An interactive learning approach on digital twin for deriving the controller logic in IEC 61499 standard

Midhun Xavier *, Victor Dubinin † Sandeep Patil*, Valeriy Vyatkin* ‡
* Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden
† Independent researcher
‡Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland

Email: midhun.xavier@ltu.se, dubinin.victor@gmail.com, sandeep.patil@ltu.se, vyatkin@ieee.org

***Abstract*—In this paper, we describe a method to automatically derive the controller for an automated process by an interactive learning approach using a simulation model developed in Visual Components 3D simulation software. The latter is used to record the events of the processes and the controller is generated as an IEC 61499 function block. To create different process scenarios, the actuator signals are triggered manually in appropriate order. The controller logic in Petri net is derived by process discovery algorithms with help of recorded events and conversion of Petri net to IEC 61499 function blocks is done by a software tool configured with a set of transformation rules.**

***Index Terms*—cyber-physical automation systems, IEC 61499, Process discovery, Visual Components**

I. Introduction

Design and validation of Cyber-Physical system's (CPS) [1] in industrial automation takes significant time due to the complex interaction of intelligent mechatronic components equipped with embedded control devices. The IEC 61499 standard [2] is considered a suitable approach to design such systems, but the manual development process is still considered to be error-prone, and it takes a lot of development effort.

Simulation models are very useful for validation, virtual commissioning purposes and to accurately describe the behavior of the automation systems. The controllers can be connected with the simulation models to understand better how the developed control code behaves on it.

In this paper we attempt to go a step further and use simulation models for deriving control code without programming, but as a result of interactive activity. The control code generated from this data-driven approach greatly reduces the engineering effort to program such systems.

The simulation models are easier to interact with than with the real production line. It is possible to record the events that occur in the simulation model. To record the sequence of events, one needs to generate a process scenario and trigger the controller signal manually. The recorded event log is used for the extraction of controller logic with the help of process mining algorithm. The powerful mining tools like ProM [3], DISCO [4], etc., can be of great help to automate this process.

The Visual Components [5] 3D simulation software is one of the commercially available tools. The actuator and sensor signals can be manually triggered with the help of this tool. The paper is organized as follows: Section II discusses the related work and problem statement. Section III explains the methodology for event log generation, extraction of controller logic in detail. Section IV describes the illustrative example of a simulation model and presents the result of the work. Finally, Section V concludes the paper and outlines future goals.

II. Related works and problem statement

The process mining technique [6] [7] is widely used to extract the business process models and improve the business by analyzing it. The business process models extracted from trace of activities gives a better representation of process scenarios and these models can be used for conformance checking. ProM [3] is one of the popular open-source tools used for the purpose of discovering the process models, conformance checking, verification of event log, visualization, and simulation of process models. Process mining is not widely explored in the field of industrial control systems but some of the researchers used this for conformance analysis of industrial control systems.

Data and Process Mining Applications on a Multi-Cell Factory Automation Testbed [8] explains how process mining can be applied in factory automation. In this approach, industrial control systems record event logs, create a process model using control flow algorithms and then it is used for the improvement of factory automation. Anomaly detection using event log is another area of research that can be applied in industrial control systems and this paper [9] describes anomalous activity in the control system by comparing the device log data with the process model. The deviation in event log can be identified by conformance checking and this helps to identify the cyber-attacks in the system. The outlier detection and alarm analysis of industrial plants can be done through the process mining technique [10]. The recorded event logs are stored in a database and the model is constructed using a process discovery algorithm. The verification of the event log with respect to the derived process model is used for outlier detection.

Paper [11] explains extraction of the formal model of plant from event logs for the purpose of verification. The formal model of plant from traces expressed in SMV is verified by CTL specifications with the help of NuSMV symbolic model

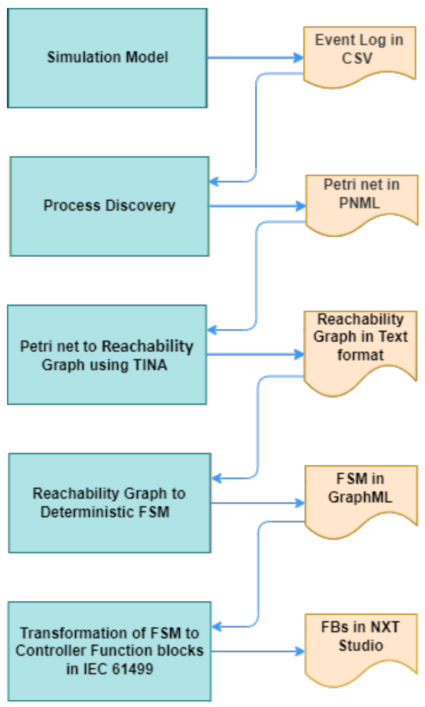

Simulation Model
Event Log in CSV
Process Discovery
Petri net in PNML
Petri net to Reachability Graph using TINA
Reachability Graph in Text format
Reachability Graph to Deterministic FSM
FSM in GraphML
Transformation of FSM to Controller Function blocks in IEC 61499
FBs in NXT Studio

Fig. 1. Methodology

checker. The process mining application is not all explored to derive the control logic or plant model with the help of recorded events of the system. In this paper, we propose a methodology to derive the control algorithms automatically from the traces of the desired behavior of the system.

