Dynamic Software Updating of IEC 61499 Implementation Using Erlang Runtime System

Laurin Prenzel, Julien Provost

Technical University of Munich, Safe Embedded Systems,
85748 Garching bei München, Germany
(e-mail: laurin.prenzel@tum.de, provost@ses.mw.tum.de)

Abstract:
Dynamic Software Updates (DSU) permit to decrease downtimes caused by updates or bug fixes and thus increase productivity, which is an ever present target during development of industrial production systems. This study implements a compiler to transform an IEC 61499 model into executable code for the Erlang Runtime System (ERTS) which natively features DSU, and investigates its feasibility. As a case study, a small production plant is implemented and updated on-the-fly with new features and safety fixes. This case study shows that DSU by using the ERTS is feasible. However, additional information for the update structure, content and schedule is required from an external source.

Keywords: Dependable manufacturing systems control; Discrete Event systems in manufacturing; Flexible and reconfigurable manufacturing systems

1. INTRODUCTION

Current industrial automation plants are controlled by programmable logic controllers (PLCs), soft-PLCs or industrial PCs (IPCs). Although the programming interface remains unchanged, using a software-based PLC or an IPC enables the implementation of methods that were impractical on a hardware-based PLC. One of these new methods is Dynamic Software Updating (DSU).

There have been endeavors to implement Dynamic Software Updating in numerous fields (Seifzadeh et al. (2013)). Since the lifetime of a production facility can be very long, it is inevitable to eventually update its software. This may include the implementation of new features, an increase in performance, or simple bug fixes. Depending on how severe this change is, the update of the facility may be unfeasible due to downtimes caused by the shutdown, update and restart phases of the plant. By using DSU, modifications of the model can still be prepared offline and follow the same modeling procedure, but the downtime can be drastically decreased and in the best case completely eliminated, thus increasing the productivity of the plant over the life cycle.

In this paper, the feasibility of DSU for production automation will be demonstrated by implementing it with the functional programming language Erlang and the Erlang Runtime System. Originally developed in the telecom industry for distributed, highly available systems, Erlang facilitates the use of finite state machines and DSU.

Although it is generally possible to manually program an industrial automation plant in Erlang, this is not the intention of this paper. Since production plants were traditionally controlled by PLCs, the IEC 61131 standard defines specific languages for this purpose (IEC 61131-3 (2003)). In 2005, the IEC 61499 standard for function blocks for distributed control systems was published (IEC 61499-1 (2012), Zoitl and Lewis (2014), Vyatkin (2007)) and will be used in this paper.

The approach of this study would permit companies to use DSU without introducing a new programming language to the workflow. Furthermore, this study investigates the additional information needed to enable DSU on an IEC 61499 implementation. The case study shows that by using DSU the non-reactive time during an update can be reduced to approximately 20ms on a small-scale system.

The remainder of the paper is structured as follows: Section 2 presents a theoretical outline over the two main elements used in this paper: the IEC 61499 standard and the Erlang programming language and Runtime System. Section 3 describes the actual implementation of the IEC 61499 standard in the Erlang Runtime System and how DSU can be achieved in practice. This is followed by the use case in section 4. Finally, the case study and future extensions are discussed.

2. BACKGROUND

This section introduces the main components of this study: the IEC 61499 standard and the Erlang programming language and Runtime System.

2.1 IEC 61499 Standard

The IEC 61499 standard is the successor of the IEC 61131 standard for programmable logic controllers (PLC). It

⋆ This research was partially supported by the German Federal Ministry for Economic Affairs and Energy (BMWi ZIM project ZF4304702BZ6) at the Technical University of Munich.

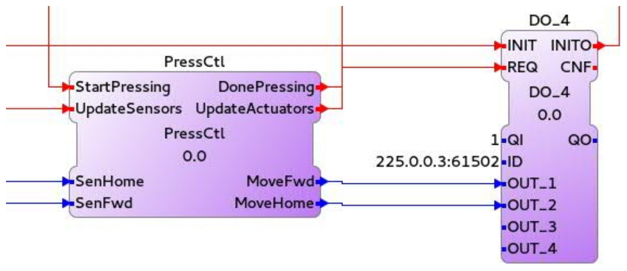

Fig. 1. Example of IEC 61499 function block network, generated with 4diac (Strasser et al. (2008))

describes the use of function blocks with a separated event and data flow for distributed control systems by combining graphical aspects (function blocks) with text elements (algorithms) (IEC 61499-1 (2012), Zoitl and Lewis (2014), Vyatkin (2007)). Parts of a function block network are depicted in Figure 1.

