FBBeam: An Erlang-based IEC 61499 Implementation

Laurin Prenzel
Technical University Munich
Munich, Germany
laurin.prenzel@tum.de

Julien Provost
Technical University Munich
Munich, Germany
julien.provost@tum.de

***Abstract*—The IEC 61499 is a modeling language for distributed control systems. Despite numerous research results existing on this topic, industry acceptance is lacking. This paper aims to investigate the benefits of reusing an existing soft real-time runtime system for the implementation of the IEC 61499. For this purpose, *FBBeam*, a compiler that automatically converts IEC 61499 models to Erlang source code, was implemented. Possible execution semantics are presented and compared to the Erlang execution model. An initial case study examines the scalability of a multi-tasking runtime environment. The results indicate that Erlang is able to utilize multiple CPU cores efficiently and can distribute the load dynamically. *FBBeam* represents an opportunity to reutilize an existing runtime environment for research on dynamic updating, distribution, monitoring, maintenance, and fault-tolerance for *Industry 4.0* or *Cyber Physical Production Systems*.**

***Index Terms*—Erlang Runtime System, Asynchronous Execution Semantics, Implementation and Evaluation, Multi-tasking**

I. Introduction

The IEC 61499 has been implemented numerous times: both by academics and by industry. Given the ambiguities identified by multiple research papers, there is no shortage of ways to implement it [1, 2]. Unlike other implementations, the solution described in this paper reutilizes a runtime environment from the telecommunications industry that has been used for decades. Re-utilizing proven technology from one industry may effectively solve the problems of another.

At its inception, the IEC 61499 was developed to add a layer of execution semantics to the languages defined by the IEC 61131-3 to facilitate distribution, flexibility, and dynamic reconfiguration [3]. The distributed and event-driven architecture allows for flexible systems to tackle the challenges of the next decades, such as Cyber Physical Production Systems and Industry 4.0 [4]. The introduction of component-level distribution was intended to reduce the communication overhead, while allowing a more fine-grained dissemination [5]. A major advantage of the IEC 61499 standard is the encapsulation of functionality in software components without global state. Not only does encapsulation facilitate reusability, it allows the modification of components without causing issues with seemingly unrelated subsystems [6, 7], and it enables the care-free dissemination of components over networks and resources, thus permitting distribution. In addition, the model-based approach lends itself to formal verification [8]. On the other hand, there are a number of design and execution ambiguities preventing the IEC 61499 standard from being fully accepted by industry [2].

Erlang, on the other hand, is a programming language built for the needs of the telecommunication industry. It was specifically developed "to provide a better way of programming telephony applications". As such, the requirements of concurrency, scalability, distribution, and dynamic reconfiguration were vital at the start of the project in 1986 [9]. Erlang uses a virtual machine that allows fast context switches and is mostly used in application areas such as telecommunication, instant messaging, and FinTech [10]. Recently, there have been efforts to introduce Erlang to the control of embedded systems with real-time constraints [11, 12].

In contrast to the industry-driven development of Erlang, industry acceptance of the IEC 61499 is lacking, despite the advantages the standard offers. Requirements that led to the development of the IEC 61499 were also considered during the development of Erlang: e.g. encapsulation, distribution, and dynamic reconfiguration. As a consequence, Erlang fits nicely together with the IEC 61499. In addition, Erlang is proven technology with years of experience in industrial applications and a mature ecosystem including deployment, monitoring, and debugging tools. Prototypical implementations of an IEC 61499 implementation in Erlang without automatic code generation were presented in [7, 13]. This paper introduces *FBBeam*, a compiler that converts IEC 61499 models automatically into Erlang source code. The reasoning for the chosen execution semantics is explained, the multi-tasking scalability is demonstrated in a case study, and benefits of reusing Erlang for the Runtime Environment are discussed.

The main benefit of the implementation described in this paper is the reuse of a highly scalable, multi-tasking runtime environment. This allows further research in the directions of dynamic updating, distribution, and real-time scheduling of event-triggered systems, while profiting from the existing ecosystem. On the other hand, given that Erlang was developed for soft real-time applications, this implementation may not be deterministic enough for critical hard real-time systems.