III. Methodology

The Figure 1 shows the workflow which starts with recording the events with the help of a simulation model. The user interacts with the simulation model by manually controlling signals and each event occurred in the system is recorded in event log in CSV format. The event log contains the process information about the system, and it is used to extract the process model in terms of Petri net [12] using process discovery algorithm. The Petri net in Petri Net Markup Language (PNML) format is given as input to TINA [13] tool for the stepwise simulation and conversion of Petri net to reachability graph. The reachability graph is converted to a deterministic finite state machine and its representation in GraphML helps to visualize the process model and finally the implementation of Controller function block is done with the help of transformation rules. The detailed explanation of the methodology is given in the following subsections III-A, III-B and III-C.

A. Event log generation

The plant model is constructed in the Visual Components 3D environment. The simulation model works according to the actuator signal values and it shows working behavior of the system as well as the output as sensor values. Here, we are manually changing the actuator signal values and making

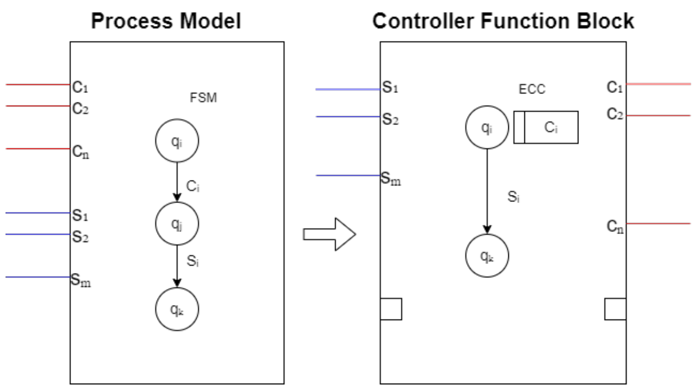


Fig. 2. Process Model to Function Block Interface Transformation

the system work in accordance with the process scenario. If we want to run a simulation model in a different way, then it is possible to generate such a process scenario by manually triggering the actuator signals. The simulation model developed in the Visual Component tool helps to record these events. The event log records the 'caseId' for each process run, global state of the system, time and component at which the event occurred, and finally records signal and its corresponding value. The detailed description and representation of example log construction is explained in Section IV.

B. Process model discovery from event log in terms of Petri net

The recorded event log consists of overall behavior of each process scenario executed in the simulation. To extract the process behavior from the event log, we use process discovery algorithms. Most of the Process discovery algorithms extract the process models in Petri net format [14] and it is easy to understand and visualize the system behavior. The alpha algorithm [15] helps to derive the process model from the recorded event log. The ProM tool has several process discovery algorithms so this tool can be used to process create models in form of Petri net. It is also possible to apply the alpha algorithm directly but ProM helps also in event log preprocessing, model visualization, conformance checking etc. The event log expressed in eXtensible Event Stream (XES) is given as input for the process discovery algorithm and conversion of CSV to XES is done with the help of ProM by mapping standard XES attributes under 'Case' and 'Event' columns.

C. Petri net to IEC 61499 function blocks for controller

The Petri net extracted from the event log is used to derive the controller logic and it is represented as the Execution Control Chart (ECC) of an IEC 61499 function block. As a rule, corresponding events are associated with the transitions of Petri nets, which are used to describe processes. The event can be treated as one of the necessary conditions for firing the transition. However, it should be noted that “nameless” transitions are possible in the net model, which can fire spontaneously, without being stimulated by any event.

