Dynamic Software Update of Stateflow Charts using Erlang Runtime System

Sebastian Q. Roder, Julien Provost

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

Abstract: Reprogramming the controller of an industrial automation system usually requires to halt the system. In this paper, a novel method that allows reprogramming a controller at runtime is presented. The control behavior is modeled using parallel finite state machines, Stateflow being used as an example of modeling tool. Automatic translation to Erlang code is implemented and Dynamic Software Update is enabled using Erlang Runtime System. The presented method is applied and evaluated on a case study as a proof of concept.

Keywords: Industrial Automation, Erlang, Dynamic Software Update, Reprogramming, Stateflow, Parallel Finite State Machines

1. INTRODUCTION

Often, during the life-time of an industrial automation system there is a need to reprogram the controller. It might be necessary to correct a bug, to improve the performance of the system, or to adapt it to new tasks.

The usual reprogramming procedure implies to halt and restart the system, which also consists of ramp down and ramp up phases. In the first place, downtimes and reduced throughput cause losses. Furthermore, companies may not apply updates as often as they could, to avoid those losses. In this way, plants may run longer than necessary with uncorrected bugs or suboptimal performance.

The goal of this work is to investigate a novel method that permits reprogramming a controller at runtime. Downtimes can then be reduced and productivity increased.

The workflow of this method is depicted in Fig. 1. A MATLAB Simulink Stateflow state chart, termed Stateflow in the remainder of the paper, is used to model the controller. The *model* is automatically translated to Erlang *code* and executed with Erlang Runtime System (ERTS) (see Armstrong (1996); Armstrong et al. (2016); Ericsson AB (2016) for more details). In an update scenario, the original *model* is modified to *model**. Then, again, the *model** is automatically translated to Erlang *code**. The final and most important step in reprogramming the controller is to switch the running program to the updated software version.

The concept of modifying a program at runtime is referred to as Dynamic Software Update (DSU). This field of research is promising and has been investigated widely, as detailled by Seifzadeh et al. (2013). However, it is not widespread in industrial automation. Nevertheless, DSU is state of the art in many non-stop areas such as

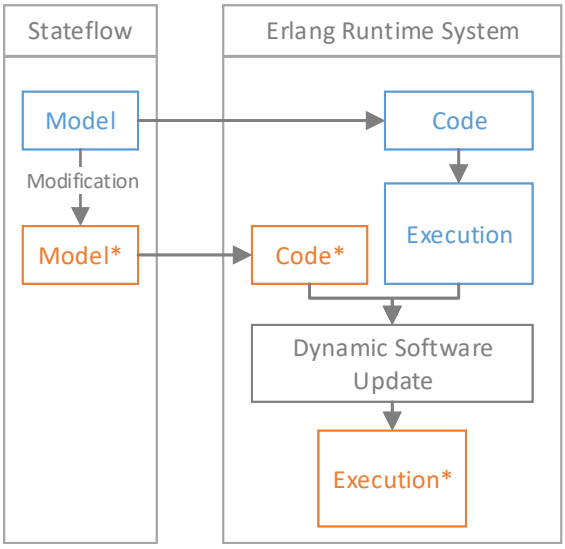

Fig. 1. The concept of reprogramming the controller at runtime.

instant messaging or banking, where the code has to be maintained while the system is running.

Erlang is both a programming language and a runtime system that supports Dynamic Software Updating also called Hot Code Upgrade. It is wildly used for applications in telecommunication, e-commerce and instant messaging. Armstrong et al. (2011) show Erlang's key features like concurrency and distribution in large scale applications, as well as soft real-time capability and robustness.

Using Erlang's advantages in new areas is attractive, especially in industrial automation, where fault-tolerant, distributed, (soft) real-time, and highly available applications are needed too.

