

Modern industry systems and Cyber-Physical Systems (CPSs) are increasingly focusing on distributed automation [1]. To model such systems, the IEC 61499 provides an event-driven modeling paradigm based on Function Blocks (FBs) [2]. A big advantage of this approach is that a system can be spread over different devices with different architectures, such as PCs, Programmable Logic Controllers (PLCs), or Field Programmable Gate Arrays (FPGAs). Several studies have shown that it is practical to distribute systems on several devices [3]–[9].

Some efforts were made to run IEC 61499 systems directly on FPGAs, and an increase in performance was reported. Internal execution is split into receiving events, executing algorithms, and sending resulting events and data, and a buffer is used for the arriving events, which holds the information until it is processed by the following parts. [10], [11]

While [11] uses a bus system to transport events and data, [10] flattens the system and connects the different components directly.

Although FPGAs seem to perfectly fit the highly parallel structure of FBs, new problems need to be solved. Synchronous FPGA programs quantize time, leading to the possibility of simultaneous events. The possible overload of the event queue due to too many events has been discussed [12], but the problem is different on FPGAs, as on typical General Purpose Processor (GPP) instructions are executed sequentially, ensuring events occur one after another.

Furthermore, IEC 61499 allows the fan-in of events, which means that a single event input can be triggered by several FBs [2]. This can lead to the phenomenon that, in a single time step, a specific event is triggered multiple times. These event can be grouped and handled as a single event, or each event trigger is handled as an independent event. The standard does not describe what the expected behavior is in such cases. Neither event storms [12], nor concurrent events have been considered in [10], [11].

The contribution of this paper is a general programming paradigm for the IEC 61499 targeting FPGAs. This paradigm is introduced in Section II. In Section III the paradigm is utilized to implement a sample Basic Function Block (BFB), and the behavior and timings are simulated and discussed. Section IV concludes the work and provides an outlook for future work.

II. Proposed Programming Paradigm

The IEC 61499 defines an 8-step sequence for the execution of FBs for a received input event [13]. This sequence can be divided into two phases: the first part is responsible for receiving the incoming event and preparing the FB for execution, while the second part is responsible for processing of received data and sending output data and events to potentially connected FBs.

In the first phase, the event buffer (see Section II-A) accepts events, updates relevant data, and notifies the Event Execution Control (EEC). A second entity, named Execution Unit (ExU), is then responsible for the actual computation of the event with its data. Based on current state, internal functionality, and received event, specific algorithms are triggered and new output events are generated.

The events for the communication between multiple FBs are impulses without duration. Single-bit wires for events are used to implement similar behavior in a synchronous Very High Speed Integrated Circuit Hardware Description Language (VHDL) architecture. The value is set to `active` for one clock cycle for an event. If the same event occurs two times successively, the wire is `active` for two cycles.

* Corresponding author

A. *The Buffer*

A buffer is needed to save multiple events when they occur simultaneously. As events are spent after they are executed, they can be discarded, and memory can be reused for a later event. To exploit this behavior, a ring buffer is used.

IEC 61499 defines that data and events are related, but the data port input may change before the event is executed, so the data must be saved too. Only related data are updated, new but unrelated data are not considered. Although this is considered bad programming [14], a FB's algorithm can change the FB interface's input values. Therefore, the ExU needs to store the current data state, while the buffer saves all related data at the time of the event. When the event is processed, ExU updates only the data that are changed by the event. This increases the amount of memory needed, but simplifies the structure greatly.

Further, the implementation of the buffer must be able to write multiple events into the buffer in a single clock tick. This behavior can be achieved in VHDL by using a loop statement inside a process (see Fig. 1).

When the number of supported parallel write operations increases, the complexity of a synthesized circuit increases. To make an estimation, the cyclomatic complexity [15] for each memory slot is calculated and then summed up. This leads to to Eq. (1), with Table I showing the results for the first numbers. Based on this equation a computational complexity of $\mathcal{O}(n)$ can be estimated for the parallel writes. It should be noted that the true amount of resources used for the circuit depends on the hardware and tool chain used.

$$\begin{aligned}&\sum_{i=n_{ins}-(n_{parWrites}-1)}^{n_{ins}} i+1 = \\ &n_{ins}\cdot n_{parWrites} + \frac{3\cdot n_{parWrites}}{2} - \frac{n_{parWrites}^2}{2} = \\ &n_{parWrites}\cdot\left(n_{ins}+\frac{3}{2}-\frac{n_{parWrites}}{2}\right) \qquad (1)\end{aligned}$$

TABLE I: Sum of Cyclomatic Complexity for each memory slot.

number of	Number of input events						
parallel writes	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	-	5	7	9	11	13	15
3	-	-	9	12	15	18	21
4	-	-	-	14	18	22	26
5	-	-	-	-	20	25	30

Two general solutions exist if the buffer must be able to observe multiple events of the same kind from earlier FBs during one tick of the clock. Either all received events are processed as one event, or, if it is considered relevant how often the event is triggered, each event must have an input on its own, and the buffer stores them separately. In the first case, a logical OR of all events of the same type is enough. The latter concept is more complex and is shown in Fig. 2.