Function Blocks Several types of function blocks (FBs) can be used to encapsulate the functionality. The basic FB is controlled by an Execution Control Chart (ECC), which schedules the execution of algorithms and the transmission of outgoing events when triggered by incoming events. The ECC behaves like a Moore machine, i.e. actions only depend on the current state. Other FBs can be used to structure the application (composite FB, subapplication block) or to introduce other functionality (service interface FB) but will not be considered in this study. The IEC 61499 standard does not strictly specify a language for the algorithms.

Input/Output Connections The FBs are organized in an application where their event and data inputs and outputs are connected. Data connections have to be unambiguous, i.e. a data input can only be connected to one data output. In order to refresh a data input, it has to be associated with an incoming event by using the with-qualifier. A data input without a with-qualifier is constant. Likewise, data outputs need a with-qualifier to specify when they are updated. Although data and event flows are separated, data can only be updated with an event.

The events are passed to the ECC in the order they were received. If an event does not match an outgoing transition condition of the current state, the event is discarded and the ECC waits for the next event. Transition conditions may consist of an event and a data condition.

2.2 Erlang and the Erlang Runtime System

Erlang is a functional programming language that was developed for communication and network services. It is capable of spawning and sustaining a large number of lightweight processes that can communicate, store data and execute code. The properties of the process are defined by the module from which it is spawned. The Erlang Runtime System (ERTS) takes care of scheduling and managing the processes (Armstrong (1996),Armstrong et al. (2016)).

Open Telecom Platform The Open Telecom Platform (OTP) contains so called *behaviours* that were formalized from common applications. Generic parts of the code are stored in the *behaviour*, while the callback module contains only what is necessary specifically for this module.

`gen_fsm` Behaviour The OTP *behaviour* `gen_fsm` implements a finite state machine. The user may define the properties of the state machine by creating the callback module. It can be split into two parts: The first part contains general functions, such as how to handle synchronous or asynchronous calls and how the process is started, updated and terminated.

The second part characterizes the transition functions, consisting of the name of the current state, a guard, an action, and the target state. During operation the events are compared to the transition functions of the current state and if there is no match, they are put back to be evaluated later. Alternatively, by including a self-loop-type transition function that always finally matches, events can be discarded after evaluation. Because the action is attached to the transition, `gen_fsm` state machines are Mealy machines. Listing 1 depicts a generic example of a transition function for a `gen_fsm` state machine with a self-loop-transition.

Listing 1. `gen_fsm` transition function example

```
state(event,StateData) ->
    % Action
    {next_state, NextState, NewStateData};
state(_,StateData) ->
    {next_state, state, StateData}.
```

Dynamic Software Updating The OTP contains a framework for Dynamic Software Updating (or Hot Code Loading as termed in Erlang). The update procedure is coordinated by the *release handler*. A *release* consists of *applications* that are built from Erlang *modules*. Releases and applications have specific files defining what modules to use.

Additionally, there are files specifying what processes to start, stop or update during an up- or downgrade of a release or application (`appup` and `relup`-files). The release handler uses these files to coordinate the update.

Updating a process is done in 4 steps:

(1) Suspend the process
(2) Load the new module, transform the internal state and upgrade to the new module
(3) Remove the old module
(4) Resume the process

Suspended processes can receive messages but cannot react to them. They will instead be kept in the mailbox. The process has to be suspended if the internal state is modified to prevent a change of state during the update. The Erlang Runtime System allows the loading of two versions of a module, an *old* and a *current* version. When a new module is loaded, the other one switches from *current* to *old*. The processes continue running the *old* version until they are told to switch. The internal state is then transformed via a `code_change` function that has to be implemented in the new version of the callback module.

In case the update does not change the internal state but is just a code extension, suspension is not necessary and the process can just switch from *old* to *current* code.