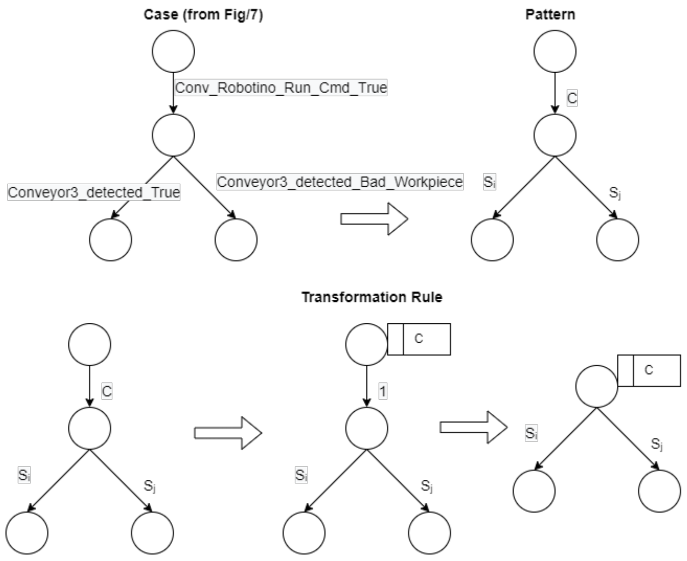


Fig. 3. FSM to ECC transformation rule

The reachability graph of a Petri net (if it is finite) is in fact a finite state machine (FSM). When it is constructed for Petri nets describing processes, the FSM can also have spontaneous transitions and, thus, in general the resulting FSM will be a non-deterministic one. To simplify implementation, a non-deterministic FSM should be transform to deterministic one. In this case, determinization will be reduced to getting rid of spontaneous transitions.

The TINA system [13] is used to analyze Petri nets and build a reachability graph. It can take Petri net in PNML format as an input and produces the reachability graph in text format. In order to convert the non-deterministic FSM to deterministic FSM and visualize the derived FSM, our software tool called 'Converter of TINA reachability to GraphML' is used. The deterministic FSM in XML format can be visualized using Gephi [16] or Yed [17] GraphML editors and it represents the whole process workflow of the entire closed-loop system.

1) TINA reachability graph to GraphML application:

This methodology can be used for the generation of the plant model and implementation of the monitor to identify any deviation from the existing process scenario. To incorporate these functionalities later, we need to extend this approach by introducing an intermediate FSM model (otherwise Petri net can be directly used to generate ECC).

Below the transformation of the textual representation of the reachability graph of a Petri net (which, in fact, is a FSM) in the TINA modeling system to the GraphML [18] format used to represent attribute graphs, is considered.

The conversion process can be divided into three phases:

1) Reading textual input data and generating internal program data structures on their basis.
2) Determinization of the FSM.
3) Generating the output GraphML file.

The presence of the second phase is explained by the fact that so called "spontaneous" transitions can exist in the finite state models. The spontaneous transition is not associated with any input signal and can be activated at any time. In finite automata theory, spontaneous transitions are marked with the symbol λ (lambda). In the TINA system, such transitions of Petri nets ported from the ProM system [19] are denoted as 'tau'. A FSM containing spontaneous transitions can be considered non-deterministic, so it becomes necessary to determine it. The advantage of the deterministic FSM is the simplicity of its implementation. In addition, when determining by getting rid of spontaneous transitions, the number of states of the FSM decreases.

In this work, two approaches to the determination of this type are used. The first method is based on the contraction of arcs representing spontaneous transitions, and the second one is based on the reachability principle. The essence of the first approach is as follows: two states connected by a lambda arc are merged into one state. In this case, the incoming and outgoing arcs of both states are combined, and the lambda arc is removed. This rule is applied until there are no lambda arcs left in the converted FSM. However, this method is not universal so it is applicable only in the case of a tree-like topology of lambda arc connection. In the second (universal) method, two states are contracted (merged) into one state only if one state is reachable from another state through a chain of lambda arcs.

The FSM model in TINA is represented as a set of descriptions of states and transitions in a text file with the .kts extension. Each state is described by the following text fragment (in EBNF form):

$state < numericIDofsourcestate >$

trans$< inputsignalname >$ /
$< numericidentifieroftargetstate >$

The structure of the description of the FSM model in the GraphML format is as follows. First, there is a description of the attributes of the graph model using the $< key >$ tag, for example:

$< keyattr.name = "label"id = "label"attr.type = "string"for = "node"/ >$

$< keyattr.name = "EdgeLabel"id = "edgelabel"$
$attr.type = "string"for = "edge"/ > 4$

In this case, the label attribute of string type is described for graph vertices, as well as the edgelabel attribute of string type is described for graph arcs. Next, there is a set of $< node >$ tags that describe the vertices of the graph. In the automaton interpretation, these vertices are states of the FSM. Example description below:

$< nodeid = "8" >< datakey = "label" >$
$8 < /data >< /node >$

To designate transitions (arcs) between automaton states the $< edge >$ tag is used, which contains the following attributes: id is a transition identifier; source is an identifier of the source state of the transition; target is an identifier of the target state of the transition. The $< edge >$ tag has a nested $< data >$

tag that defines the name of the transition. The example of a transition description is below:

$< edgeid = "8" target = "0" source = "9" >$
$< datakey = "edgelabel" > Repeat < /data >$
$< /edge >$

To represent the result FB, a special FB markup language is used, described in the second part of the IEC 61499 standard. The $< InterfaceList >$ tag with nested $< EventInputs >$ and $< EventOutputs >$ tags is used to describe a FB interface, which consists of input and output events. A separate event is described by $< Event >$ tag. The $< BasicFB >$ tag describes a basic FB. It has the embedded $< ECC >$ tag corresponding to an ECC. The ECC, in turn, consists of states ($< ECState >$ tags) and transitions ($< ECTransition >$ tags). Actions in states are defined by $< ECAction >$ tags.

