Implementation and Evaluation of IEC 61499 Basic Function Blocks in Erlang

Laurin Prenzel
Safe Embedded Systems
Technical University Munich
Email: laurin.prenzel@tum.de

Julien Provost
Safe Embedded Systems
Technical University Munich
Email: provost@tum.de

***Abstract*—Despite several architectural advantages for the challenges of future manufacturing systems, the IEC 61499 standard is currently not widely accepted by industry. One advantage of the IEC 61499 is the concept of downtimeless system evolution. An extension of this, dynamic software updating, which allows switching out running processes and deal with unplanned changes, is readily available in the programming language Erlang. This paper investigates the real-time performance of an asynchronous, parallel IEC 61499 basic function block implementation in Erlang, a functional programming language with a soft real-time, concurrent runtime environment. As a result, although hard real-time performance is not guaranteed and the runtime environment is executed on top of a regular operating system, the performance is consistent and promising for future implementations and extensions.**

I. Introduction

The IEC 61499 standard was introduced as a possible successor of the IEC 61131-3 standard for industrial control systems. Despite several architectural advantages, the standard is yet to be widely accepted by the industry. The distributed and event-driven concept allows for more flexible systems to tackle the upcoming challenges of the next decades such as the *internet of things* or *industry 4.0*.

Flexibility, reconfigurability and distribution are some keywords associated with this standard [1]. The main advantage is the encapsulation of independent functionality in function blocks without global states. This feature facilitates the reuse of function blocks as modules for many different applications on different platforms. It allows the modification of the function block network without causing unexpected issues with seemingly unrelated subsystems [2] and it enables the dissemination of function blocks over large networks and resources, thus permitting real, physical distribution. In addition, the model-based approach lends itself to formal verification [3, 4]. On the other hand, there are a number of design and execution ambiguities preventing the IEC 61499 standard to be fully applied [5].

Erlang, as a functional programming language, first appeared in 1986 as a proprietary language for the use in telecommunication systems [6, 7]. Nowadays, Erlang serves as an open-source language for distributed, fault-tolerant and highly available systems with many large scale applications, mostly in the telecommunication sector.

These features alone promote Erlang for an IEC 61499 implementation. Nonetheless, Erlang exceeds in the execution of highly concurrent systems, as it is capable of sustaining thousands of synchronous processes with low memory requirements. The main benefits of using Erlang are the ability to load new code versions during runtime and the framework for safe dynamic software updates. While the IEC 61499 also allows online reconfiguration or, as it is commonly referred to, downtimeless system evolution, Erlang takes this concept further [2, 8].

This paper describes an asynchronous, parallel implementation of the IEC 61499 basic function block in Erlang and investigates the soft real-time performance in the Erlang Runtime System (ERTS). The point here is not to prove hard real-time behavior that the ERTS, as a soft real-time system, does not guarantee. The goal is to highlight a readily-available architecture for the implementation of distributed, flexible and reconfigurable systems and to analyze what performance a soft real-time system without sophisticated optimization can deliver out of the box. This would allow further research on dynamic software updating and automatic update generation and verification without the need to implement a new runtime environment.

Section II introduces basic concepts of the IEC 61499 standard and presents the main features of Erlang and the Erlang Runtime System. The function block execution model and the subsequent implementation in Erlang is described in Section III. Section IV introduces the performance evaluation procedure and Section V follows with the results that are discussed in Section VI.

II. Background

This section introduces the IEC 61499 standard and its basic function block. Afterwards, the programming language Erlang and its most important characteristics are described.

A. IEC 61499

Apart from the standard itself, Zoitl and Lewis [9] offer an extensive overview of the concepts of the IEC 61499 standard.

The main element of the standard is the function block. It can be composed in large networks by linking the event and data connections. The set of function blocks describing the solution to a control problem is bundled in an application. The