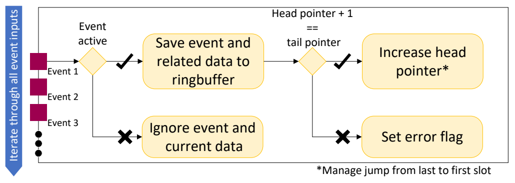
Iterate through all event inputs
Event 1
Event 2
Event 3
Event active
Save event and related data to ringbuffer
Ignore event and current data
Head pointer + 1 == tail pointer
Increase head pointer*
Set error flag
*Manage jump from last to first slot

Fig. 1: Visualization of the process to save incoming events into the buffer. To mark an overflow, an error flag is used.

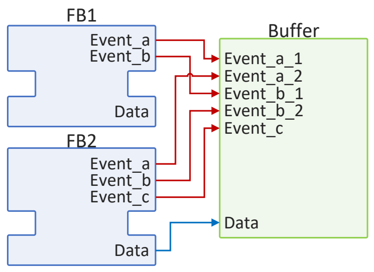

Fig. 2: Ring buffer event input wires. `FB1` and `FB2` both have an event output `Event_a`. Each one is connected to its own one-bit wire (red) and the buffer considers both as event input of the type `Event_a`.

B. *The Execution Unit*

The ExU combines EEC and the execution of internal algorithms (cf. [16]). For that, ExU communicates with the buffer in a provider/consumer manner, where the ExU is the consumer, controlling the buffer via the Tail Pointer (TP). On request of ExU the buffer provides the data and event of the next queued buffer entry to be consumed.

In case of a BFB, the EEC refereed in Step 4 is a state machine, called Execution Control Chart (ECC) in IEC 61499, and can be implemented as such [13], [17]. The BFB is of special interest in the context of this work, as it directly encapsulates functionality inside IEC 61499. The Simple Function Blocks (SFBs) can be considered as a reduced form of a BFB when no complex ECC is required, removing the ECC altogether and mapping each input event to one algorithm.

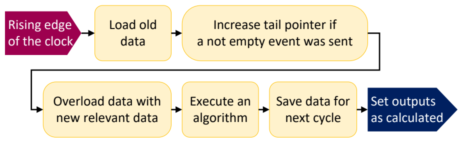
Rising edge of the clock
Load old data
Increase tail pointer if a not empty event was sent
Overload data with new relevant data
Execute an algorithm
Save data for next cycle
Set outputs as calculated

Fig. 3: Concept of the execution algorithm, which depends on the current state and the received event.

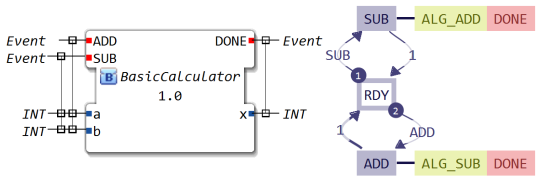

Fig. 4: A BFB that performs the operation $x = a + b$ or $x = a - b$, depending on whether the ADD or SUB event is triggered.

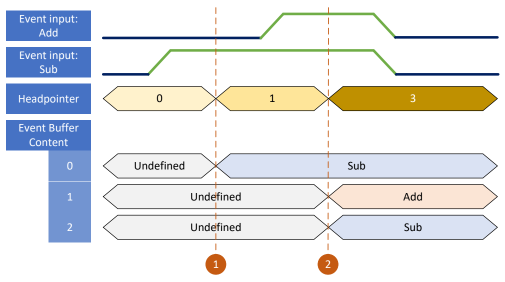

Fig. 5: Waveform of the buffer when three events arrive. Marker 1: the first event is read and the SUB event is saved into memory slot 0, and the head pointer is incremented. Marker 2: two events arrive simultaneously, and are saved into memory slots 1 and 2. The head pointer is incremented by two to three.

Other available FB types are aggregating several other FBs to describe their internal algorithms (Composite Function Block (CFB)) [13], or are responsible for the interaction with components outside the scope of IEC 61499, such as hardware-access (Service Interface Function Block (SIFB)) [18].