Section II introduces the fundamentals of the IEC 61499 and Erlang. Their respective execution semantics are compared in Section III. Section IV explains the implementation and Section V presents an example system and evaluation.

II. Background

A. IEC 61499

The IEC 61499 standard is an architecture for distributed control system built on top of the languages defined by the IEC 61131-3. The IEC 61499 currently has three parts: The first part describes the architecture for distributed systems. The second part introduces requirements for software tools, e.g. a *Document Type Definition* that describes an XML exchange format. Part three is currently withdrawn and part four consists of compliance profiles for systems, devices, and software tools.

The standard defines models to specify a solution for a control problem. The main element is the function flock (FB), which serves as a software component encapsulating functionality and data. The IEC 61499 contains multiple Function Block models, for example: *Basic FBs* implement a state machine with algorithms. *Service Interface FBs* may be used to implement I/O functionality. *Subapplications* contain a Function Block network and can be used to structure an Application hierarchically.

In addition to the FB models, models of higher abstraction levels are used to describe the system and the control solution: The *Application* model contains a network of Function Blocks with the purpose of solving a control problem. The *System* model contains devices and resources that the *Applications* are mapped to by the *Distribution* model.

The implementation introduced in this paper currently focuses on the *Basic FB*, the *Subapplication*, and the *Service Interface FB*, as well as the *Application* model.

B. Erlang

Erlang is an open-source functional programming language with roots in the telecommunication industry. It was developed for distributed, highly-available, and concurrent systems. As such, an implementation of the IEC 61499 is quite a natural fit. The fundamentals of the programming language and environment are detailled in many books and online resources [14, 15]. The most important elements of Erlang are:

- The functional programming language
- The Open Telecom Platform (OTP)
- The Erlang Runtime System (ERTS)

The Open Telecom Platform (OTP) is a collection of applications and behaviours that facilitate the implementation of common system architectures.

The Erlang Runtime System (ERTS) is the virtual machine in which the compiled Erlang code is executed. Code is executed in processes, which themselves are blocks of memory encapsulating the state and protecting the virtual machine from errors. The ERTS is optimized for highly concurrent and available systems. Computation time is allocated fairly and dynamically over the currently executable processes. Executable processes receive a "time slice" measured by a reduction count (number of function calls) of 4000 before they are preempted.

The execution order of processes is defined with priorities. Processes are put into run queues according to their priority when they are ready to execute (see Figure 1). There are three

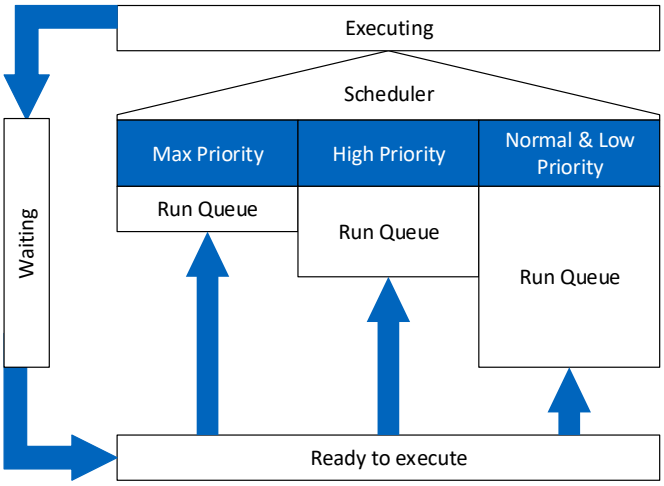

Fig. 1. Erlang scheduling and run-queue concept

run queues per scheduler: *max* and *high* priorities have their own run queues, while *normal* and *low* priority processes share the same run queue. Processes are taken from the run queue in a FIFO-manner, except for *low* priority processes, that have to reach the top of the run queue 8 times, before they are executed. The *max* run queue preempts all other run queues when processes are rescheduled, and is reserved for runtime system processes. *High* is available to the user, although it should be used carefully, because it may lead to blocking or process starvation.

A process is only interrupted if its reduction count is consumed; although it may finish early. Thus, a higher priority process is only executed after the currently running process has yielded—in the worst case after 4000 reductions. Natively-implemented functions (NIFs) that are coded in C and called from an Erlang process are not preempted by default, and can thus lead to the blocking of the scheduler.