Algorithm 1: FSM to ECC Transformation

```
Data: $M = (List < States >, List < Arcs >)$
$States = \{s_0, s_1, s_2, \dots\}$
$Arcs = \{a_0, a_1, a_2, \dots\}$
$s_i = (name_i, outSignal_i)$
$a_j = (name_j, source_j, target_j)$
$control\_signals = \{c_0, c_1, c_2, \dots\}$
Result: $TM = (List < States >, List < Arcs >)$
$TM\_States = \{List < States >\}$
$TM\_Arcs = \{List < Arcs >\}$
for arc in M_Arcs do
    $TM\_Arcs.add(arc)$
end
for state in M_States do
    $TM\_States.add(state)$
end
for arcs in TM_arcs do
    if $control\_signals$ has $arc.name$ then
        $index\_source = findIndex(TM\_States =>$
          $TM\_state.name = arc.source)$
        $TM\_state[index\_source].outSignal =$
        $arc.name$
        for arcSource in TM_Arcs do
            if $arcSource.source = arc.target$ then
                $arcSource.source \leftarrow arc.source$
            end
        end
        $TM\_Arcs.remove(arc)$
        $TM\_States.remove(arc.target)$
    end
end
```

2) Transformation of FSM to Controller FB in IEC 61499:

The obtained FSM expressed in GraphML from the reachability graph consists of a list of states and a list of edges or arcs.

$$M = (List < States >, List < Arcs >) \tag{1}$$

Each state consists of a name and output event signal. This output signal is initialized to null and it can be assigned to the control signal while transformation rules are applied.

$$s_i = (name_i, outSignal_i) \tag{2}$$

The edge has source state , target state and name (i.e. sensor or actuator signal).

$$a_j = (name_j, source_j, target_j) \tag{3}$$

To derive the controller logic, we need to apply the transformation rules on top of the derived FSM. Whenever an actuator signal value occurs at the edge of FSM then the following rules are applied.

- The corresponding edge and its target state are removed from the ECC.
- Actuator signal triggers as an output event at the source state of the removed edge
- The edge starts from the removed state is connected back to the source state of removed edge.

The interface of the controller is shown in the Figure 2. Controller function block takes all sensor signals as input events and produces actuator signals as output. The FSM of the process model is converted to ECC using the transformation rule. If control signal C_i is triggered from state q_i to q_k then the state q_j is removed from ECC and state q_i produces c_i event.

The Figure 3 explains the FSM to ECC transformation with the help of an example case from Figure 6. The Conv_Robotino_Run_Cmd_True is a control signal and FSM takes different paths according to the sensor values of the Conveyor3_detected_True and Conveyor3_detected_Bad_Workpiece and this pattern is expressed in the Figure 3. When the control signal appears on the edges of the FSM then the rule is straight-forward i.e. control signals replaced by '1' signal and their respective output place produce corresponding control signal as output. It is possible to simplify the ECC further by removing edges with condition "1" or "True" and its target state. The transformation from FSM to ECC is represented as the Algorithm 1.

IV. Case Study : Automatic Generation of Controller by Interactive Learning

A. General Description of simulation in Visual Components' 3D environment

The plant model constructed in Visual Components 3D manufacturing simulation software tool is shown in the Figure 4. The production system consists of a conveyor line, composed of 2 conveyor sections (conveyor3 and conveyor4), gripper and autonomous guided vehicle Robotino with a conveyor section mounted on top. Conveyor1 and Conveyor2 are not connected to the production line and never used in this experiment.

There are six actuators and four sensors on the production system. The conveyor line composed of Conveyor3 and Conveyor4 has one actuator each for to run