The ExU is responsible for updating the data according to previous events. After the new data are loaded, a state change may be triggered and the event processing starts. Finally, the out-ports are set and TP is increased. This process is triggered every clock cycle, but the TP is only incremented if the buffer is not empty (see Fig. 3).

III. Experiments

To evaluate the event and data flow of the proposed concepts, two systems were simulated with Xilinx's Vivado©Design Suite. Experiments focus on when and how events are forwarded and executed by components of the system. For the simulation, the buffer size is set to four and all four memory slots can be written in a single clock cycle. To distinguish between the buffer write phase and the execution phase, the buffer reads from the previous FBs at the falling clock edge and the execution of an event is triggered at the rising clock edge.

First, a calculator with two event inputs is tested to show the behavior of a BFB (see Fig. 4). It adds or subtracts the two data inputs according to the event it receives. In this simple design, it is easy to observe the communication and execution of the buffer and the ExU.

Specific sequences are chosen to show how the events are executed when they arrive one after another and when they arrive at the same time.

The second simulation shows the routing of an event to a system with nested FBs.

A. Simulation

The waveform shown in Fig. 5 shows the signals of a buffer when 3 events arrive. At Marker 1 the SUB Event is received and registers, saving the corresponding data in memory slot 0. At Marker 2, both events ADD and SUB are received simultaneously. The general order of execution of these events is ambiguous, but the buffer imposes an order that can influence the system's results. As the IEC 61499 does not define a correct behavior for such a situation, this issue cannot be resolved in a general way.

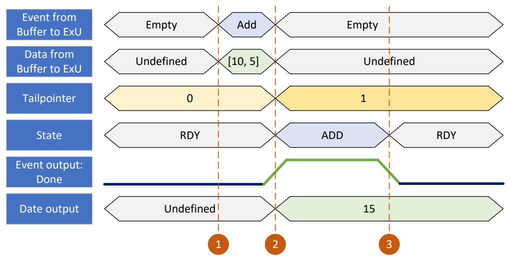

Fig. 6: Communication between the buffer and ExU. Marker 1: event and data arrives at the buffer and is saved to slot 0. As the TP points to slot 0, event and data is forwarded to the ExU (see first row). Marker 2: event and data are read, the state of ExU changes to adding, and the TP is incremented. The buffer forwards the content of slot 1. Marker 3: the processing of the event is finished. The event output is set to low and the next event is read. No event is available, so the state is set to Ready.

The waveform shown in Fig. 6 shows the communication between the buffer and the ExU. Marker 1 is placed at a falling clock edge, reading events, while at Markers 2 and 3 a rising edge occurs, when the buffer contents are read by ExU. The outputs of the FB are read at the falling edge, between the Markers 2 and 3.

B. Composite Function Blocks

CFBs act as containers for other FBs forming reusable subroutines. Higher-level subroutines can be created by creating CFB with multiple layers of CFBs. To achieve the same behavior regarding multiple events and event fan-in, CFBs also needs the proposed ring buffers, as described in Section II-A. But this will delay the event by one clock cycle to reach the FB which finally executes the event. As an example, the previous

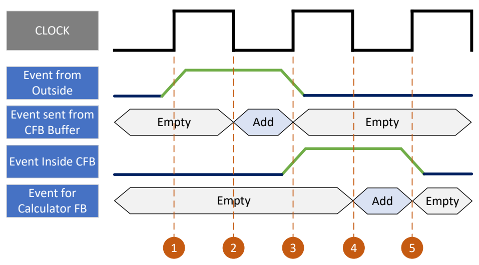

Fig. 7: Waveform showing delay of Composite Function Block.
Marker 1: Event arrived at the CFB and is saved into the buffer. Marker 2: At the next falling clock edge the ExU of the CFB reads from its ring buffer and propagates the event further (Marker 3). Marker 4: The event is read by the buffer of the `Calculator`. Marker 5: The event arrives at the ExU of the `Calculator` FB and is executed.

introduced calculator is placed inside a CFB as only element. The interface of the CFB is similar to the calculator and the event and data is connected directly to the corresponding inner ports. This behavior is shown in Fig. 7.

In summary, it has been shown, that IEC 61499 FBs can be translated to VHDL and executed on FPGAs. Due to FPGAs synchronous execution semantics, special care must be taken to ensure the correct execution of events and their associated algorithms. The buffer and ExU components presented ensure that each received event/data pair is preserved and executed for IEC 61499 FBs, enabling that FPGAs can be targeted by IEC 61499 systems.

IV. Conclusion and Future Work

This work proposes a programming paradigm to implement IEC 61499 systems on FPGAs, focusing on the event handling and the possibility of simultaneous events, which becomes possible due to the time quantization needed for synchronous FPGAs. To solve this problem, a buffer is introduced to handle events when they arrive simultaneously. The execution of the event and its associated algorithms is separated into another entity, and the simulations show the feasibility of the proposed paradigm.

