Automatic state machine reconstruction from legacy PLC using data collection and SAT solver

Daniil Chivilikhin, Sandeep Patil, *Member, IEEE*, Konstantin Chukharev, Anthony Cordonnier, and Valeriy Vyatkin, *Senior Member, IEEE*

***Abstract*—Nowadays an increasing number of industries are considering moving towards being Industry 4.0 compliant. But this transition is not straightforward: transfer to new system can lead to significant production downtime, resulting in delays and cost overruns. The best way is systematic seamless transition to newer and advanced technologies that Industry 4.0 offers.**

This paper proposes a framework based on automatic synthesis methods that learns the behavior of an existing legacy programmable logic controller (PLC) and generates state machines that can be incorporated into IEC 61499 function blocks. Proposed algorithms are based on Boolean satisfiability (SAT) solvers. The first algorithm accepts a set of noisy PLC traces and produces a set of candidate state machines that satisfy the traces. The second algorithm accepts error-free traces and synthesizes a modular controller that may be distributed across several physical devices. The toolchain architecture is exemplified on a laboratory scale Festo mechatronic system.

I. Introduction

Programmable logic controllers (PLC) have been the workhorse of industrial automation for decades. It is often the case that software of legacy PLCs is hard to understand since it was developed not following approaches and languages known to the current engineers. Sometimes the source code of PLC programs is not available at all.

Transition to the Industry 4.0 state of the technology could be hampered by the fear not to be able to integrate the legacy machines into modern automation systems, often referred to as cyber-physical systems (CPS). Substituting the legacy PLCs by modern devices may require complete redesign and retesting of their software, which is a major investment.

Most often, this is due to human factor: in any project people come and go. Thus, at some point an engineer responsible for an automation system may encounter a situation when: (a) he is not familiar with the implementation language of the control algorithm, (b) or the control algorithm source code is lost, (c) or the documentation is lost. In such cases it is hard to even support normal system operation, let alone to modify or modernize it.

D. Chivilikhin and K. Chukharev are with the Computer Technologies Laboratory, ITMO University, St. Petersburg 197101, Russia (email: chivdan,kchukharev@itmo.ru).

S. Patil is with the Department of Computer Science, Electrical, and Space Engineering, Luleå University of Technology, Luleå 97187, Sweden (email: sandeep.patil@ltu.se).

A. Cordonnier is with ENEDIS, Saint-Pierre-la-Palud 69210, France (email: anthony.cordonnier@enedis.fr).

V. Vyatkin is with the Department of Electrical Engineering and Automation, Aalto University, Espoo 02150, Finland, with the Department of Computer Science, Electrical, and Space Engineering, Luleå University of Technology, Luleå 97187, Sweden, and also with the Computer Technologies Laboratory, ITMO University, St. Petersburg 197101, Russia (e-mail: vyatkin@ieee.org).

This work suggests a method of re-implementing the logic of legacy PLCs in the form of state machines, with their subsequent deployment in function blocks of IEC 61499 standard, that takes advantage of the emerging data collection infrastructure of automated plants and automatic synthesis techniques. The method helps to replicate the functionality of a legacy PLC on a new technological basis by observing its behavior, collecting the corresponding data and then automatically generating a state machine controller in IEC 61499 format that mimicks the observed PLC behavior. The synthesized controller can thus become an integral part of automation systems compliant with Industry 4.0 standards.

Real-time data collection is becoming one of the most important aspects of a successful control system hierarchy. There were four main pillars in any industrial automation control system: (1) PLC Control system, (2) interaction between the PLC and Supervisory Control And Data Acquisition (SCADA), (3) interaction between SCADA and Manufacturing Execution System (MES), (4) Scheduled Maintenance.

But research has shown that the changing market needs have added a fifth pillar to the above process – data collection [1]. To address this issue, industry started with manual data collection, however soon it became inadequate because manual data collection is not real-time, is inaccurate, and many times biased. Real-time automated data collection has now become an important integral part of industrial automation ecosystems.