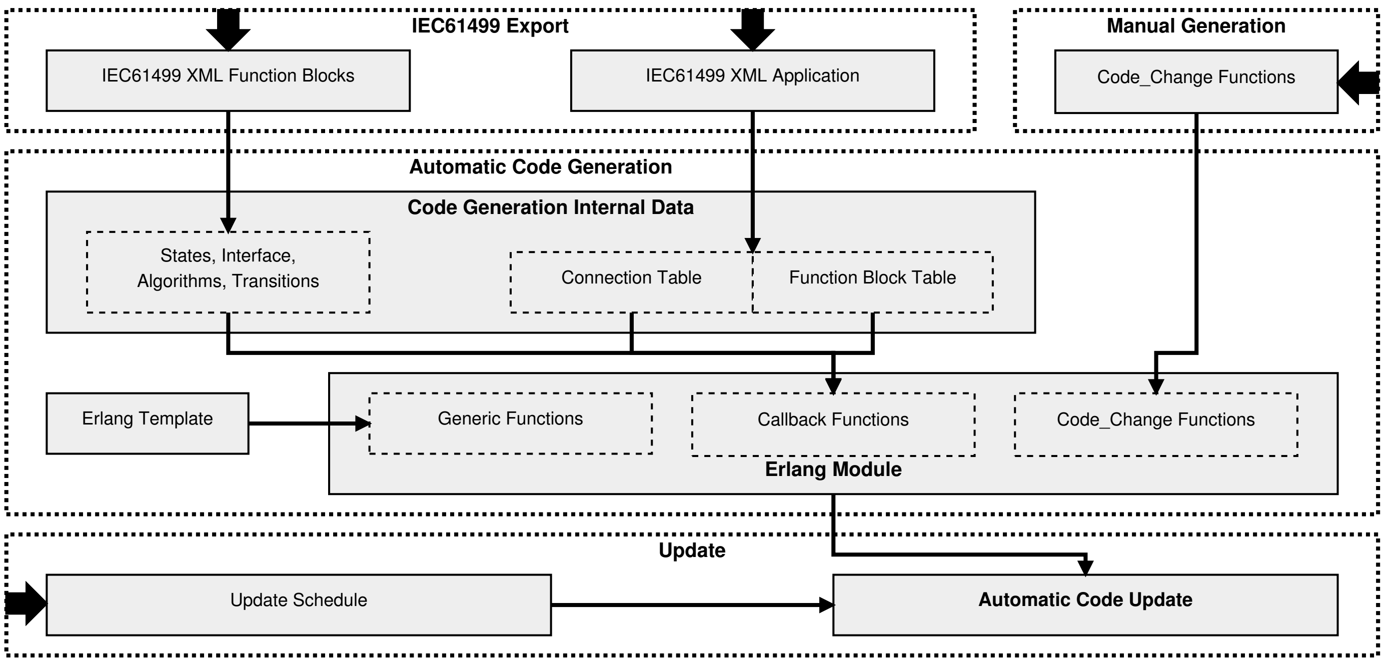

Fig. 2. Flow chart of code generation and update procedure

3. IMPLEMENTATION

Before the code generation can be automated, the transformation protocol between the IEC 61499 model and the Erlang implementation has to be defined:

- Every instance of a IEC 61499 function block can be represented by an Erlang process and its ECC can be implemented with the OTP behaviour for finite state machines (`gen_fsm`).
- IEC 61499 Moore machines can be translated into Erlang `gen_fsm` Mealy machines by moving the algorithm from the state to the transition.
- Since the IEC 61499 does not strictly specify the language of the algorithm, the algorithms were restricted to basic instructions in Erlang.
- IEC 61499 data and event flows can be merged since data can only be updated with an event. If an event updates multiple data inputs coming from different function blocks, the corresponding event can be rerouted to collect all necessary data on its way.

Once the general transformation protocol is selected, a software tool to translate IEC 61499 models to Erlang modules is implemented.

3.1 Automatic Code Generation

In order to generate code from the IEC 61499 model it has to be exported. The standard defines an XML-format that can be used for this purpose. Individual XML-documents for the function blocks and the applications that contain the necessary information are created.