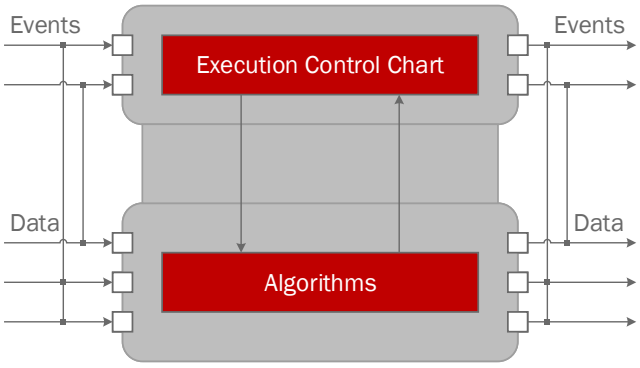

Fig. 1. IEC 61499 Basic Function Block

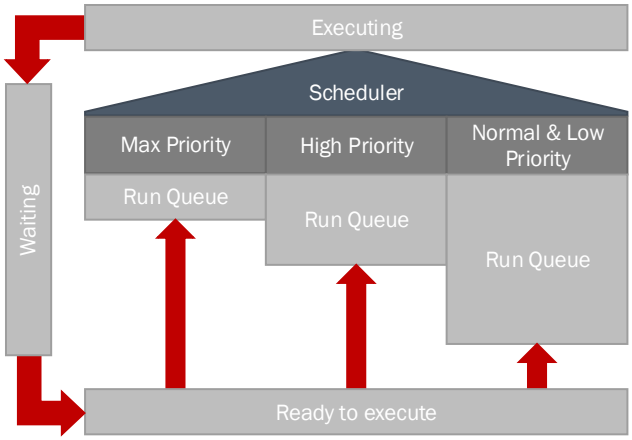

Fig. 2. Erlang process execution and run queues

individual function blocks may be distributed over multiple resources and devices with the corresponding models.

The basic function block allows the manual implementation of custom algorithms. The execution of the algorithms is conducted by the Execution Control Chart (ECC). Triggered by incoming events, the ECC requests the execution of an algorithm and issues outgoing events. A schematic view of the basic function block is presented in Figure 1.

Besides the basic function block there are three other main types of function blocks: Subapplication blocks, composite function blocks, and service interface function blocks. The first and second help structuring the application, whereas the latter serves as an interface to external resources.

There exist multiple possible implementations of the function block network. An overview is given by [10] and more recent developments are summarized by Vyatkin [11]. Common execution modes are sequential, cyclic, and parallel.

B. Erlang

Erlang is a functional programming language with roots in the telecom industry. It shines in the application of distributed, highly-available, concurrent systems. As such, a parallel implementation of the IEC 61499 fits quite naturally.

The basics of the programming language and environment can be found in many different books and online resources [7, 12]. The most important elements are:

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

The language itself is already sufficiently described by the literature. It is a functional language with immutable data and a strong focus on recursion. The compiled `BEAM` code is then executed in a virtual machine, described in the following section.

1) Erlang Runtime System: The Erlang Runtime System (ERTS) is the environment in which the compiled Erlang code is executed. It is optimized for highly concurrent (thousands of processes) and highly available systems. As such, it distributes the computation time fairly over the currently executable processes. Every executable process receives a "time slice" measured by a reduction count (number of function calls) of 4000 before it is preempted. This number may change in different versions, in the past the default value was 2000 reductions.

For every available CPU thread one scheduler is spawned at startup. This scheduler has three run queues for four priority levels. `Max` and `high` priority have their own run queue. `Normal` and `low` priority processes share the same run queue, but a `low` priority process is only executed after it reached the top of the run queue 8 times. Figure 2 displays the different run queues and the process cycle.

The `max` priority is reserved for system processes. `High` may be used, although it is discouraged and may lead to issues such as priority inversion or process starvation. In general, focusing too much on optimization is discouraged.

Between the schedulers there are two contrasting paradigms: *load balancing* and *load compaction*. *Load balancing* tries to balance the load over all available schedulers. *Load compaction* (default) tries to fully employ the smallest number of schedulers to allow hibernation of the rest.

The preemption of a processes can only occur after the reduction count or if the process finishes early. Thus, a higher priority process is only executed after the currently running process has yielded, in the worst case after 4000 reductions.

Also, natively-implemented functions (NIFs) that are coded in C and called from an Erlang function do not yield by default, and can thus lead to blocking of the scheduler.

2) Garbage Collection: The ERTS requires garbage collection. Every Erlang process has its own heap and stack growing towards each other. When there is not enough space available between them, garbage collection has to be performed. If enough space can be reclaimed, the process will resume normally, otherwise the heap size is increased. The initial heap size of a process can be set manually. If less heap size is necessary, it will eventually be resized [13].

In many other virtual environments, garbage collection is performed for the entire system, thus blocking for a long time. In Erlang, it is confined to individual processes. Thus, garbage collection can delay the execution of a process, but it will not

directly affect other processes or block the whole system.

Besides the major garbage collection, a minor, generational garbage collection can be performed. The heap is split into young and old elements. A minor garbage collection only collects the young heap and ignores the old heap. This reduces the overhead of collecting long-lived elements repeatedly. The major garbage collection will be performed after a specified number of minor runs, or if the minor garbage collection can not reclaim sufficient memory.

The garbage collector can also be triggered manually by invoking the corresponding functions. Garbage collection is one of the reasons why Erlang can not be considered hard real-time. By triggering the garbage collector manually the probability of it occurring in undesirable situations, i.e. in critical states, can be reduced.