This paper also highlights the problem of order for simultaneous events, which is not considered in IEC 61499. Although event execution order influences the behavior of the system, there is no way to determine the chronological order when two events arrive simultaneously. This affects FPGAs in particular, since they quantize time into clock cycles.

The next step to deploy IEC 61499 systems on FPGAs is the evaluation of the proposed ring buffer, as it takes a lot of resources. The determination of the correct buffer size is a complex problem and strongly depends on the system. Implementations on GPPs, like Eclipse 4diac [19] use a centralized buffer to handle events. The use of one buffer for several FBs limits the execution of events to one event per clock cycle. Although this leads to a decrease in performance, this can decrease resource consumption. This trade-off must be further studied to determine its usefulness.

References

[1] G. Lyu and R. W. Brennan, "Towards IEC 61499-based distributed intelligent automation: A literature review," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 4, pp. 2295–2306, apr 2021.
[2] I. TC65/WG6, *IEC 61499: Function blocks for industrial-process measurement and control systems – Parts 1 to 4*. Geneva: International Electrotechnical Commission (IEC).
[3] S. Olsen, J. Wang, A. Ramirez-Serrano, and R. W. Brennan, "Contingencies-based reconfiguration of distributed factory automation," *Robotics and Computer-Integrated Manufacturing*, vol. 21, no. 4-5, pp. 379–390, aug 2005.
[4] R. Brennan, P. Vrba, P. Tichy, A. Zoitl, C. Sünder, T. Strasser, and V. Marik, "Developments in dynamic and intelligent reconfiguration of industrial automation," *Computers in Industry*, vol. 59, no. 6, pp. 533–547, aug 2008.
[5] P. Leitão, A. W. Colombo, and S. Karnouskos, "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges," *Computers in Industry*, vol. 81, pp. 11–25, sep 2016.
[6] N. Cai, M. Gholami, L. Yang, and R. W. Brennan, "Application-oriented intelligent middleware for distributed sensing and control," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 947–956, nov 2012.
[7] P. Gsellmann, M. Melik-Merkumians, A. Zoitl, and G. Schitter, "A Novel Approach for Integrating IEC 61131-3 Engineering and Execution Into IEC 61499," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 8, pp. 5411–5418, 2021.
[8] M. Melik Merkumians, P. Gsellmann, and G. Schitter, "Hierarchization and Integration of IEC 61131-3 and IEC 61499 for Enhanced Reusability," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021)*, 2021, pp. 1–4.
[9] M. Melik-Merkumians, T. Baier, M. Steinegger, W. Lepuschitz, I. Hegny, and A. Zoitl, "Towards OPC UA as portable SOA middleware between control software and external added value applications," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, 2012, pp. 1–8.
[10] H. Pearce and P. Roop, "Synthesizing IEC 61499 function blocks to hardware," in *2019 International Conference on Electronics, Information, and Communication (ICEIC)*. IEEE, jan 2019.
[11] D. O'Sullivan and D. Heffernan, "VHDL architecture for IEC 61499 function blocks," *IET Computers & Digital Techniques*, vol. 4, no. 6, pp. 515–524, nov 2010.
[12] D. Pescha and M. Horauer, "Event storms in IEC 61499 applications," in *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*. IEEE, nov 2018.
[13] A. Zoitl, *Modelling control systems using IEC 61499*, 2nd ed., ser. IET control engineering series 95. Stevenage, Herts, United Kingdom: The Institution of Engineering and Technology, 2014.
[14] M. Wenger and A. Zoitl, "Re-use of IEC 61131-3 structured text for IEC 61499," in *2012 IEEE International Conference on Industrial Technology*. IEEE, March 2012, pp. 78–83.
[15] T. McCabe, "A complexity measure," vol. SE-2, no. 4, pp. 308–320.
[16] A. Zoitl, *Real-Time Execution for IEC 61499*. ISA, 2008.
[17] P. Gsellmann, M. Melik-Merkumians, and G. Schitter, "Comparison of Code Measures of IEC 61131–3 and 61499 Standards for Typical Automation Applications," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA 2018)*, vol. 1, 2018, pp. 1047–1050.
[18] M. Melik-Merkumians, M. Wenger, R. Hametner, and A. Zoitl, "Increasing Portability and Reuseability of Distributed Control Programs by I/O Access Abstraction," in *Proceedings IEEE Emerging Technologies and Factory Automation (ETFA 2010)*, 2010.
[19] Eclipse. 4diac - Framework for Industrial Automation & Control. [Online]. Available: https://eclipse.org/4diac/