The approach of this work considers controllers, that are modeled as parallel finite state machines. This style of

⋆ 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.

modeling is widespread. The method presented in this paper has been applied to Stateflow as an illustration, but could easily be extended to other modeling languages based on finite state machines, e.g. GRAFCET. The method consists of two parts: Firstly, converting the model to code, that is executable in ERTS. And Secondly, to perform a DSU between such Erlang executables.

In the second section of this paper background information is provided. The third section explains the proposed method in detail. This method is then applied in a case study in section four. Finally, in the fifth section this method is evaluated and topics for further research are presented.

2. BACKGROUND

2.1 Related Work

In the area of distributed systems, reconfigurable designs are an interesting topic. To put it simple, the idea is to ease modifying a system by using modular, distributed software architectures. When reconfiguring the system physically, software components can be reused and changes are only applied locally.

A more difficult task is to reconfigure the system at runtime, also referred to as *Dynamic Reconfiguration*. This is strongly related to Dynamic Software Update (DSU), because it includes changing the code at runtime. Whereas, DSU means even more, like bug-fixing or code optimization at runtime. Brennan et al. (2008) describe different approaches for dynamic reconfiguration and identify basic requirements. Further concepts are presented by Wahler et al. (2009), Wahler and Oriol (2014) and Atmojo et al. (2017).

This work makes use of Erlang's capability for DSU and combines it with a common modeling style, here Stateflow respectively parallel finite state machines. A closely related approach is presented by Prenzel and Provost (2017). They consider IEC 61499 – a industrial standard for process and control systems –, that is used to model distributed systems. Both projects aim to investigate Erlang's usability for industrial applications in combination with existing modeling styles.

2.2 Stateflow

Stateflow is an environment for modeling and simulating decision making systems. Finite state machines can be modeled including hierarchy, parallelism and history. Stateflow state charts can be used in combination with MATLAB Simulink models to represent hybrid systems, e.g. a decision making controller interacting with a continuous plant.

2.3 Modeling of State Machines

In the process of developing a controller the first step is to model the desired control behavior. The method proposed in this paper supports parallel finite state machines as a control model. Actually, those models are only a subclass of Stateflow charts, which can extend to higher complexity

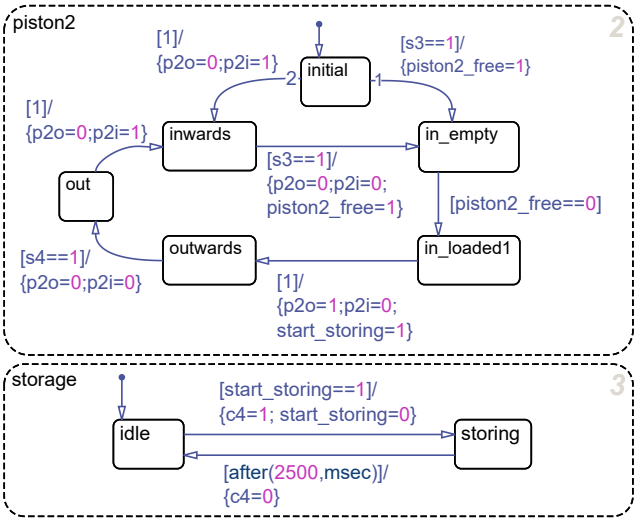

Fig. 2. Detail of a Stateflow model: parallel state machines (dashed outline), containing the sequential substates (solid outline) and transitions labeled with [condition]/{actions}; the execution order is placed in the top right corner of the parallel states.

and multiple abstraction levels. Therefore some Stateflow features are not yet supported.