3) Open Telecom Platform: The Open Telecom Platform (OTP) is a set of applications and behaviours that facilitate the implementation of common systems. For example, instead of manually implementing a process resembling a state machine, the OTP contains a formalized behaviour for this purpose (*gen_statem*). The behaviours supply the necessary functions to build large, interconnected systems by reusing similar patterns. Common behaviors exist for: State machines, servers, event handlers, and supervisors. Supervisors allow the starting, monitoring and graceful termination of a network of processes. In the event of a failure, supervisors may restart parts of the application to recover the functionality. It is also possible to create custom behaviours.

The OTP offers the concepts of applications and releases. Applications are sets of modules that implement a partly independent or encapsulated functionality. The set of applications to solve a particular problem can be bundled into a release. A release is a fully functional system containing its own copy of the ERTS. It can thus be executed on any target system.

III. Methods

After introducing the concepts of the IEC 61499 and Erlang, the semantics and execution of the IEC 61499 basic function block can be analyzed further. This analysis is the foundation for the implementation of the basic function block in Erlang. Ferrarini and Veber [10] describe different possible implementations of the IEC 61499 standard. The implementation in this paper corresponds to a multitasking implementation with time slices without a fixed function block scan order. Regarding the semantics defined by Vyatkin [11], the implementation follows an asynchronous, parallel execution. Wherever further choices about the execution semantics were necessary, the choice facilitating an implementation in Erlang was made. This section describes one possible IEC 61499 implementation in Erlang, not all possible IEC 61499 implementations.

Further, the implementation is limited to the basic function block. Possible implementations of other types are described by Prenzel and Provost [8].

A. IEC 61499 Basic Function Block Analysis

Before using a basic function block (FB), its type has to be defined. Its instance can then be used in a function block network in an application. In the application, this function block instance is defined by the following properties:

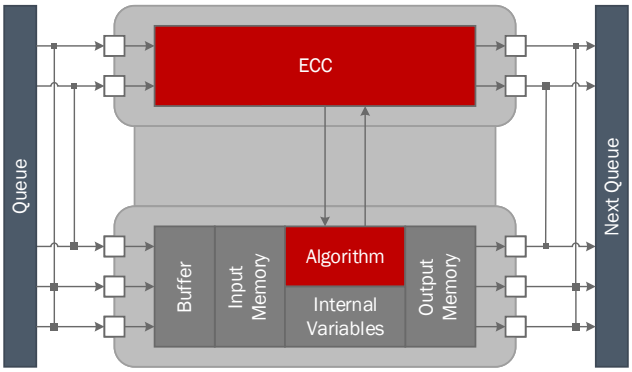

Fig. 3. IEC 61499 Function Block Layout

- A type and instance name
- Event inputs and outputs, possibly connected to other FBs
- Data inputs and outputs, possibly connected to other FBs

These properties are commonly stored in the application file. The basic function block type, on the other hand, is characterized by:

- The Execution Control Chart (ECC)
- A set of algorithms
- A set of internal, input, and output variables
- A set of *with* qualifiers for inputs and outputs

The ECC describes the relation between input events, algorithms and output events in the form of a Moore-type finite state machine. At the occurrence of an input event, the available transitions are evaluated. When a new state is entered, the corresponding algorithms are executed and outgoing events are sent. The transitions may have guards using all available variables.

The algorithms have access to the variables and can calculate new output and internal variables. The IEC 61499 standard does not strictly specify the language for the algorithms.

The set of variables holds the input-, output-, and internal variables. On the occurrence of an input event, input variables connected to this event by the *with*-qualifier are updated. When output events are sent, the corresponding data outputs are propagated to the connected function blocks. Internal variables are only updated by the algorithms.

In this implementation, due to the separate channels for events and data, the data has to be buffered and sampled. When a data message arrives, its content is stored in a buffer. This buffer contains one set of values of all data inputs. If a new data message arrives, it overwrites the currently buffered value. On the occurrence of the correct input event message, this input is sampled from the buffer, thus writing it into the input memory. The full layout of the basic function block is depicted in Figure 3. The calculated output value is stored in the output memory until the corresponding output event is sent and the output value is propagated to the connected function blocks.

B. IEC 61499 Basic Function Block Implementation

An approach to automatically generate an Erlang system from an IEC 61499 application is described by Prenzel and Provost [8]. Their approach served as a proof of concept for updating an automatically-generated IEC 61499 application, but not all features of the basic function block were yet supported. This paper focuses on the full implementation and evaluation of a basic function block with data connections and functional algorithms.