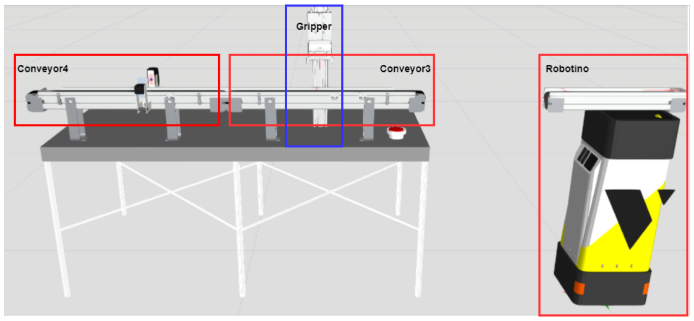
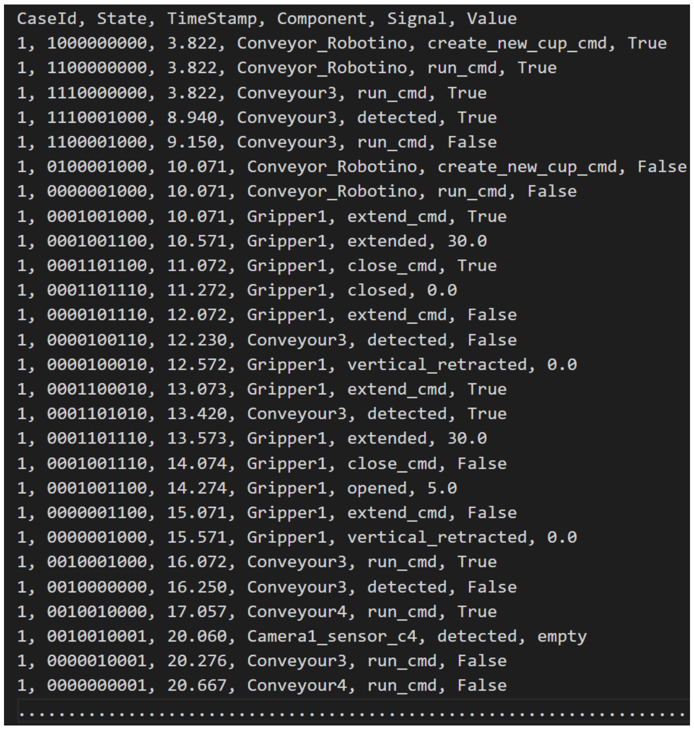


Fig. 4. Simulation in Visual Components



```
CaseId, State, TimeStamp, Component, Signal, Value
1, 1000000000, 3.822, Conveyor_Robotino, create_new_cup_cmd, True
1, 1100000000, 3.822, Conveyor_Robotino, run_cmd, True
1, 1110000000, 3.822, Conveyour3, run_cmd, True
1, 1110001000, 8.940, Conveyour3, detected, True
1, 1100001000, 9.150, Conveyour3, run_cmd, False
1, 0100001000, 10.071, Conveyor_Robotino, create_new_cup_cmd, False
1, 0000001000, 10.071, Conveyor_Robotino, run_cmd, False
1, 0001001000, 10.071, Gripper1, extend_cmd, True
1, 0001001100, 10.571, Gripper1, extended, 30.0
1, 0001101100, 11.072, Gripper1, close_cmd, True
1, 0001101110, 11.272, Gripper1, closed, 0.0
1, 0000101110, 12.072, Gripper1, extend_cmd, False
1, 0000100110, 12.230, Conveyour3, detected, False
1, 0000100010, 12.572, Gripper1, vertical_retracted, 0.0
1, 0001100010, 13.073, Gripper1, extend_cmd, True
1, 0001101010, 13.420, Conveyour3, detected, True
1, 0001101110, 13.573, Gripper1, extended, 30.0
1, 0001001110, 14.074, Gripper1, close_cmd, False
1, 0001001100, 14.274, Gripper1, opened, 5.0
1, 0000001100, 15.071, Gripper1, extend_cmd, False
1, 0000001000, 15.571, Gripper1, vertical_retracted, 0.0
1, 0010001000, 16.072, Conveyour3, run_cmd, True
1, 0010000000, 16.250, Conveyour3, detected, False
1, 0010010000, 17.057, Conveyour4, run_cmd, True
1, 0010010001, 20.060, Camera1_sensor_c4, detected, empty
1, 0000010001, 20.276, Conveyour3, run_cmd, False
1, 0000000001, 20.667, Conveyour4, run_cmd, False
..................................................................
```

Fig. 5. Event log

the conveyor i.e. $Conveyor3\_run\_cmd = True$ for starting the conveyor and $Conveyor3\_run\_cmd = False$ to stop it. Likewise, Conveyor3 and Conveyor4 have one sensor each for detecting workpiece ($Conveyor3\_detected$ and $camera1\_sensor\_c4\_detected$). The Gripper component can move upwards and downwards with help of $Gripper1\_extend\_cmd$. In order to grab and release the workpiece, the Gripper component uses the $Gripper1\_close\_cmd$ actuator signal. It consists of two sensors, the $Gripper1\_extended$ sensor which helps to determine whether the gripper extended or not and the $Gripper1\_closed$ sensor to identify whether the workpiece is released or not. Robotino component creates a workpiece (a 3D object looks like a cup) and connects with the conveyor line to transfer the workpiece for further processing. The following table shows all actuator and sensor signals used for this experiment.