Erlang may use more than one scheduler to allow multitasking. During startup, one scheduler may be spawned for every available CPU thread, and the load is distributed dynamically. There are two load distribution paradigms: *load balancing* and *load compaction*. *Load balancing* balances the load evenly over all available schedulers. *Load compaction* (default) fully employs the smallest number of schedulers to allow hibernation of idle schedulers. Processes are transferred between schedulers based on task-stealing and task-migration: idle schedulers steal processes from busy schedulers, and periodically, all schedulers redistribute their work according to a migration plan [16].

III. Execution Semantics

This section compares possible execution semantics of the IEC 61499. Following this, the constraints the Erlang Runtime System poses in terms of execution and scheduling are described. This opens up different choices for an IEC 61499

implementation. Finally, the chosen execution semantics of *FBBeam* are presented.

A. IEC 61499 Execution Semantics

The IEC 61499 does not strictly define the execution semantics of its models. This has led to a number of papers outlining the ambiguities [2]. Ferrarini and Veber [17] described different possible execution semantics on a theoretical level. A recent review of current IEC 61499 runtime environments is described by Prenzel et al. [1].

The main difference between the execution semantics of the standard is the scheduling of the Function Blocks. Since the FBs are event-triggered, in principle they can receive an event at any time. The scheduling mechanism must allocate the sparse computational resources to the FBs. In practice, some types of execution semantics were crystallized out of the IEC 61499:

Cyclic All FBs are executed in a fixed order in every cycle, in a similar way to the IEC 61131-3. This guarantees a fixed cycle time but introduces an overhead, since not every FB needs to be executed in every cycle. One implementation that uses this semantic is ISaGRAF [18].

Synchronous Also known as "depth-first scheduling": events are sent as synchronous function or method calls. FBDK follows the synchronous semantics [19].

Asynchronous Events are collected in one or more event queues. The events are taken from the event queues according to a scheduling mechanism, e.g. first-in-first-out, round-robin, or earliest-deadline-first. *4diac FORTE* is an asynchronous implementations [20].

All semantics have advantages and may be more suitable for one or the other application. Cycle-based implementations offer the most determinism, while asynchronous implementations offer more flexibility.

B. Erlang Execution Model

The internals of the Erlang Runtime System are sparsely documented and subject to changes and optimizations. The most thorough resource is [16]. Parts of the scheduling were already explained in Section II-B. This section focuses on the scheduling semantics from the point of view of a process.

Figure 2 shows the state machine of an Erlang process. The blue elements represent the typical cycle during execution. Processes are waiting, become runnable because of a message or a timeout, and are eventually scheduled. Messages are stored in a mailbox inside the recipient's heap, or, if the recipient is locked, in a separate heap fragment.

The processes are scheduled based on run queues, which they enter once they are *runnable*. The behavior of the run queues is described in Section II-B. Processses are taken from their run queues in a first-in-first-out order. The process may run for 4000 reduction before it is preempted, and this count includes garbage collection. Generational garbage collection per process is triggered when the combined heap and stack size exceeds the current limit for this process. Either sufficient

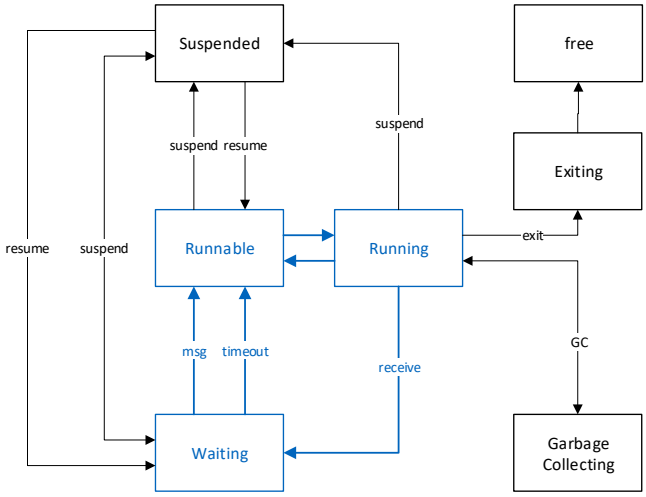

Fig. 2. State machine depicting the operation of an Erlang process [16]

garbage is cleared, or the limit is increased. Priorities affect the order in which processes are taken from the run queues. A currently running process can not be interrupted, until it has reached its reduction count or has yielded.