Kruger and Basson [14] showed an implementation of a resource holon in Erlang/OTP, although unrelated to the IEC 61499, and concluded that Erlang is well suited due to its modularity, scalability, customisability, maintainability, and robustness characteristics.

Since Erlang is a functional programming language, the function block is implemented as a set of functions. To simplify this, the OTP behavior `gen_statem` for finite state machines is used.

The implementation makes heavy use of records (a key-value construct) and is fully specified, allowing the use of the dialyzer for checking of type safety [15]. Displaying full functions would be out of the scope of this paper, but Listing 1 shows the exemplary specification for the `handle_event/4` function. The number behind the function name (`/4`) resembles the number of arguments for this function.

There are two types of functions to be implemented: Generic functions that are the same for all possible basic function blocks (such as `handle_event/4`), and functions that are specific to a certain function block type. To initialize the instance of a function block type, a set of variables is passed to it during startup.

```
handle_event(event_type(), message(),
  state(), {internalData(), connections()}) ->

  {next_state, state(),
    {internalData(), connections()}}.
```

Listing 1. handle_event/4 callback function

1) Generic Functions: The generic functions are mostly the functions expected by the `gen_statem` behavior, i.e. functions for initialization, startup, termination, update and event handling. Termination and update functions can be used for possible fault tolerance mechanisms or dynamic software updating.

Figure 4 displays the structure of the `handle_event/4` function that is called whenever a new message is picked from the mailbox. Depending on the type of message, different sequences are followed. For data messages, the buffer is updated and the function returns. For event messages, the ECC is called after the inputs are sampled. The ECC itself first calls the state machine to find executable transitions. If no transition can be triggered, the ECC returns. If a new state is entered, the corresponding action for this state is performed, i.e. algorithms are executed and data and events are distributed to other function blocks. Finally, the ECC is called again to find available transitions without events. This recursive call

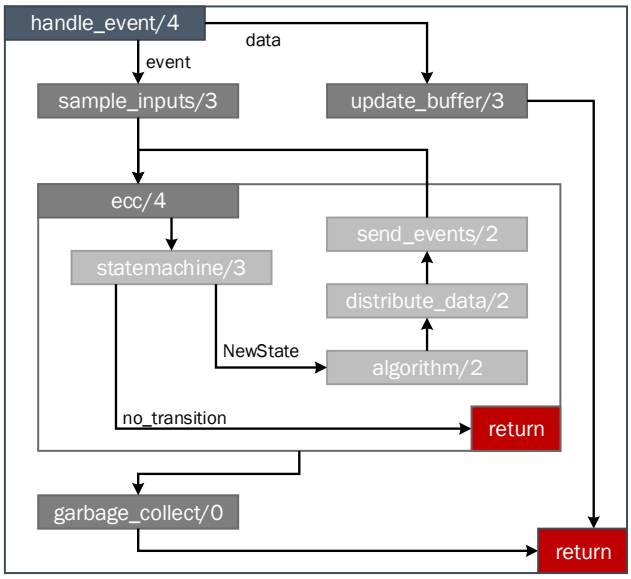

Fig. 4. IEC 61499 Function Block Timing

will run until no transition can be fired anymore. Consequently, by design, the ECC should not contain live locks and must terminate eventually.

2) Function Block Specific Functions: While the `handle_event/4` and `ecc/4` functions are the same for every basic function block type, `sample_inputs/3`, `update_buffer/3`, `statemachine/3`, `algorithm/2`, `distribute_data/2`, and `send_events/2` are function block specific functions. That means they have to be generated from the function block type specification. Their implementation in Erlang is very straight forward, as can be seen in the specification in Listing 2.

```
statemachine(event_in() | no_event,
  state(), internalData()) ->

  state() | no_transition.
```

Listing 2. statemachine/3 function

Every transition can be defined by the event, the current state and a guard on the internal data. This `statemachine/3` function then returns the corresponding state or `no_transition`.

3) Initialization: The Erlang process is part of a supervision tree and started by the responsible supervisor. The supervisor spawns the process from the Erlang module defining the function block type with an additional set of arguments. Those arguments are used to initialize the process and later, the function block. The first argument is the instance name, which is used to register the process. This is more efficient than using process identifiers, as the function block instance, by design, must have an unambiguous name. In addition, there is a set of initialization values, in case the function block instance has constant inputs, and a connection table. The connection table describes for every outgoing event and data connection the name of the receiver and what name is expected.

4) Language of the algorithm: The IEC 61499 standard does not strictly define the language of the algorithm. For an implementation in Erlang, multiple options are available. The most forward approach, and the one used for the evaluation in this paper, is to implement the algorithm directly in Erlang.