Fig. 2 shows an example of the supported model class as a set of parallel state machines (in the following briefly state machines). These state machines are flat and consist of sequential states (in the following briefly states). Within a state machine, the states are linked with transitions labeled with a condition and an action. The model class has the following characteristics:

- The state machines are executed in a given execution order to ensure deterministic behavior.
- A state can have several outgoing transitions. Again, a execution order ensures determinism in case of more than one transition being enabled.
- Transitions are labeled with a condition and a transition action.
- Conditions are boolean expressions and can consist of equality expressions on data variables, e.g. $s3 == 1$, or temporal expressions referring the moment of entering the state, e.g. $after(2500, msec)$.
- Actions are compositions of assignations to data variables, e.g. $c4 = 1$.
- Data variable can be input, output or user defined variables. Input variables can only be accessed for reading, whereas the others can also be written.
- Each state machine has an initial state.

As stated above, some features of Stateflow state transition diagrams are not yet supported:

- Junctions
- Hierarchy (exceeding the two described levels)
- History Junctions
- Events
- Condition Actions
- State Actions

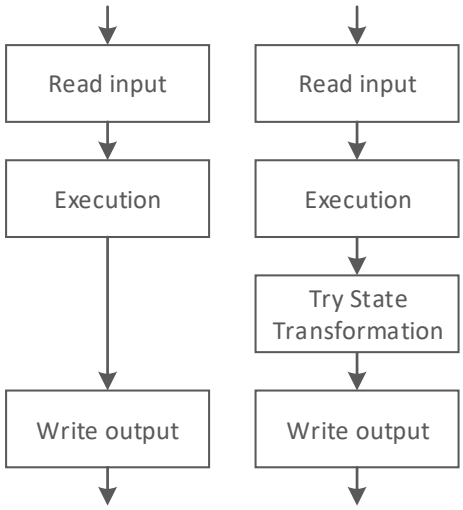

Fig. 3. (left) cycle in normal execution mode (right) cycle in update mode: extended with state transformation attempt.

2.4 Dynamic Software Update

Erlang code is structured in translation units, called modules. A running Erlang application consists of several modules and also processes, which execute the functions stored in the modules. Under an update scenario, a module is modified, compiled and loaded to ERTS, where a code server holds the old and new software version of the module. There are several ways to perform the update itself regarding when and how the processes switch to the new software version.

One way is to stop a process and restart it with the new software version. A more advanced way is to switch the process to the new software version in the sense of DSU. This might be necessary, when the running process contains important data, e.g. values of variables or references to other processes. This information is referred to as the state of the process. According to Seifzadeh et al. (2013), it is very important to transform the state of the process to the new software version. The update might have affected the data structure.

3. METHOD

3.1 Erlang Code Generation and Execution

As shown in Fig. 1, the model is translated to Erlang code. Therefore, two types of information are extracted from the model. On the one hand, the information to initialize the controller, i.e. the set of all initial states. On the other hand, the information to execute the model, i.e. the state transition information. The state of the controller, respectively the state of the process as introduced in the previous section, is the set of all active states.

Before the execution starts, the controller is initialized with the set of initial states. Then the controller starts executing cyclically as shown in Fig. 3 (left). Firstly, the input is read from the plant. Secondly, the controller is executed according to the state transition information. Finally, the output is written to the plant.

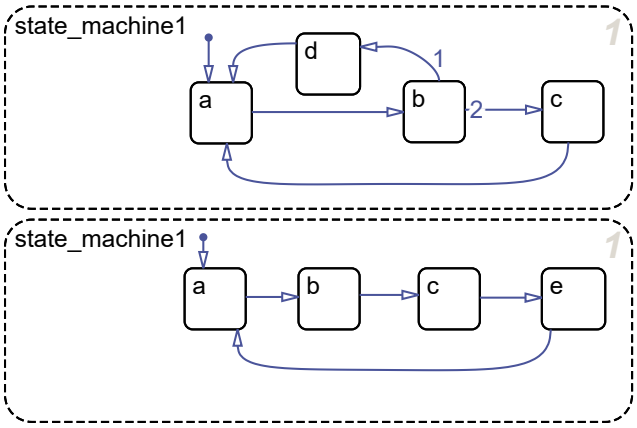

Fig. 4. The state machine from version 1 (top) and version 2 (bottom) are paired. Matching candidates for a transformation are a, b or c. In d the transformation is not possible.