This type of execution is optimized for telecommunication devices. Processes communicate asynchronously and receive a fair amount of execution time, which leaves the system responsive even under high load. This behavior was observed in [13], where the reaction time of an IEC 61499 FB in Erlang under high load was analyzed. Depending on the computational effort of handling a message, a process may consume multiple messages during one scheduling without context switches.

C. Choice of Execution Semantics

As a general purpose language, Erlang allows the implementation of arbitrary IEC 61499 semantics. Nevertheless, some semantics are better suited than others. A purely cyclic execution in Erlang is suboptimal, because Erlang was made for dynamic, concurrent, and event-triggered execution. The synchronous execution works well if the application state may be stored in objects and the events are implemented as method calls to these objects. In Erlang, all data is encapsulated in processes, and message passing is asynchronous. Synchronous calls are possible (as the combination of two asynchronous calls), but block the execution of the previous process. The most native semantic for Erlang is an asynchronous implementation, as it fits the execution model of Erlang and enables multi-tasking.

Currently, the most used asynchronous IEC 61499 implementations is *4diac FORTE* [20]. The execution of *4diac FORTE* is based on the event-chain concept introduced by Zoitl [21]. An event-chain is a succession of FB executions started through an event occurrence at a FB source and ending in a FB sink. Events are fed to the beginning of this event chain, and the event chain will call all FBs along this chain. By

using this concept it is possible to apply real-time constraints to each chain and to guarantee the order of events.

In addition to the event-chain concept, Zoitl [21] discusses the disadvantages of an implementation where every FB is carried out as its own real-time task:

1) Large number of concurrent tasks
   - Limited number of tasks in common RTOS
   - High memory consumption
   - Overhead from task switches
2) Need for real-time constraints for every Function Block
3) Complicated schedulability analysis

While these constraints certainly hold for operating system tasks, Erlang is able to circumvent the first set of disadvantages: It was made for fast context switches and by default allows 262,144 concurrent processes, although this limit may be set manually to a maximum of 13 million. A newly spawned process requires 309 words of memory (<3 KB). Although there certainly is overhead from task switches, this overhead is much smaller than the overhead of switching OS tasks. Real-time constraints are unquestionably a concern, especially since Erlang was intended for soft real-time. Nevertheless, since the IEC 61499 standard does not contain the possibility of specifying real-time constraints either, they cannot be used for schedulability analysis and dynamic scheduling is the only choice available, apart from a cyclic execution.

Alternatively, an implementation similar to the event-chain concept of [21] could be achieved in Erlang if all messages were processed by an event handler first, although this would introduce overhead. Currently, the previously described fully asynchronous implementation is simpler and more intuitive for the Erlang Runtime System.

IV. Implementation

This section describes the IEC 61499 implementation of *FBBeam*. The solution is built on the experiences gained from the prototypical implementation described in [13]. *FBBeam* allows the automatic code generation from an IEC 61499 model, defined by the XML exchange format, into Erlang source code that may be executed directly within the Erlang Runtime System. After the selected execution semantics are introduced, the current limitations of the implementation are described.

A. FBBeam Execution Semantics

As characterized in Section III-C, the Erlang Runtime Environment allows the IEC 61499 to be implemented in different ways. In this paper, an implementation similar to the one in [13] was chosen. FBs are implemented as individual processes and messages are sent autonomously and directly to the recipient process. Process scheduling is handled by the Erlang Runtime System. As a result, no additional event handlers are necessary and processes will behave the same, independently of how they may be distributed. All processes share the same priority, thus executable FBs will get their fair share of execution time in a FIFO order. As shown in [13], a usual execution cycle of a FB will consume much less than the available 4000 reductions. If multiple events are in the mailbox, the FB will use all of them until it is preempted.

In addition to the FB processes, additional processes are used as supervisors, monitoring a set of processes and restarting them if they crash. Currently, the supervisors are generated according to the hierarchy present in the IEC 61499 model in the form of Subapplications. This solution also limits the effects of an update of a Subapplication to this particular supervisor. In the future, supervisors could be used for fault-tolerance, e.g. if a process crashes due to faulty source code, a faulty model, or incorrect connections.