The most convenient approach would be to convert the algorithms to C code and call them as natively implemented functions in Erlang. The performance is generally even better than Erlang code, but long algorithms may lead to blocking, as they can not be interrupted by the scheduler.

The third approach is the automatic conversion from one language (IEC 61131-3, C, Ladder Logic, ...) to a language more compatible with Erlang, i.e. Erlang itself or Elixir. This is currently investigated.

IV. Performance Evaluation

The previous section described the implementation of the IEC 61499 basic function block in Erlang. As initially stated in the introduction, this paper aims to evaluate the real-time performance of the Erlang Runtime System (ERTS) for this purpose. Since it is not possible to put a deterministic upper bound on the worst case execution time in Erlang, only an empirical analysis of the distribution of the reaction time can be produced.

A. Methods

Real-time performance is a topic intensively studied in literature. Wilhelm et al. [16] created an important overview of how to estimate the worst case execution time (WCET) of a system. More related to the IEC 61499, Zoitl [17] presents a method for the real-time execution of this standard.

The result of interest in this paper is the reaction time of a function block and what parameters it depends on. In combination with a control flow analysis, this result may be used to determine the reaction time of a function block event chain, similar to the description by Zoitl [17], although empirical.

To measure the reaction time, a basic function block is implemented and equipped with time measuring capabilities. In this case, the function block will return a set of timestamps to the requesting process. The full evaluation setup is described in the next section.

In total, 8 values are collected per measurement. Those are a counter, the current system time, the monotonic process reduction count, and 5 durations of the function block execution:

1) T1: Event send time to the FB
2) T2: Data sampling time
3) T3: Execution control chart execution time
4) T4: Garbage collection time
5) T5: Event send time from the FB

T1 is the time it takes to send a message from a high priority process to a normal priority process. **T2** is the time to update the input memory from the buffer according to the event. **T3** is the time to execute the ECC. **T4** is the time to perform the garbage collection. **T5** is the corresponding counterpart to T1, i.e. sending a message from a normal priority process to a high priority process.

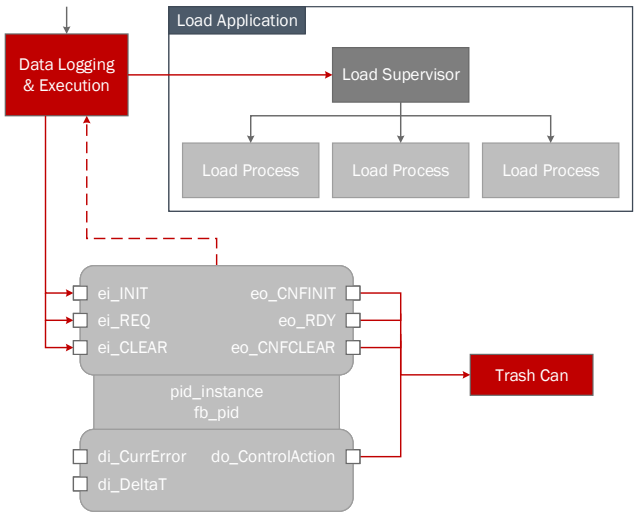

Fig. 5. IEC 61499 Function Block Timing

B. Evaluation Setup

To make realistic measurements, the function block has to be embedded in an environment, as depicted in Figure 5.

There is a `high` priority process in charge of orchestrating the tests (Data Logging & Execution). It starts the load, requests the function block execution, and stores the data. The outputs from the function block are sent to a process serving as a trash can. Additionally, there is an application to apply additional load to the ERTS. This application allows the spawning of processes that repeatedly perform an expensive computation, thus filling the run queue.

The implemented function block, implemented as a `normal` priority process, performs the action of a PID-controller, calculating an algorithm when requested and distributing its results to other processes. Its interface is shown in Figure 5 and the ECC is depicted in Figure 6. The dashed connection from the top of the function block interface stands for the additional time-measuring output.

To achieve realistic and consistent execution, the function block is triggered every 25 ms. This value was chosen as a compromise between the highest possible resolution (frequent measurement), economical generation of data, and avoidance of feedback from the measurement. This execution is achieved by measuring the total execution time and waiting for the remaining time. In case the execution takes longer than 25 ms the next cycle is started immediately after the previous cycle.

The additional load fills the run queue of the scheduler, thus causing the function block process to compete with other, `normal` priority processes. The load processes are always executed for the full amount of reductions, in this case 4000, before they are interrupted.