The code generation process flow is depicted in Figure 2. The *IEC 61499 XML application* file contains the information about the instances of function blocks and their connections. The resulting connection table consists of the sender, recipient, message and trigger event for every event connection. In case data flows have to be considered, this table can be edited to redirect the event flow to fetch the corresponding data and add it to the message.

Once it is known which function blocks are used in the application, the *IEC 61499 XML function block* file is read and analyzed to obtain the interface, algorithms, states and transitions of a specific function block. This leads to the generation of the Erlang *Callback Functions* that are finally inserted into a prepared *Erlang Template*. The template supplies the generic parts for all callback modules, such as start and terminate functions.

For every function block instance, one *Erlang module* is generated. In this study, the modules are started manually, but Erlang enables the use of supervisors to automatically spawn processes. This will be considered in future work.

For an update, the `code_change` functions have to be supplied as well. These functions are currently created manually.

3.2 Dynamic Software Updating / Hot Code Loading

The process of updating an operational plant consists of two parts: the offline preparation and the online update.

In this study, the system will be upgraded manually to demonstrate the feasibility of upgrading a running system without having to generate complete releases and applications. Nonetheless, the updating procedure is identical to how the integrated release handler performs an update (as described in section 2.2.3).

Offline Preparation This phase includes the generation of the new code version and planning of the update procedure.

The user decides for every process whether it will be kept, terminated or updated in the new version. Kept processes are not influenced by the update. If it is updated, the internal state of the process is transformed and transferred to the new process.

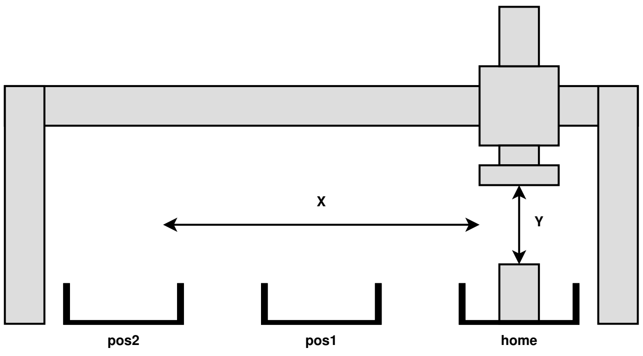

Fig. 3. Evaluation platform overview

Once the update structure and content is created, an update schedule specifies the order of starting, updating and terminating. This leads to the generation of `relup` and `appup` files.

If the update is too severe and internal states are not compatible, it may be more efficient to terminate the old FB and create a new one.

Online Update The update can then be applied to the plant. The updating procedure of the plant looks like this:

(1) Compile modules
(2) Start new modules
(3) Suspend all processes from module to be updated
(4) Load, update and resume processes
(5) Repeat from 3 until all modules are updated
(6) Terminate old processes

Plant operation is uninterrupted during the first two steps. Suspension of a process will prohibit it from reacting to incoming events, but the events are stored in the mail box for later use. Simple updates (code extensions) may not require suspension. After all processes are updated, the plant is again fully functional. This non-reactive time takes place in the matter of milliseconds and is thus much shorter than usual downtimes.

If interruptions are undesirable, the update can be scheduled to be executed only at specific positions, for example in idle states or when a new workpiece is introduced.

4. USE CASE

4.1 Evaluation Platform

The test platform consists of a portal crane that moves workpieces from the input position to multiple stations (see Figure 3). There are 3 actuators (X-axis, Y-axis, magnetic lock) and 8 sensors (3 light sensors, 5 position switches). The actuators and sensors are controlled by 3 different remote input output modules (RIOMs) that are addressed via a Modbus TCP protocol. Other communication protocols could be used in Erlang by linking a C/C++ library to a process.

Three manually programmed Erlang processes were put in place to cyclically set and read the coils and send events to their subscribers. These processes could be embedded in the IEC 61499 model as Service Function Blocks.