B. Compilation

The code generation is performed in three steps. The compiler, generating the Erlang source code from the IEC 61499 XML documents, is implemented in Python 3.

1) The IEC 61499 XML exchange format files are read, parsed, and an internal representation of the IEC 61499 model is created. Python objects collect all information regarding FB instances, connections, Subapplications, and Applications.
2) The internal representation is transformed into an internal representation of the Erlang source code. FB Instances are gathered in class modules, and the information necessary to generate supervisors is gathered.
3) Finally, the internal representation is inserted in a set of templates for the Erlang source code modules. For Service Interface FBs, the source code is taken from a prepared library, and the functions defining the instances are appended.

C. Current Limitations

In its current status, *FBBeam* is used for research projects that focus on dynamic reconfiguration and real-time performance. Thus, new features are implemented when the need arises. Since many of the features of IEC 61499 are native to Erlang, the implementation may be simpler than in other general-purpose languages.

Of the models of the IEC 61499, currently only the System, the Application, and some of the FB models are used. From the Function Block models, the Basic FB, the Subapplication, and the Service Interface FB are supported. A library of Service Interface FBs is being extended continuously and currently contains an interface for Modbus TCP. Adapters are not supported yet and the Distribution models of the IEC 61499 are currently not used. The concept of distribution and inter-node communication is natively available for Erlang, but those features must be mapped to the Distribution models of the IEC 61499.

Within Basic FBs, the translation of algorithms is an open field for research. A converter from IEC 61131-3 ST code to Erlang has previously been developed, but is not yet integrated, since the conversion suffers from the conceptual differences between ST and Erlang. Alternatively, ST code could be compiled into C code, which can be executed inside the Erlang

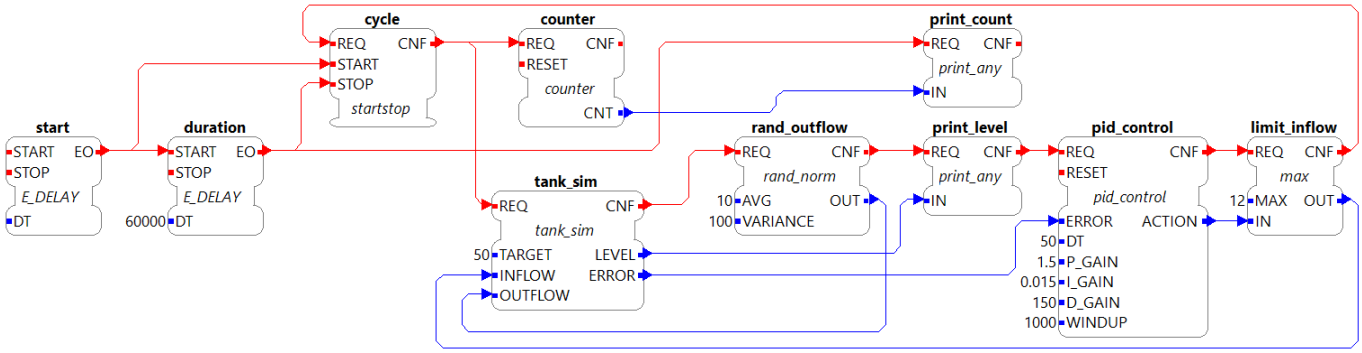

Fig. 3. IEC 61499 Application model of a tank simulation case study. The simulation is executed continuously for 60 seconds, after which the total number of executions is displayed.