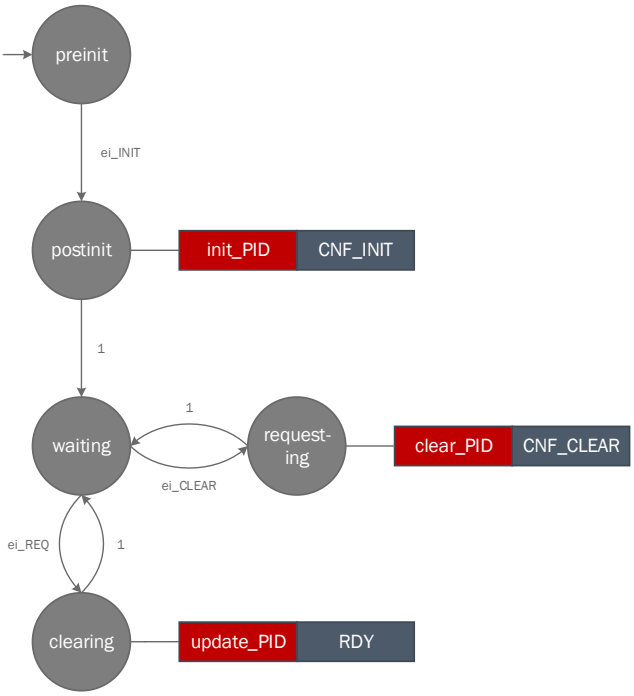

Fig. 6. IEC 61499 PID ECC

C. Tests

In a realistic setup, the function block would be part of a larger network. This concurrent execution causes the run queue of the scheduler to fill with processes. In normal systems, this *additional load* for an event-based system depends on the input events and their frequency and is highly fluctuating.

In this benchmark, in order to achieve the most deterministic result, the load processes are executed for the full amount of reductions and are continuously requested. Five test cases are executed with varying numbers of concurrent processes, starting with two and moving up to 32 in discrete steps. Lower numbers of processes show large fluctuations, whereas more processes would exceed the cycle time. This number resembles the number of processes waiting in the run queue before the function block.

The test platform for this test is a Raspberry Pi 3 Model B with Raspian Jessie and Erlang 20. The only modification of the Raspberry Pi is to manually disable CPU throttling. Only one scheduler is spawned to prevent work stealing between the schedulers. The ERTS is started with a `nice` value of -20 to prevent interruptions as much as possible.

Each test is performed for 7.200.000 executions, which is equivalent to 50 hours, or 250 hours in total, for all 5 test cases.

V. Results

The performance evaluation yielded 2 main results:

- Table I comparing the maxima, minima, mean values and standard deviations of the individual test cases for every measured variable.
- A scatter plot (Figure 7), visualizing the temporal distribution of the total time data. The tests were performed consecutively, but they are overlayed in the scatter plot to allow an easier comparison.

The table (Table I) shows the relevance of the contributing durations. The *Send Time 1*, which represents the time it takes for the process to receive a time slice, is dominating in all test cases. Second is the *Execution Control Chart Time*. The manually enforced garbage collection takes a considerable chunk of the execution time of the function block. Sampling of the buffer values is nearly negligible. The *Send Time 2* can serve as a reference value for the reactivity of the scheduler, as in this case, the recipient of the message runs with a higher priority and will be scheduled immediately.

The scatter plot (Figure 7) reveals periodic interruptions every hour and once every 24 hours (around 6:25 AM). For 32 processes, a faint sinusoidal shape of the disturbances can be perceived. The more remarkable outliers for 2 and 4 additional processes all resemble the distribution of the next higher test case.

In addition, the process consistently required 91 reductions for the execution in every cycle. This includes the algorithm itself and all overhead introduced by the function block implementation.

VI. Discussion

The results present the performance of an asynchronous, parallel function block implementation on a simple, commonly-available hardware platform with a fair round-robin scheduler.

It is important to note the limitations of the evaluation. Only one scheduler was used, thus the Raspberry Pi platform was only able to use one core for Erlang. This also implies that the operating system can run other tasks in parallel on other cores. If four schedulers were spawned, parts of the performance would potentially improve by 300%, although more interruptions by the OS could be expected. The current operating system does not guarantee real-time constraints. Further optimizations may be applied by using more advanced operating system tools or real-time operating systems. This was not within the scope of this paper.

The *Send Time 1* and consequently the *Total Time* scale linearly with the number of additional processes in the queue. This is very distinct for the mean and minimum. For 2 and 4 processes, the maximum is distorted due to the outliers visible in the scatter plot. This result follows the characteristics of the scheduler. A longer run queue causes a longer wait time to be executed. *Send Time 2* is independent of the number of additional processes, because the high priority process will get executed immediately. Intuitively, setting many processes to a higher priority will diminish the advantage. Both the garbage collection and ECC show notable outliers worth further investigation. The cyclic interruptions visible in the scatter plot are most likely due to the operating system. The main disadvantage of the current framework is the dependency on a non-real-time operating system.