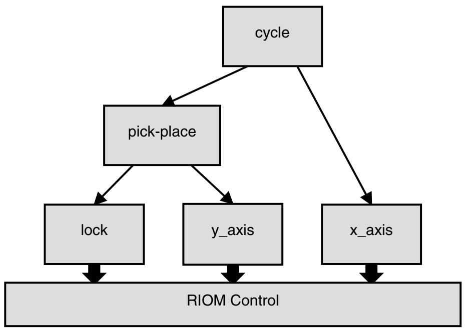

Fig. 4. Function block layout model 1

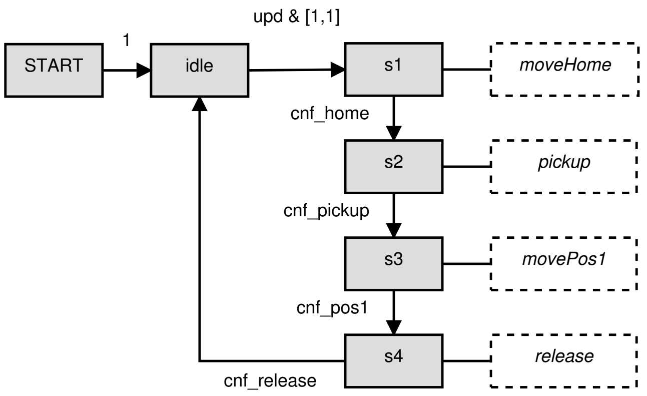

Fig. 5. Execution Control Chart for 'cycle' FB of model 1

4.2 Model

For the use case of dynamically updating a production plant during operation, two models had to be implemented: a flawed model and an improved version.

The first model (Figure 4) consists of 5 function blocks. Three function blocks control the movement in the X- and Y-direction and the locking mechanism. One intermediate block controls the process of picking up and putting down pieces. The cycle block manages the overall cyclical behavior of the plant. Its ECC is depicted in Figure 5. This model reacts to the arrival of a new workpiece by picking the workpiece up and putting it down at the first machine.

The second model improved the procedure by waiting for the station to be free, utilizing the second machine, and returning to the home position after delivery instead of waiting at the machine. This change was introduced by expanding the ECC of the top cycle block (Figure 6). A decision point was introduced where the crane waits for a clearance from the stations. The function block controlling the X-direction was replaced by three individual blocks for the three available positions (Figure 7). This lead to a total of 7 function blocks. The intermediate block responsible for the pickup and release process remained unaffected.

4.3 Erlang Code Generation

The software presented in this study reads the IEC 61499 model and returns equivalent Erlang modules. Those modules are then compiled and started manually. Finally, the cyclically executed RIOM control blocks can be started, connecting the Erlang implementation to the real plant.