Actuator signals
Conveyor_Robotino_run_cmd_True
Conveyor_Robotino_run_cmd_False
Conveyour3_run_cmd_True
Conveyour3_run_cmd_False
Conveyor_Robotino_create_new_cup_cmd_True
Conveyor_Robotino_create_new_cup_cmd_False
Gripper1_extend_cmd_True
Gripper1_extend_cmd_False
Conveyour4_run_cmd_True
Conveyour4_run_cmd_False
Gripper1_close_cmd_True
Gripper1_close_cmd_False
Sensor signals
Conveyour3_detected_BadWorkPiece
Conveyour3_detected_True
Conveyour3_detected_False
Camera1_sensor_c4_detected_empty
Gripper1_extended_30.0
Gripper1_opened_5.0
Gripper1_vertical_retracted_0.0
Gripper1_closed_0.0

The 3D view in simulation gives better understanding about the system's behavior. It is possible to construct different processing traces by manipulating the actuator signals. This plant has two processing scenarios, the first scenario explains normal behavior of the system whenever a work piece arrives and the second one describes how the plant reacts whenever a bad work piece is detected.

The processing sequence of the plant is as follows:

1) Robotino creates a new 3D object of the workpiece (i.e. structure like a cup) and it transfers the workpiece to the conveyor3.
2) Conveyor3 starts running and stops at the point where Gripper can grab the workpiece. Gripper processes the workpiece and places it back to the conveyor3 and then the conveyor starts running, transferring the workpiece to conveyor4 and this process is repeated in a cyclic order.
3) Whenever a bad workpiece is detected then Gripper does not perform any actions.

B. Event log Description

The different traces in the event log are generated by following appropriate triggering of actuator signals manually. These events are recorded in CSV format as shown in the Figure 5 and it consists of six columns: CaseId, State, TimeStamp, Component, Signal and Value. CaseId is unique for each processing sequence scenario, State is the encoded combination signal values of sensors and actuators, TimeStamp represents the time at which the event is triggered and finally, the combination of Component, signal and value column helps to form a complete description of an activity of an event. The State is a string value constructed by joining the encoded signal values of sensors and actuators and it helps to

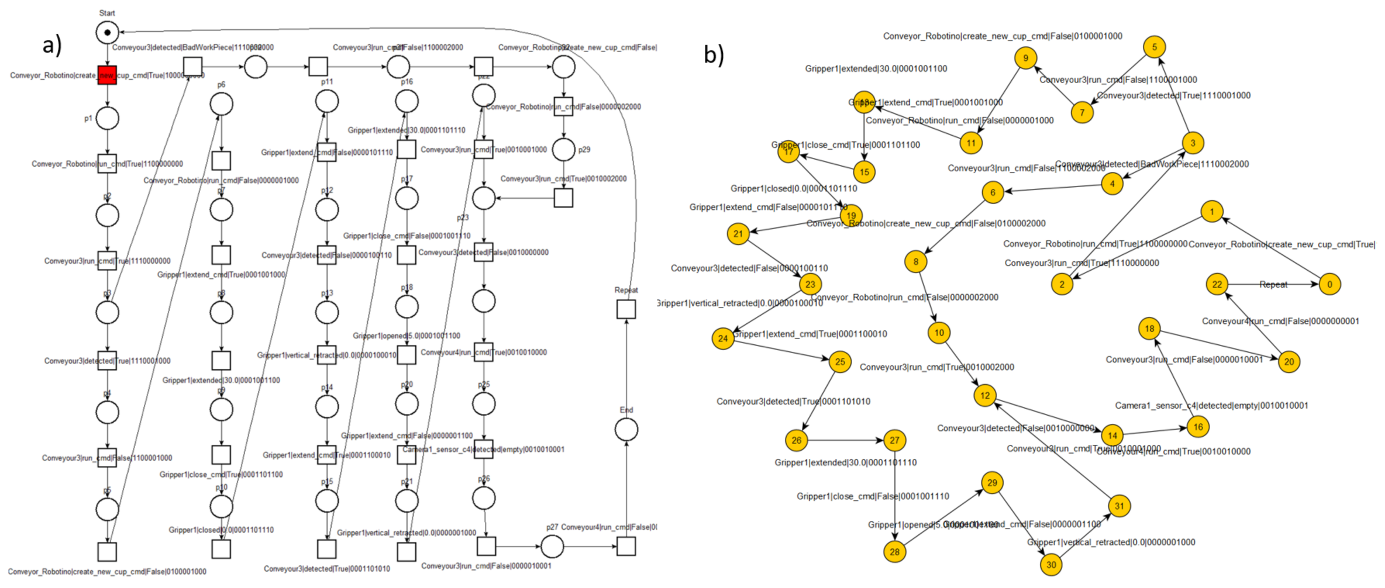


Fig. 6. a) The Petri net during its step-wise simulation observation b) The obtained FSM from Petri net

determine the unique status of the system. The event log is sorted according to the timestamp and is taken for the further process. The event log expressed in CSV format is converted to XES format because most of the process discovery algorithms accept this format. To convert CSV to XES format, it is necessary to specify the 'Case' and 'Event' columns. The 'Case' column is selected as 'CaseId' and the combination of component, signal, value and state as the 'Event' column.