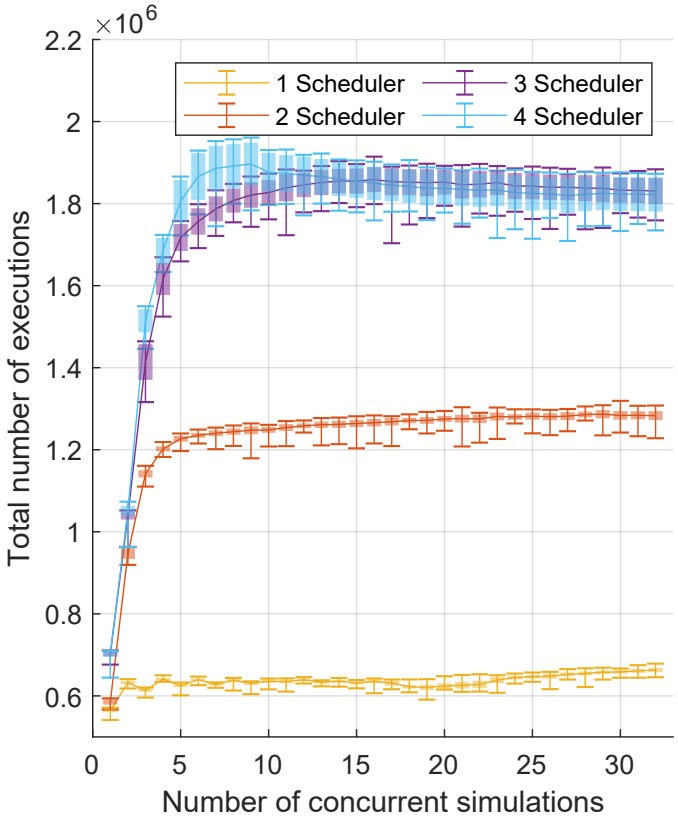

Fig. 4. Event chain executions for 1-4 scheduler and 1-32 concurrent event chains fo 60 seconds. Error bars indicate minimum and maximum values in 120 runs, the continuous line connects the average values.

Runtime System, although this circumvents all safeguards that make Erlang fault-tolerant. Currently, algorithms may be defined in Erlang itself.

V. Case Study

To demonstrate the multi-tasking capabilty of the Erlang Runtime System, a case study of a physical simulation is implemented. Figure 3 displays the *Application* model of the simulation. A PID controller is connected to a FB network simulating a physical process with a random disturbance. This simulation is executed continuously and the number of executions reached over 60 seconds is counted.

The application is modified to run up to 32 concurrent simulations with 32 PID FBs in parallel, and the simulation is run with between 1 and 4 Erlang schedulers. This experiment is repeated for 120 runs and the results are displayed in Figure 4. In total, the simulation was run for over 256 hours.

The case study indicates, that without any further optimizations, Erlang is able to distribute the workload efficiently and automatically over multiple CPU cores. With four schedulers, the application has to compete with the operating system for resources, thus the performance can only exhibit linear behavior up to 3 schedulers. Full utilization of the additional schedulers requires a larger number of concurrent simulations.

VI. Discussion

Reusing proven technology for the implementation of the IEC 61499 has multiple advantages. The Erlang Runtime System is able to support a large number of concurrent processes. It is not limited to the number of operating system threads, and allows fast context switches. As illustrated by the case study, Erlang is able to run large numbers of Function Blocks concurrently, and can efficiently employ the available CPU cores. The load is distributed dynamically and processes receive equal opportunities to handle their messages / events. When more than one scheduler is used, performance may scale almost linearly.

If real-time constraints or event rates were available, it could be possible to find a better scheduling mechanism, such as *earliest deadline first* or *rate monotonic scheduling*. Erlang is intended for dynamic soft real-time systems, not for static or cyclic hard real-time applications. Since static, cyclic hard real-time implementations of the IEC 61499 already exist [18], *FBBeam* shows how a dynamic, event-triggered, and scalable multi-tasking IEC 61499 implementation may look like.

In addition, an implementation of the IEC 61499 in Erlang allows the reuse of a proven and growing ecosystem. Recently, Santos et al. [22] demonstrated that there is a need to move the IEC 61499 ecosystem forward in terms of fault-tolerance. *FBBeam* opens opportunities for investigations into how exist-

ing frameworks of Erlang may be adapted for the IEC 61499, for example:

- Unit & system testing
- Distribution, deployment, and monitoring
- Dynamic reconfiguration
- Function Block error handling and fault-recovery

VII. Conclusion

Despite architectural advantages of the IEC 61499 as a modeling language for distributed control systems, industry acceptance is still lacking. Erlang, on the other hand, is a technology initiated and developed by industry itself, which is improved continuously. This paper aimed to outline an implementation of the IEC 61499 in Erlang and to pinpoint opportunities and limitations.