IEC 61499 [2], [3] is a new standard addressing the need for a new paradigm of distributed control systems. However, there are many legacy systems based on IEC 61131-3 [4]. Research [5], [6] has shown IEC 61499 to be a good fit for distributed system design and development, so in this paper the data collection application is built using the IEC 61499 design paradigm.

Preliminary results were published in [7], where we:

- developed a hardware and software architecture for collecting behavior traces from legacy PLCs in production;
- developed an algorithm for reconstructing controller logic in the form of a single *monolithic* state machine from PLC behavior traces which is based on translation to the Boolean satisfiability problem (SAT) and accounts for possible errors in collected traces;
- demonstrated the proposed solution on an example of a laboratory scale mechatronic system.

This paper extends results of [7] in two directions.

1) We improved the SAT-based controller logic reconstruction algorithm, which now enumerates synthesized solutions in the order of increasing number of errors in traces.
2) We developed a SAT-based algorithm for *modular* controller synthesis which enables distributing the generated control logic across several function blocks.

The proposed toolchain and algorithms can be used for (1) recording behavior traces from a legacy PLC, (2) automatically correcting errors in collected traces, and (3) generating a modular state machine controller based on corrected traces.

The rest of this paper is structured as follows. In Section II related research on state machine synthesis and data collection is reviewed. Section III describes the proposed hardware and software architecture for data collection from PLCs. In Section IV the suggested algorithmic approach to synthesizing a monolithic state machine from collected data is described. Section V introduces the algorithm for modular state machine synthesis. Section VI describes the case study on which the suggested approach was tested, threats to the validity of results are discussed in Section VII, and Section VIII concludes.

II. Related work

A. State machine synthesis

The most efficient methods for state machine synthesis are based on propositional encoding [8] – e.g., the problem is reduced/translated to SAT, and corresponding tools (SAT solvers) are used for finding a satisfying assignment of the SAT formula. Most SAT solvers implement complete decision procedures based on the conflict-driven clause learning algorithm [9] – apart from finding satisfying assignments, they can prove that no such assignment exists. Thus, using SAT solvers for state machine synthesis enables construction of minimal-sized models [10], [11], [12] – one can prove that a smaller model does not exist. Heuristic state merging methods are fast [13], [14], but, being based on incomplete decision procedures, cannot ensure the minimality of the result.

When behavior data is collected from a working PLC, noise is inherently introduced into collected behavior traces. The case of noisy behavior examples has only been considered in a couple of works: evolutionary [15] and SAT-based [16] algorithms for learning deterministic finite automata from noisy dictionaries. We believe that the reason for such a small number of works on state machine construction from noisy data is two-fold. First, it is due to vast computational complexity of the noisy problem(s). Indeed, in the worst case, the number of possible solutions increases exponentially with the number of allowed errors – one needs to consider all possible positions of each error. The second reason is probably that for many applications the presence of errors in data is not crucial: for example, probabilistic models generated for analysis of the system rather than for system replacement may include erroneous traces.

Considering modular state machine synthesis, the most closely related paper is [17], where a modular formal model of the plant is synthesized from behavior traces. There are two major differences from the method proposed in this paper: first, [17] builds nondeterministic state machines so it cannot be applied to synthesizing deterministic controller models; second, the modular decomposition in [17] is given as input to the algorithm and not synthesized automatically as in our approach.

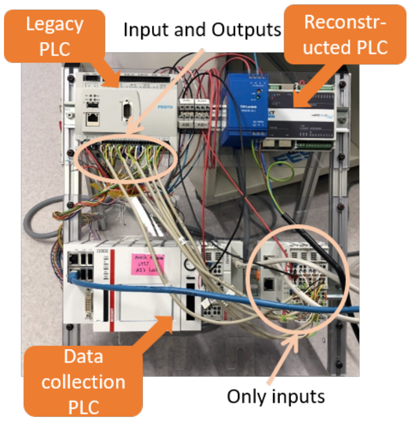

Fig. 1. Hardware architecture for data collection

B. Data collection

Data monitoring techniques have been widely researched for decades now [18]. The most common usages of data collection or machine monitoring are for the purposes of preventive maintenance [19], condition monitoring [20], machine availability [21] for downtime planning. Sensory monitoring techniques are well researched and established [22] and their challenges are well researched and addressed as well [23]. However, research on data collection for software optimization, software upgrading and reverse engineering is limited.