C. FSM generation from event log

The process discovery method is used to identify the behavior of the system. In order to extract the process from the system, the process mining algorithm called alpha is applied on top of the recorded event log and it converts the process sequences to a Petri net. The obtained Petri net is shown in Figure 6 (a) during its step-wise simulation observation. The initial marking on 'Start' place is added for the simulation and conversion purpose of Petri net. To achieve the process to run in cyclic, a new transition 'Repeat' is added which connects from 'End' to 'Start'. The finite reachability graph can be considered as a Finite State Machine and the TINA tool helps to construct the reachability graph from the Petri net and save it in text format. The reachability graph expressed in text format is converted to deterministic FSM in GraphML format is done with help of our software tool 'TINA reachability graph to GraphML application'. FSM representation can be edited or visualized with help of GraphML editor tools like Gephi, Yed etc. The FSM representation in Yed GraphML editor is shown in the Figure 6 (b). The obtained FSM consists of two loops: First loop explains normal working behavior of the plant and the second one is the shorter one that explains 'How does the system behave whenever a bad workpiece arrives?' and the 'Repeat' signal makes the FSM run in cycles.

D. IEC 61499 representation of Controller

The FSM extracted from the event log describes the whole system behavior. To derive the controller in IEC 61499 standard, we use the transformation rules that were discussed earlier. The IEC 61499 function block interface of the controller is shown in the Figure 7 (a). It consists of sensor signals as input events and actuator signals as output events. The 'State' string can be neglected because it was used to identify the unique state in FSM. The ECC of the controller is shown in the Figure 7 (b). Whenever any actuator signal appears on the edges of FSM is transformed as output events and sensor signals remain the same as condition in ECC. The consecutive edges without any conditions in ECC (i.e. ideally "True" or "1" condition) can be merged into the common state and all actuator signals can be expressed as events in a separate 'Action' of ECC. The event 'Repeat' is changed to 'R' because 'Repeat' is a keyword in NxtStudio software.

V. CONCLUSION AND FUTURE WORK

The interactive learning approach helps to create controller in IEC 61499 function block automatically. The developed approach gives promising results. The process logic expressed in FSM helps to understand the process in a better way. The working of the same simulation model under different process scenarios can be implemented as a different controller function block. It is possible to drive the system by activating the actuator signals and this simple approach helps in deriving the same process logic.

The proposed approach requires the global state information of the system and it needs to be recorded on the event log whenever an event occurs. It is difficult to get a 'snapshot' of the states of all sensors when there is a current active event. To do this, we must poll all sensors but there will be considerable time delays in the transmission of the information via the