As mentioned before, the execution of the Stateflow model happens in a strictly sequential manner to ensure determinism. Thereby, the cycle time and the memory consumption grow linearly in the worst case with the number of:

- state machines per model
- outgoing transitions per state
- components per condition
- assignations per transition action

Because of the strict sequential execution of Stateflow models, Erlang's capability for concurrency and distribution cannot be fully utilized. It would indeed be possible (in the sense of the IO and final state determinism) if the subsystems were independent.

So far, the code generation was only discussed with respect to the state transition behavior, but data variables are also part of the controller. For time reasons, data variables are not considered for automatic code generation and updating. It is assumed that updating does not affect the variables respectively future version work on the very same set of variables. Therefore a data server is written manually to manage variables and link them to physical connectors. However, it does not seem to be too complex to achieve automatic code generation and DSU of the data server.

3.2 State Transformation of Finite State Machines

In the following, the state transformation of parallel Finite State Machines is presented as well as a condition for when it can be performed.

Two state machines are said to be paired, if they have the same name. States within these state machines are called matching, again, if they have the same name. It is assumed that matching states represent the same physical state of the plant. Under this assumption, a condition for state transformation can be defined: for all paired state machines, there must exist a state in the new version, that matches the active state in the old version.

When this condition is satisfied, the state transformation splits up into three different cases:

- If a state machine exists in both versions, i.e. is paired, the active state is retained, as it has a matching counterpart in the new version. This is reasonable, because it was assumed, that matching states represent an equivalent physical condition. Fig. 4 shows different possible matching candidates for transformation.
- If a state machine only exists in the old version, i.e. it was deleted, the active state is also deleted.
- If the state machine only exits in the new version, i.e. it was added, it is started in the initial state.

The set of retained and initialized states is the state of the controller after the transformation.

It has to be mentioned, that designing updates requires some expert knowledge. Careless use might lead to undesired behavior. Especially matching of inappropriate states or deletion of states might cause data inconsistency or data loss.

3.3 Dynamic Software Update Process

In Fig. 1 the Process of DSU is displayed. The controller is executing an old software version. In parallel a new version is loaded to the code server. On a user command, the system switches from normal execution to an update mode, shown in Fig. 3 (right). The normal execution cycle is extended with a state transformation attend.

Thereby, the state transformation condition is evaluated, as defined in the previous section. If the condition holds, the state can be transformed to the new version and the update is successful. The following execution cycle will run with the new software version in normal execution mode. Otherwise, the system stays in update mode and retries state transformation until the update is successful.

Thus, the update will not be performed immediately, but is deferred until a transformable state of the controller is reached, i.e. a state where the state transformation condition holds. This might not always be possible. It would be useful to introduce Updateability as a new term to describe a successful update scenario:

- A transformable state exists and can be reached
- A transformable state will be reached within n cycles

Further analysis on the Updateability aspect is not in the scope of this work, but will be considered in future work.

4. APPLICATION

The previously presented method is applied to an educational plant as a proof-of-concept and to evaluate its performance.

4.1 Hardware Setup

An educational plant [1], shown in Fig. 5, is used to build a demonstration application. In Fig. 6 the structure of the plant is illustrated in detail. The u-shaped band conveyor transports workpiece blocks. At the corners pistons push the blocks to the next conveyor segments. On two machining stations a processing can be simulated. Some positions on the band conveyor are equipped with light sensors to detect a workpiece.