III. Hardware and software architecture for the proposed toolchain

The goal of PLC data collection is to record both the input and output values whenever there is a change in any of the input/output (I/O) values. Fig. 1 shows the hardware architecture for collecting the traces. In the used setup inputs of a secondary PLC running an IEC 61499 application that collected data traces were connected with the I/O's of the legacy PLC. Since only inputs were used, it was a safe way to directly connect the I/O's.

The data collection application logged data every time any I/O's of the legacy PLC changed state. In our example we only used Boolean data and it was easy to track the rising and falling edges of each of the I/O's of the legacy PLC. Each time the I/O's are sampled, a line with status (either 0 or 1) of all the I/O's is logged into a file stored on the PLC.

Fig. 2 shows the toolchain overview with five steps: (1) the black-box legacy PLC is physically connected with a special data collection PLC; (2) several use cases are run on the real physical system while the data collection PLC records traces of legacy PLC behavior; (3, 4) gathered PLC traces are used to synthesize an equivalent state machine controller in the form of one or several IEC 61499 function blocks; (5) the synthesized IEC 61499 function block controller is uploaded to a third

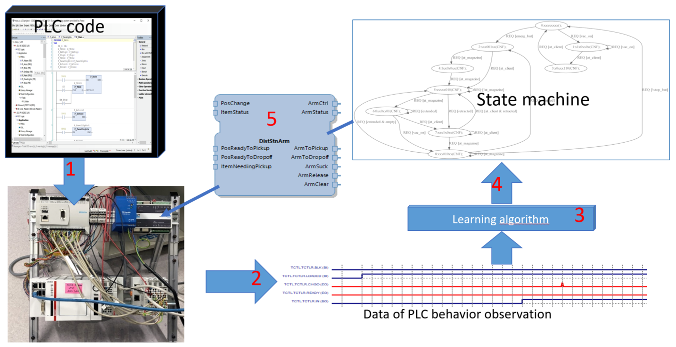

Fig. 2. Toolchain overview

PLC (shown in the upper right corner in Fig. 1) that is used to run the synthesized function block application.

IV. Monolithic state machine controller synthesis from noisy behavior traces

In this section we describe the proposed algorithm for synthesizing a monolithic IEC 61499 state machine from collected PLC traces $\mathbb{T}$. A trace is a list of elements. Each trace element includes a tuple of input variable values (later referred to as *input*) and a tuple of output variable values (later referred to as *output*). For example, below is a trace of three elements for five input and three output variables:

$$\langle 01010, 001\rangle; \langle 01011, 011\rangle; \langle 11010, 101\rangle.$$

The proposed method solves the problem of generating an IEC 61499 state machine (execution control chart of a basic function block) with the minimum number of states that repeats the behaviors represented by traces. The suggested method is based on an existing approach of state machine identification for IEC 61499 function blocks [24], but additionally accounts for possible errors in collected PLC traces.

The following model of an IEC 61499 state machine is used in this work. Denote X and Z the sets of Boolean input and output variables correspondingly. A state machine is a set of states connected with transitions; one state is marked as *initial*. A state is attributed with an output action which is represented by an output event that is generated upon entering the state, and an output algorithm that modifies output variable values. We assume that a state algorithm is represented by a string over the alphabet $\{0, 1, x\}$ of length $|Z|$. If the i-th element of the algorithm is "0" or "1", the i-th output variable is set to this value when the algorithm is executed; if the value of the algorithm element is "x", the value of the corresponding output variable is not changed in this state. A transition is labeled with an input event and a guard condition which corresponds to a tuple of values of all input variables, thus the number of transitions from one state is at most $2^{|X|}$ for each input event. In practice it is smaller and bounded by the number of input tuples observed in the traces.

This model can represent any ECC of a basic function block with Boolean input and output variables. Note that input and output events and their associations with the variables are neglected here, since legacy PLC behavior is not event-based. Therefore, only one input (`REQ`) and one output (`CNF`) events are used in the synthesized state machines.