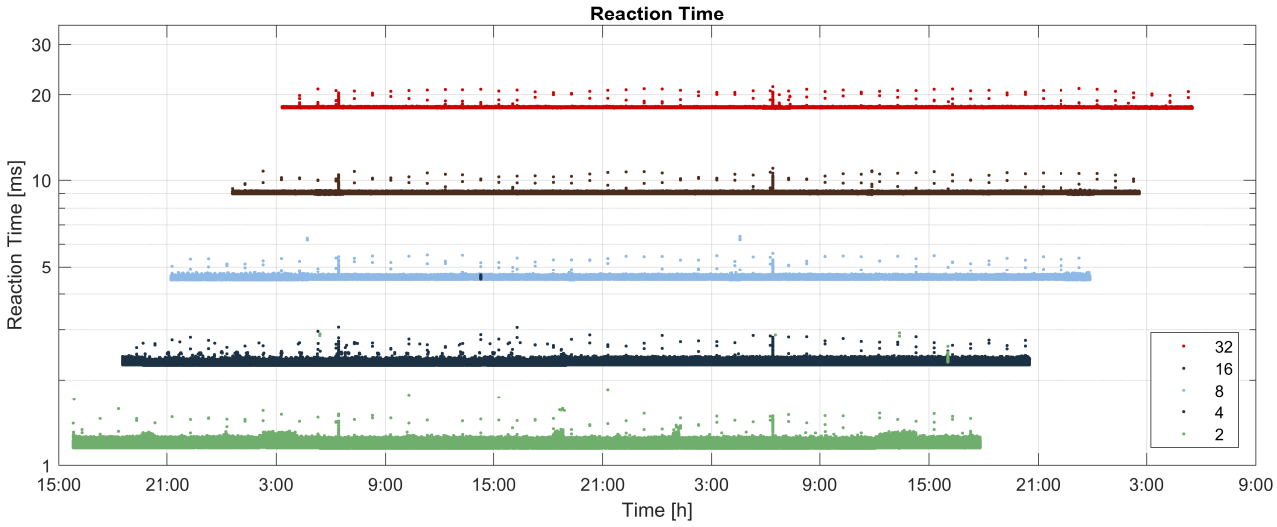

Fig. 7. IEC 61499 Function Block Timing

TABLE I
TABULAR PERFORMANCE ANALYSIS RESULTS

	#	Max [ms]	Min [ms]	Mean [ms]	StdDev
Total Time	2	2.9252	1.1452	1.1653	0.0131
	4	4.6620	2.2497	2.2844	0.0181
	8	6.3932	4.4769	4.5339	0.0195
	16	11.0155	8.9183	9.0264	0.0225
	32	21.3346	17.8875	18.0100	0.0340
Send Time 1	2	2.8468	1.1171	1.1358	0.0116
	4	4.6028	2.2222	2.2554	0.0168
	8	6.3147	4.4491	4.5045	0.0181
	16	10.9293	8.8894	8.9968	0.0209
	32	**21.2492**	17.8592	17.9803	0.0323
Sample Time	2	**0.0762**	0.0011	0.0013	0.0001
	4	**0.0766**	0.0011	0.0013	0.0002
	8	**0.1092**	0.0011	0.0013	0.0002
	16	0.0071	0.0011	0.0013	0.0001
	32	0.0065	0.0011	0.0013	0.0001
ECC Time	2	0.1241	0.0182	0.0206	0.0032
	4	**0.1940**	0.0180	0.0204	0.0033
	8	**0.2212**	0.0182	0.0206	0.0035
	16	0.1111	0.0183	0.0208	0.0037
	32	0.1072	0.0185	0.0209	0.0040
GC Time	2	**0.1031**	0.0067	0.0076	0.0006
	4	**0.1081**	0.0065	0.0074	0.0008
	8	**0.1269**	0.0065	0.0074	0.0008
	16	0.0192	0.0066	0.0075	0.0006
	32	0.0590	0.0066	0.0075	0.0006
Send Time 2	2	0.1204	0.0127	0.0180	0.0049
	4	**1.2427**	0.0127	0.0182	0.0050
	8	0.1294	0.0127	0.0164	0.0037
	16	0.1048	0.0128	0.0183	0.0037
	32	0.1927	0.0128	0.0186	0.0054

The results allow two separate interpretations of the maximum number of processes executable within the 25 ms deadline. The current framework uses 32 concurrent processes as a maximum load. In a worst case scenario, where 32 processes are busy blocking the scheduler, a real-time task may still be executed within 25ms. This corresponds to the fairness property of the scheduler, as long running processes will eventually be preempted. On the other hand, the current function block implementation required under 100 reductions for an ECC execution. Thus, preemption is rather unlikely. In a realistic setup, where each function block process will use much less than 4000 reductions, a much larger number of processes can fit inside the 25 ms window. Assuming 500 reductions per function block and 4 schedulers, 1024 individual function blocks may be executed while consistently keeping the 25 ms deadline.