[1] fischertechnik – Indexed line with 2 machining stations

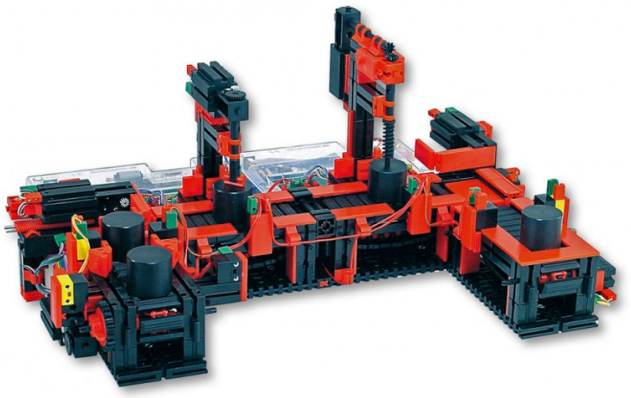

Fig. 5. Plant for demonstration application. (Source: IKH DIDACTIC SYSTEMS)

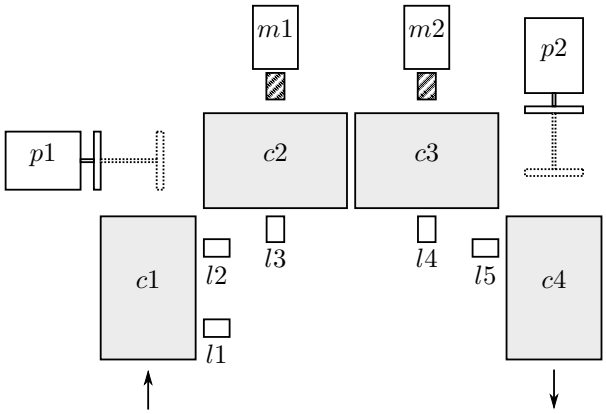

Fig. 6. The U-shaped band conveyor in detail: The independent conveyor segments cX transport workpieces. Pistons pX move the workpieces around the corners. At to machining stations mX a processing can be simulated. At some positions the conveyor band is equipped with light sensors lX to detect workpieces.

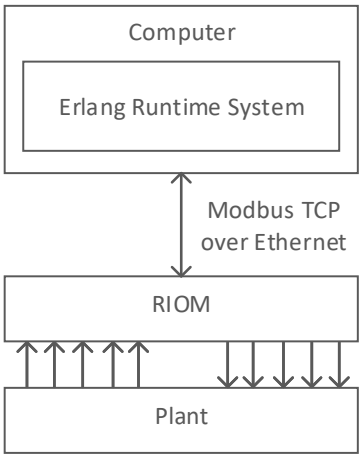

Fig. 7. Hardware architecture of the demonstration application.

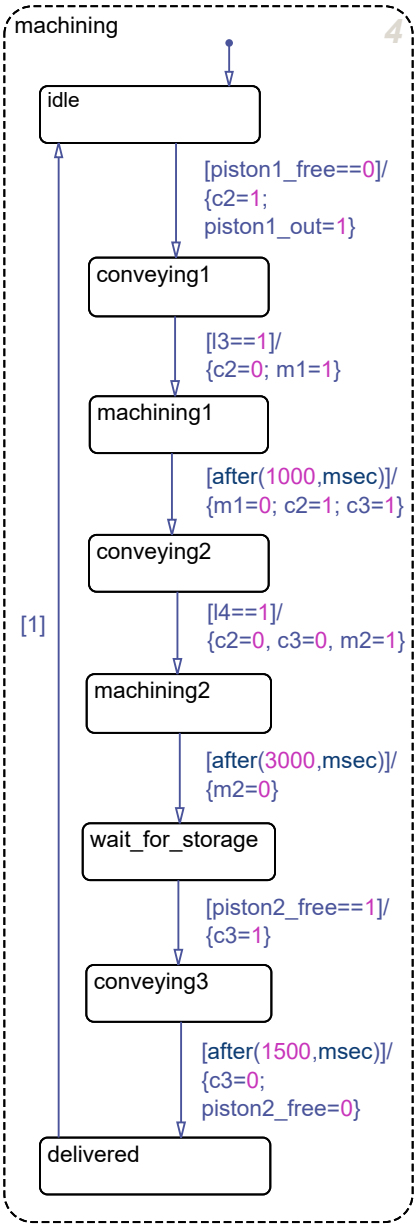

Fig. 8. This state machine controls the main line of the plant including both machining stations before the update.