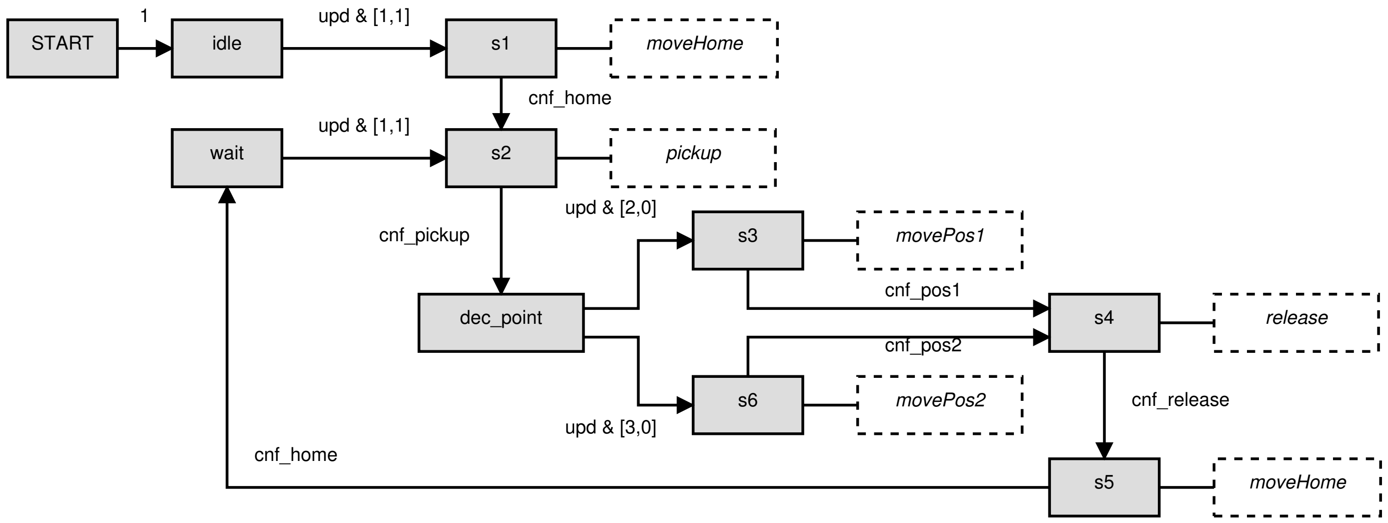

Fig. 6. Execution Control Chart for 'cycle' function block of model 2

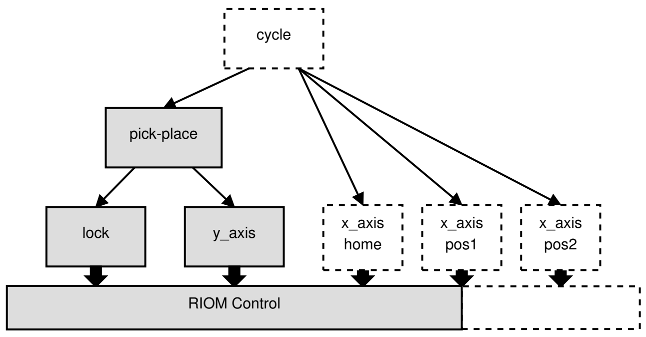

Fig. 7. Function block layout model 2

Following the execution of the first model, an update is modeled as described in section 3.2. Figure 7 shows the structure of the update and which blocks are affected. Grey boxes are kept and will not be changed. Dashed boxes are new or affected by the update.

There are two important differences between modeling the first version and modeling an update.

- Blocks that are selected to be updated or that have the same role in the new application must be registered under the same name. If the block is new to the application, it must have a unique name.
- If a function block is updated, the internal state of the ECC may have to be transformed. In this study, no internal variables were used and only the current state of the ECC was transformed.

The internal state transformation is implemented by the `code_change` functions. They can be generated from a table describing which state in ECC1 will be transformed to which state in ECC2. If the internal state is more complex, the `code_change` functions are more complicated as well.

In this study, ECC2 contains additional states and a modified transition structure, but the states already defined in ECC1 are still linked to the same functions. Therefore, no special `code_change` functions had to be implemented.

4.4 Plant Operation and Update

The plant operation can be split into five steps:

(1) Starting of the first model
(2) Running of the first model
(3) Offline generation of new code
(4) Update
  (a) Compile modules
  (b) Start new modules
  (c) Suspend process to be updated
  (d) Load, update and resume process
  (e) Terminate old processes
(5) Running of second model

In the first step, the code is compiled and started. After this, the plant is available to be used.

Whenever the user has generated a new code version and wants to update the system, he will at first compile and start the new processes. In this scenario, a controller for the third RIOM as well as the three blocks controlling the X-direction were started. Since the old blocks are unaffected by this, the system continues its normal execution.

Once the new modules are started, the cycle block is updated. Therefore, it is suspended, the new version is loaded, the internal state is updated and finally it is resumed. At last, the blocks that are no longer needed are terminated. In this scenario, the old controller for the X-direction is shut down.

Update Time Because the process cannot react during suspension, it was crucial to investigate the suspension duration. In this study, the suspension time amounted to approximately 20ms, which is well below the cycle time of 100ms.

Suspension duration is critical for processes directly controlling the RIOMs, as it may delay a time-sensitive answer (for example stopping a motor). In this use case the updated process did not have any critical states. Starting the new processes for the X-direction before the old process is terminated was uncritical, as the behavior is redundant and superfluous events are dropped by the ECC.

5. DISCUSSION & FUTURE WORK

This study presented an initial approach to update a production plant during operation. This concept will be extended in future applications.