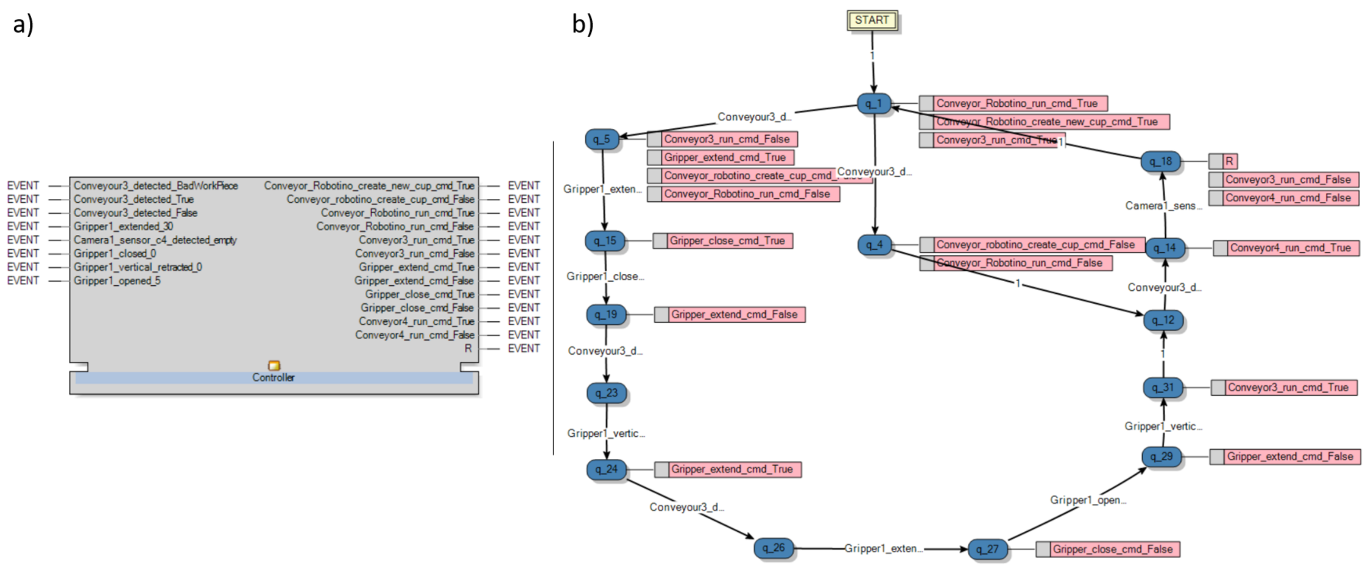


Fig. 7. a) Function block interface of Controller b) ECC representation in of Controller

network, so the data read from the sensors will be marked with different timestamps than the event's timestamp. The distributed controllers in the network consist of synchronized clocks with all the sensor readings where time stamped could be used but this makes the system more complex and sensor polling time delays remain the same.

This approach needs to be validated and tested in different simulation models or even in real systems to determine the accuracy of the extracted controller logic. The development of optimal ECC of complex systems consisting of multiple controllers is considered as the next step in the future. Formal verification of the system helps to identify the possible errors before deploying an automatically generated controller on the real system. The IEC 61499 formal verification tool chain [20] integrated with the proposed approach can be used for automatic verification and validation using specifications.

VI. Acknowledgements

This work was sponsored, in part, by the H2020 project 1-SWARM co-funded by the European Commission (grant agreement: 871743). We are also thankful to Visual Componenets (https://www.visualcomponents.com/) for providing with licenses for the 3D simulation tool.

References

[1] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. MIT Press, 2017.

[2] "IEC 61499-1: Function Blocks Part 1: Architecture," 2012.

[3] B. F. Van Dongen, A. K. A. de Medeiros, H. Verbeek, A. Weijters, and W. M. van Der Aalst, "The prom framework: A new era in process mining tool support," in *International conference on application and theory of petri nets*. Springer, 2005, pp. 444–454.

[4] C. W. Günther and A. Rozinat, "Disco: Discover your processes." *BPM (Demos)*, vol. 940, no. 1, pp. 40–44, 2012.

[5] "Visual components," https://www.visualcomponents.com/.

[6] R. Agrawal, D. Gunopulos, and F. Leymann, "Mining process models from workflow logs," in *Advances in Database Technology — EDBT'98*, H.-J. Schek, G. Alonso, F. Saltor, and I. Ramos, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 467–483.

[7] W. Van Der Aalst, "Process mining: Overview and opportunities," *ACM Transactions on Management Information Systems (TMIS)*, vol. 3, no. 2, pp. 1–17, 2012.

[8] N. Khajehzadeh, "Data and process mining applications on a multi-cell factory automation testbed," Master's thesis, 2012.

[9] D. Myers, S. Suriadi, K. Radke, and E. Foo, "Anomaly detection for industrial control systems using process mining," *Computers & Security*, vol. 78, pp. 103–125, 2018.

[10] J. Abonyi and G. Dorgo, "Process mining in production systems," in *2019 IEEE 23rd International Conference on Intelligent Engineering Systems (INES)*. IEEE, 2019, pp. 000 267–000 270.

[11] M. Xavier, J. Håkansson, S. Patil, and V. Vyatkin, "Plant model generator from digital twin for purpose of formal verification," in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2021, pp. 1–4.

[12] C. A. Petri, "Kommunikation mit automaten," 1962.

[13] B. Berthomieu*, P.-O. Ribet, and F. Vernadat, "The tool tina–construction of abstract state spaces for petri nets and time petri nets," *International journal of production research*, vol. 42, no. 14, pp. 2741–2756, 2004.

[14] W. M. Van Der Aalst and B. F. v. Dongen, "Discovering petri nets from event logs," in *Transactions on Petri nets and other models of concurrency vii*. Springer, 2013, pp. 372–422.

[15] W. Van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE transactions on knowledge and data engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[16] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Proceedings of the international AAAI conference on web and social media*, vol. 3, no. 1, 2009, pp. 361–362.

[17] R. Wiese, M. Eiglsperger, and M. Kaufmann, "yfiles—visualization and automatic layout of graphs," in *Graph Drawing Software*. Springer, 2004, pp. 173–191.

[18] "Graphml," http://graphml.graphdrawing.org/specification/dtd.html.

[19] "Prom," https://www.promtools.org/.

[20] M. Xavier, S. Patil, and V. Vyatkin, "Cyber-physical automation systems modelling with iec 61499 for their formal verification," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*. IEEE, 2021, pp. 1–6.