A remote I/O module [2] (RIOM) is connected to the plant's sensors and actuators. The RIOM and a computer communicate with the Modbus TCP protocol over Ethernet. On the computer ERTS runs the controller software. Fig. 7 illustrates this architecture.

4.2 Case study Machining Process

In the first software version four state machines build the controller. The machining process itself is modeled with the state machine *machining*. It controls the main line of the plant including the machining stations. The three other state machines control the supply conveyor, the storage conveyor and the pistons.

[2] PHOENIX CONTACT – ILB ETH 24 DI16 DIO16-2TX

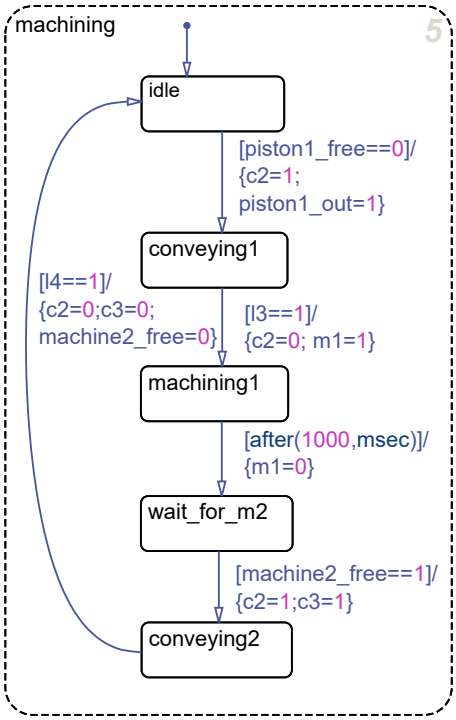

Fig. 9. This state machine controls the line in the area of the first machining station after the update.

Fig. 8 shows *machining* in the first software version. Step by step, the workpiece is handled. As soon as the piston pushes a workpiece on the band, it is transported to the first machining station. After the machining it is transported to the second machining station. The workpiece is transported to the storage system, when the second machining is finished and the storage is ready to receive the product.

In this case study, the user starts the production system with the first version of the controller. After some time, the user detects that the strictly sequential execution in the *machining* process could be optimized. In this version, the first machine is idle as long as the controller manages the second station. An upgrade of the Stateflow model will allow the two machines to work concurrently, thus the throughput will be increased.

To perform this upgrade, a new controller version is modeled. The three state machines for supply and storage conveyors and pistons are transfered to the new version without modifications. The state machine *machining* is modified and will only manage the machining process on the first station. Another state machine *machining2* is added to control the second machining station. Fig. 9 and Fig. 10 show the new state machines. For the three unmodified state machines and the added state machine the state transformation is possible in any case. The paired state machines *machining* (represented by Fig. 8 and Fig. 9) share 4 matching states: *idle*, *conveying1*, *machining1* and *conveying2*. A matching state and therefore a transformable state of the controller will eventually be reached, because the *machining* process in the first version is cyclical.