5.1 Implementation of IEC 61499 in Erlang

The software presented in this paper implements only a subset of all available features of the IEC 61499 standard and of the Erlang programming language and Runtime System. This could be expanded by the following features:

- IEC 61499: data flows, internal data, subapplication blocks, resources
- Erlang: appup-/relup files, supervisors, nodes

It is important to note that, currently, algorithms must be programmed in Erlang. Although the IEC 61499 does not strictly specify the language of the algorithms, the most common language is Structured Text (ST). Transformation of ST code to Erlang would have to deal with the different programming paradigms involved. Basic expressions such as setting internal data or basic calculations can be transformed easily, whereas transforming complicated algorithms requires more effort to preserve the initial behavior of the algorithm.

Currently, the IEC 61499 standard does not feature Dynamic Software Updating. Consequently, it does not contain the required terminology. In this study, the necessary information, i.e. the update structure, content, and schedule, was supplied manually.

5.2 Updating with Erlang

Erlang was designed for highly reliable and available systems. Nevertheless, using Erlang is not sufficient to create a reliable system. Designing a dynamic software update requires additional thought and care.

To ensure the consistency of an update, the following three elements have to be supplied by the user:

- Update Structure: The user has to decide which function blocks to keep, update or remove.
- Update Content: The internal state has to be transformed by the `code_change` functions.
- Update Schedule: It is necessary to determine a schedule for the update, including the order and how blocks are updated (i.e., if they have to be suspended).

Ideally, the generation of this information would be automated as much as possible. In most situations, the update will depend on the user's intentions and the user should thus be assisted. A tool could, for example, compare two versions of code, propose a schedule, and automatically create `relup` files, `appup` files and `code_change` functions based on preferences set by the user. Setting those preferences requires expert knowledge about the existing implementation and the plant behavior. To avoid issues during the update, the user has to choose a safe order for suspending, starting and stopping the processes. In this study, names of instances must be consistent throughout the update, as they are used to register the Erlang process. In case the update fails, Erlang can return to the previous version. Nevertheless, the update should be tested extensively prior to the update.

No matter how these three elements are generated, the combination of DSU and industrial automation yields additional issues. Most importantly, to consistently update the internal state of a process, it has to be suspended during the state transformation. Suspended processes may only collect incoming messages in their mailbox but may not react to them until they are resumed. This can cause hazardous behavior in critical states, for example when a motor is running. If interruptions are undesirable, the update can be scheduled to be executed only in safe states. In the case study presented in this paper, the non-reactive suspension time amounts to 20ms, which is well below the cycle time and therefore not an issue. Its scalability will be investigated in future work.

6. CONCLUSION

This paper demonstrated the possibility of updating a production plant during operation. The goal was to let the user program the system in a common programming language for production plants, in this case the IEC 61499 standard for distributed industrial automation systems. The resulting function block network was then translated automatically to be executable by the Erlang Runtime System. Erlang was designed for systems with requirements on high availability and natively enables Dynamic Software Updating.

This implementation presents a currently available and simple solution to reduce the downtime due to updates that could be improved and expanded in the future.

REFERENCES

Armstrong, J., Virding, S., and Williams, M. (2016). Erlang user's guide and reference manual. `http://erlang.org/doc/reference_manual/introduction.html`.

Armstrong, J. (1996). Erlang - a survey of the language and its industrial applications. In *Proc. INAP*, volume 96.

IEC 61131-3 (2003). *Programmable controllers - Part 3: Programming languages.* International Electrotechnical Commission, 2 edition.

IEC 61499-1 (2012). *Function blocks - Part 1: Architecture.* International Electrotechnical Commission, 2 edition.

Seifzadeh, H., Abolhassani, H., and Moshkenani, M.S. (2013). A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5), 535–568.

Strasser, T., Rooker, M., Ebenhofer, G., Zoitl, A., Sunder, C., Valentini, A., and Martel, A. (2008). Framework for distributed industrial automation and control (4diac). In *2008 6th IEEE International Conference on Industrial Informatics*, 283–288. IEEE.

Vyatkin, V. (2007). *IEC 61499 function blocks for embedded and distributed control systems design.* ISA-Instrumentation, Systems, and Automation Society.

Zoitl, A. and Lewis, R.W. (2014). *Modelling Control Systems Using IEC 61499.* 59. IET.