Erlang favors an asynchronous execution because of its process architecture, although an event-chain implementation, such as introduced by [21], may be feasible and advantageous in the future, especially with respect to real-time constraints. In the current implementation, all scheduling and load distribution is organized by the Erlang Runtime System.

In the author's opinion, the IEC 61499 ecosystem can benefit from many diverse implementations that may suit different purposes. If interoperability were guaranteed, an Erlang implementation may offer convenient scalability, availability, and multi-tasking in applications where soft real-time suffices. While the real-time performance as of now may not be sufficient for safety-critical industrial automation applications, this aspect could be improved in the future by introducing new scheduling paradigms to Erlang.

References

[1] L. Prenzel, A. Zoitl, and J. Provost, "IEC 61499 runtime environments: A state of the art comparison," in *Int. Conf. on Computer Aided Systems Theory*, 2019.

[2] T. Strasser, A. Zoitl, J. H. Christensen, and C. Sünder, "Design and execution issues in IEC 61499 distributed automation and control systems," *IEEE transactions on systems, man and cybernetics*, 2011.

[3] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-Art review," *IEEE Transactions on Industrial Informatics*, 2011.

[4] T. Terzimehic, M. Wenger, A. Zoitl, A. Bayha, K. Becker, T. Müller, and H. Schauerte, "Towards an industry 4.0 compliant control software architecture using IEC 61499 & OPC UA," in *International Conference on Emerging Technologies and Factory Automation*. IEEE, 2017.

[5] R. Schoop and A. Strelzoft, "Asynchronous and synchronous approaches for programming distributed control systems based on standards," *Control engineering practice*, 1996.

[6] M. N. Rooker, C. Sünder, T. Strasser, A. Zoitl, O. Hummer, and G. Ebenhofer, "Zero downtime reconfiguration of distributed automation systems: The ϵCEDAC approach," in *Holonic and Multi-Agent Systems for Manufacturing*. Springer, 2007.

[7] L. Prenzel and J. Provost, "Dynamic software updating of IEC 61499 implementation using Erlang Runtime System," in *IFAC World Congress*. Elsevier, 2017.

[8] V. Vyatkin and H. M. Hanisch, "Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems," in *International Conference on Emerging Technologies and Factory Automation*. IEEE, 2001.

[9] J. Armstrong, "A history of Erlang," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, Jun. 2007.

[10] Erlang Solutions, "Case studies & insights," https://www.erlang-solutions.com/resources/case-studies.html, 2019, accessed: 2019-5-2.

[11] "Craft and deploy bulletproof embedded software in Elixir – Nerves project," https://nerves-project.org/, accessed: 2019-1-16.

[12] Peer Stritzinger GmbH, "GRiSP – bare metal hardware for the internet of things, specially designed for Erlang," https://www.grisp.org/, accessed: 2019-1-16.

[13] L. Prenzel and J. Provost, "Implementation and evaluation of IEC 61499 basic function blocks in Erlang," in *International Conference on Emerging Technologies and Factory Automation*. IEEE, 2018.

[14] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[15] F. Hebert, *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, Jan. 2013.

[16] E. Stenman, "The BEAM book," https://github.com/happi/theBeamBook, Jan. 2019.

[17] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *Int. Conf. on Industrial Informatics*. IEEE, 2004.

[18] V. Vyatkin and J. Chouinard, "On comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations," in *Internatinal Conference on Industrial Informatics*. IEEE, 2008.

[19] "FBDK 8.0 - the function block development kit," https://www.holobloc.com/fbdk8/index.htm, accessed: 2019-4-2.

[20] A. Zoitl, T. Strasser, and A. Valentini, "Open source initiatives as basis for the establishment of new technologies in industrial automation: 4DIAC a case study," in *Int. Symposium on Industrial Electronics*. IEEE, 2010.

[21] A. Zoitl, *Real-time Execution for IEC 61499*. Instrumentation, Systems, and Automation Society, 2009.

[22] A. A. Santos, A. F. Silva, M. de Sousa, and P. Magalhães, "An IEC 61499 replication for distributed control applications," in *International Conference on Industrial Informatics*. IEEE, Jul. 2018.