A. Trace preprocessing and trace graph construction

Each trace is preprocessed: while two consecutive elements s_i and s_{i+1} have the same input value but different output values, we delete the element s_i. This partly deals with nondeterministic behaviors due to different scan cycles of the original PLC and the data collection system. Also, this procedure is safe and may not corrupt the input data. Naturally, traces collected from the PLC do not contain events, but only values of all input and output variables collected with a certain frequency. In this work events are assigned heuristically. We assign every input tuple a `REQ` input event. If elements s_i and s_{i+1} have different output values, s_{i+1} is assigned a `CNF` output event, otherwise, s_{i+1} is not assigned an output event.

After preprocessing, the trace graph (V, E) is constructed, in which vertices V correspond to outputs and edges E correspond to inputs. Note that we cannot use the traditional trace (scenario) tree used in [10], [24], [25] since due to noise in the traces a nondeterministic node of the tree may appear: node with different children for the same inputs. Thus, the trace graph is comprised of M (number of distinct traces) connected components. The set of vertices is divided into active and passive vertices $V = V^{\text{active}} \cup V^{\text{passive}}$: active ones have an associated output event (`CNF`), while passive ones do not. Similarly, active and passive edges E^{active} and E^{passive} are defined based on active/passive end node of an edge.

Work [26] also uses a trace graph for representing traces for a method of synthesizing plant behavior models in the form of nondeterministic state machines. However, in this paper we are dealing with controller logic and thus are interested in deterministic models. Hence, to account for possible errors, in addition to graph edges added due to the traces themselves, we add *hypothesis* edges in accordance with some *error model* of the PLC data collection system.

In this paper we consider a simple error model: we assume that the sole source of errors is the difference between scan cycles of the legacy PLC and the data collection PLC. Consider a situation when a change in input values of the legacy PLC leads to a change in output values. An ideal trace will first record the change in inputs and after that the change in outputs. However, in reality, since the scan cycles of the legacy and data collection PLCs are different and not synchronized with each other, it may happen that the changed output values are recorded before the input values, thus leading to a causality error in the recorded trace. More intricate and complex error models may also be considered.

In order to account for such errors, for each connected component of the trace graph we identify all pairs of consecutive nodes for which the output values are different. Consider one such pair of nodes (u, v) connected with edge uv with guard condition g_{uv} and also consider the child of node v – node l with corresponding edge vl and guard condition g_{vl}. Then we add an alternative edge from u to v but with guard condition g_{vl} (see Fig. 3 as an example). Denote the set of these additional alternative edges as $E^{\text{alternative}} \subset E^{\text{active}} \subset E$.

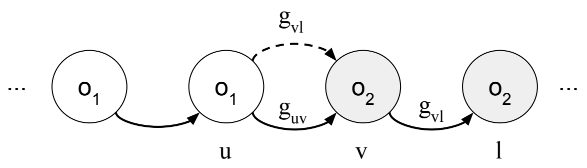

Fig. 3. Fragment of the trace graph describing the error model for PLC data collection: for the pair of nodes on the interface between different output values (o_1 and o_2) an additional (dashed) edge is considered

B. Translation to SAT

Denote N the number of states in the automaton. The main idea of SAT-based algorithms for automaton synthesis from noiseless data (e.g. [10], [16], [24], [27]) is to color the nodes of the trace graph (tree) with N colors so that the automaton produced by merging nodes with the same color into one state is deterministic and satisfies the traces. In other words, we are searching for a specific mapping of trace graph nodes and edges to automaton states and transitions. If the traces are noisy (contain errors) we have to ensure that only one of the edges connecting each pair of adjacent nodes of the graph is mapped to a transition in the generated automaton.

Colors of trace graph nodes are described by color variables $c_{v,i}$ for each node $v \in V$ and each color $i \in [0..N)$. *At least one* (ALO) and *at most one* (AMO) constraints are placed:

$$\mathrm{ALO}_i(c_{v,i}) \iff \bigvee_i c_{v,i}$$

$$\mathrm{AMO}_i(c_{v,i}) \iff \bigwedge_{i_1<i_2} (\neg c_{v,i_1} \vee \neg c_{v,i_2})\,.$$

Root nodes of the graph are attributed to the initial state 0: $\bigwedge_{\mathrm{isRoot}(v)} c_{v,0}$. For the sake of brevity we further omit explicitly describing AMO/ALO constraints on variables: we assume that they are implicitly added when Boolean variables are used to encode integer values.