VII. Conclusion

The aim of this paper was the demonstration of an asynchronous, multi-tasking IEC 61499 Basic Function Block implementation in Erlang and its real-time performance evaluation. Erlang and the IEC 61499 share many similarities, since they are both intended for distributed, concurrent, and event-triggered applications. This simplifies the implementation of the IEC 61499.

Using Erlang as an implementation language comes with many benefits, e.g. the native support for distribution, concurrency, and event-based execution, as well as the functional paradigm which is well-suited for safety and traceability. Erlang also allows dynamic software updating, i.e. updating running applications, which will be investigated in further projects.

On the other hand, Erlangs concurrency introduces overhead into the system and it can not guarantee hard real-time constraints due to the fair scheduling and garbage collection.

Erlang's flexibility and scalability is not necessarily beneficial to an IEC 61499 implementation.

This paper presented a feasible IEC 61499 basic function block implementation in Erlang. On a simple single-board computer with a soft real-time operating system, the real-time performance was consistent. Minor cyclic interruptions, most likely due to the operating system, were observed. The garbage collection was triggered manually to prevent unanticipated delays. The implementation of the function block is slim enough to not be interrupted by the scheduler, although this means the scheduler of the ERTS behaves more like a cooperative round-robin scheduler. The reaction time of the function block thus depends largely on the number of concurrent processes in the run queue. The upper bound, when every process uses as many reductions as possible until it is interrupted, was shown in this paper. In this scenario, 32 additional load processes could be executed while still keeping a 25 ms deadline.

In the future, three parallel paths should be taken to improve the real-time performance.

- The operating system is a major cause of interruptions in this framework. In principle, the operating system may induce unbounded delays. A hard real-time operating system or at least a highly optimized regular operating system may remedy this issue.
- The Erlang scheduler is optimized for a specific set of applications, i.e. telecommunication systems. The priority framework is discouraged and inflexible. In addition, the IEC 61499 is lacking in recommendations to specify real-time constraints.
- Tasks with particularly strict real-time constraints with catastrophic consequences may be outsources to dedicated hard real-time controllers. The interfacing with the ERTS is straightforward.

In addition, the performance evaluation should be performed on a realistic use case with real-time constraints. The IEC 61499 implementation described in this paper should be extended with more features. Especially dynamic software updating is worth investigating. Even if Erlang as an implementation language for the IEC 61499 may not prevail, the future of the IEC 61499 standard may benefit from different perspectives.

References

[1] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-Art review," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, Nov. 2011.

[2] 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*, V. Mařík, V. Vyatkin, and A. W. Colombo, Eds. Springer, Berlin, Heidelberg, 2007, pp. 326–337.

[3] 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 *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings*, vol. 2. IEEE, Oct. 2001, pp. 113–118 vol.2.

[4] C. Schnakenbourg, J. M. Faure, and J. J. Lesage, "Towards IEC 61499 function blocks diagrams verification," in *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, vol. 3. IEEE, Oct. 2002, p. 6 pp. vol.3.

[5] 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. Part C, Applications and reviews.*, vol. 41, no. 1, pp. 41–51, Jan. 2011.

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

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

[8] L. Prenzel and J. Provost, "Dynamic software updating of IEC 61499 implementation using erlang runtime system," in *Proceeding of IFAC World Congress 2017*, 2017, pp. 12 416–12 421.

[9] A. Zoitl and R. Lewis, *Lewis, Robert. Modelling control systems using IEC 61499: Applying function blocks to distributed systems*. IET, 2014, vol. 95.

[10] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on*, Jun. 2004, pp. 612–617.

[11] V. Vyatkin, "The IEC 61499 standard and its semantics," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, pp. 40–48, Dec. 2009.

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

[13] E. Stenman, "The BEAM book," https://github.com/happi/theBeamBook, Dec. 2017, accessed: 2017-8-14.

[14] K. Kruger and A. Basson, "Erlang-based control implementation for a holonic manufacturing cell," *International Journal of Computer Integrated Manufacturing*, vol. 30, no. 6, pp. 641–652, Jun. 2017.

[15] T. Lindahl and K. Sagonas, "Detecting software defects in telecom applications through lightweight static analysis: A war story," in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2004, pp. 91–106.

[16] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, and Others, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, p. 36, 2008.

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