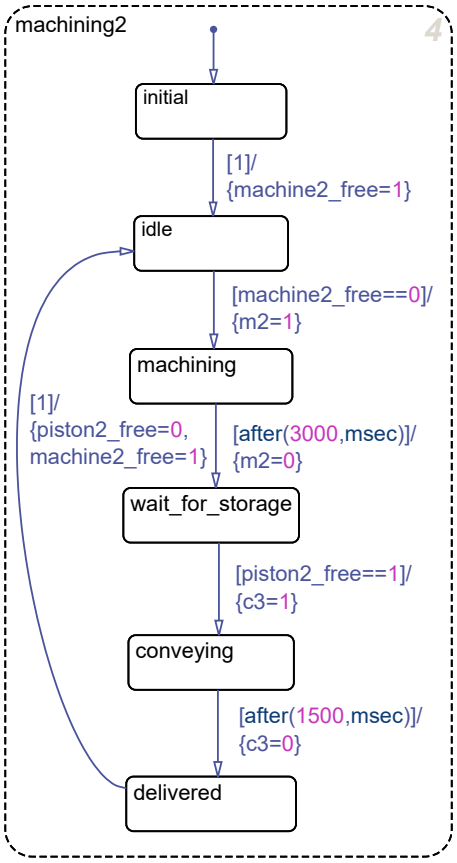

Fig. 10. This state machine controls the line in the area of the second machining station after the update.

4.3 Experimental Results

Experiments show, that the plant can be controlled with both software versions. The modeled behavior is executed without any noticeable delays. The update is easy to apply and the production is not affected in any way. Furthermore, the throughput could be increased by the update.

5. CONCLUSION

In this paper, a novel method for controlling industrial automation plants was presented, including a conceptional easy way to model a controller and reprogram it in a running system. Furthermore, a possible application of ERTS to industrial automation tasks was given. In a case study the method was applied to a real system with promising results.

So far, the data server is written manually and cannot yet be updated as mentioned in Sec. 3.1. It would be interesting to extend the code generation and DSU to the data server.

The aspect of Updatability from Sec. 3.3 could be investigated with reachability analysis. It is important to know, if an update will be successful before trying to apply it to a running system.

As mentioned before, the method presented in this paper has been applied to Stateflow as an illustration, but could be extended to other modeling languages based on finite state machines. This seems to be attractive especially for modeling languages, that support independent state machines. The concurrency capability of Erlang could be utilized and the performance of the controller improved.

For an industrial automation system real-time performance is essential. On the one hand, the hardware architecture presented in Sec. 4.1 has to be improved. Firstly, a real-time fieldbus communication protocol such as EtherCAT should be used instead of Modbus TCP and secondly, the computer system should support real-time applications. On the other hand, the software has to be modified. An important point to remind is that Erlang supports soft real-time but does not guarantee hard real-time. Nicosia (2007) shows how to improve Erlang's soft real-time performance to hard real-time.

Generally speaking, the focus of further research will be placed in applying the method to large scale systems, to determine scalability and performance of this method under realistic circumstances.

REFERENCES

Armstrong, J., Virding, S., and Williams, M. (2016). Erlang user's guide and reference manual.

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

Armstrong, J., Däcker, B., Lindgren, T., and Millroth, H. (2011). Open-source Erlang - white paper.

Atmojo, U., Salcic, Z., and Wang, K.K. (2017). Dynamic online reconfiguration in manufacturing systems using SOSJ framework. In *IEEE International Conference on Industrial Informatics (INDIN)*. doi: 10.1109/INDIN.2016.7819249.

Brennan, R.W., Vrba, P., Tichy, P., Zoitl, A., Sünder, C., Strasser, T., and Marik, V. (2008). Developments in dynamic and intelligent reconfiguration of industrial automation. *Computers in Industry*, 59(6), 533–547. doi:10.1016/j.compind.2008.02.001.

Ericsson AB (2016). *Erlang/OTP System Documentation*, 8.0 edition.

Nicosia, V. (2007). Towards hard real-time Erlang. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang*, 29–36.

Prenzel, L. and Provost, J. (2017). Dynamic software updating of IEC 61499 implementation using Erlang runtime system. In *20th IFAC World Congress, 2017*.

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.

Wahler, M. and Oriol, M. (2014). Disruption-free software updates in automation systems. In *19th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2014*. doi: 10.1109/ETFA.2014.7005075.

Wahler, M., Richter, S., and Oriol, M. (2009). Dynamic Software Updates for Real-Time Systems. *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades HotSWUp 09*, (October). doi: 10.1145/1656437.1656440.