Denote G the enumerated set of all unique inputs present on the edges of the trace graph. Transitions of the automaton are represented with Boolean variables y_{n_1,g,n_2} ($0 \le n_1, n_2 < N$, $0 \le g < |G|$) that encode whether the state machine includes a transition from state n_1 to state n_2 labeled with input g.

Specific selection of alternative edges of the trace graph is represented with variables $e_{u,v,g}$ ($u, v \in V$, $uv \in E^{\text{alternative}}$, $0 \le g < |G|$) – can the edge from node uv labeled with input g be used as a transition in the state machine. Since exactly one edge must be used for each pair of connected nodes, ALO and AMO constraints are placed on $e_{u,v,g}$.

For each active edge (u, v, g), if node u has color n_1, v has color n_2, and the edge can be used, the transition is defined:

$$\bigwedge_{(u,v,g)\in E^{\text{active}}} c_{u,n_1} \wedge c_{v,n_2} \wedge e_{u,v,g} \implies y_{n_1,g,n_2}.$$

For a passive edge there should be no defined transitions, and the color of the start vertex of the edge is propagated to its end vertex, encoding that the controller does not change state:

$$\bigwedge_{(u,v,g)\in E^{\text{passive}}} (c_{u,n_1} \implies c_{v,n_1} \wedge \bigwedge_{n_2} \neg y_{n_1,g,n_2}).$$

State algorithms are described by variables $d^0_{n,j}$ and $d^1_{n,j}$ – is the algorithm element in state n for output variable j zero or one, correspondingly. If both $d^0_{n,j}$ and $d^1_{n,j}$ are `False`, the element is interpreted as "x" (retain previous value). Denote $z_{v,j}$ the value of the j-th output variable in node $v \in V$. The following constraints encode the rules of transforming output variable values in accordance with graph edges:

$$\bigwedge_{0\le n<N}\ \bigwedge_{0\le j<|Z|} d^0_{n,j} \implies \neg d^1_{n,j}$$

$$\bigwedge_{(u,v,g)\in E^{\text{active}}}\ \bigwedge_{0\le n_1,n_2<N}\ \bigwedge_{0\le j<|Z|} c_{u,n_1} \wedge c_{v,n_2} \wedge$$

$$\wedge\, e_{u,v,g} \wedge y_{n_1,g,n_2} \implies \begin{cases} \neg d^1_{n_2,j}, & \text{if } \neg z_{u,j} \wedge \neg z_{v,j} \\ d^1_{n_2,j}, & \text{if } \neg z_{u,j} \wedge z_{v,j} \\ d^0_{n_2,i}, & \text{if } z_{u,i} \wedge \neg z_{v,i} \\ \neg d^0_{n_2,i}, & \text{if } z_{u,i} \wedge z_{v,i} \end{cases}.$$

Described constraints allow constructing an automaton that satisfies the given noisy PLC traces with respect to the selected error model. However, the resulting state machine is not unique since, in the worst case, the number of possible unique alternative edge choices is exponential. Also, not all synthesized state machines that correspond to the traces with respect to the error model are correct in terms of the ground truth – true noiseless behavior traces of the PLC, which we do not have access to.

Thus, in order to find the correct solution we need to enumerate all different state machines of minimal size that are consistent with traces – in other words, find all possible solutions and check them by some external means. For that an iterative algorithm is used, where on the i-th iteration constraints that prohibit previous choices of alternative edges in the trace graph are added:

$$\bigwedge_{0\le j<i} \left(\neg \bigwedge_{(u,v,g)\in E} e^j_{u,v,g}\right),$$

where $e^j_{u,v,g}$ is the value of $e_{u,v,g}$ found on the j-th iteration.

Note that in order to efficiently enumerate different state machines we need to maximally reduce all symmetries of the problem. Additional constraints are employed for further reduction of symmetries. First, we apply symmetry breaking predicates [16] which fix the enumeration of automaton states in the order they would be visited by the breadth-first search (BFS) algorithm. Second, we force all transitions not covered by traces to be self-loops [28]. Third, the number of distinct states reachable from each state is limited to constant K [24].

Fourth, the number of transitions in the state machine is fixed. For that we introduce variables $t_{n,g}$ ($0 \le n < N$, $0 \le g < |G|$) – whether there exists a transition from state n labeled with input g with the following definition constraints:

$$\bigwedge_{n_1,g} t_{n_1,g} \iff \bigvee_{n_2} y_{n_1,g,n_2}.$$

Then, auxiliary variables $t'_{i,r}$ ($0 \le i, r \le N|G|$) encode whether the number of defined transitions among the first i possible transitions equals to r. The following constraints define variables $t'_{i,r}$ based on the state machine:

$$\bigwedge_{0\le i\le N|G|} t'_{i,0}$$

$$\bigwedge_{\substack{0\le i\le N\\0\le g<|G|\\0\le j<N|G|}} \begin{cases} t_{n_1,g} \wedge t'_{n_1|G|+g-1,j} \implies t'_{n_1|G|+g,j+1} \\ \neg t_{n_1,g} \wedge t'_{n_1|G|+g-1,j} \implies t'_{n_1|G|+g,j} \end{cases} \tag{1}$$

Finally, the maximal number of transitions in the state machine is fixed to constant R:

$$\bigwedge_{0\le i<N|G|} \neg t'_{i,R+1}. \tag{2}$$

Note that the auxiliary variables $t'_{n,g}$ and constraints (1-2) described for limiting the number of `True` variables $t_{n,g}$ to constant R comprise a form of *cardinality* constraints [29] for Boolean variables. Such auxiliary variables and constraints can be formulated for applying cardinality to any set of Boolean variables, and we denote them as $\texttt{card}(\{t_{n,g}\}, R)$, where the first argument is the set of variables and the second argument is the constant representing the maximal number of `True` variables from the set.

Finally, we add constraints that limit the number of errors allowed in the behavior traces to constant ε by counting used alternative hypothesis edges of the trace graph:

$$\texttt{card}(\{e_{u,v,g} | (u,v,g) \in E^{\text{alternative}}\}, \varepsilon). \tag{3}$$

Described constraints define the following decision procedures of searching for an automaton that complies with noisy traces $\mathbb{T}$ by means of a SAT solver:

- with at most N states – $\texttt{findModel}(\mathbb{T}, N)$;
- and with at most K states reachable from each state – $\texttt{findModel}(\mathbb{T}, N, K)$;
- and with at most R transitions – $\texttt{findModel}(\mathbb{T}, N, K, R)$;
- and with at most ε errors in traces and previously found solutions S prohibited – $\texttt{findModel}(\mathbb{T}, N, K, R, S, \varepsilon)$.

If the underlying SAT solver call returns an UNSAT message, each of these `findModel` procedures return $\varnothing$.

The final procedure for constructing a state machine from noisy PLC traces is described in Algorithm 1. First, we try to find any automaton with the minimal number of states N. Second, we additionally minimize parameter K – maximal size of the set of states reachable from each state with one transition. Third, we minimize the total number of transitions R. Last, we search for all unique state machines with found parameters N, K, and R that use different alternative edge choices in the trace graph and have a fixed number of errors ε. Since the ground truth (correct traces) is not known, the final solution can only be found by checking generated solutions S by an external procedure – manual checking, simulation, model checking, testing, etc. Note that as the by-product the algorithm generates for each candidate solution a set of error-free behavior traces which are represented by specific selection of alternative trace graph edges. Described symmetry breaking approaches decrease the number of candidate solutions that have to be checked manually – either by loading the generated controller to the CPS, using formal verification, or with simulation testing. In addition, constraints (3) allow examining solutions in the order of increasing number of errors ε. In our case study this feature proved useful since the correct solution has been found for $\varepsilon = 1$, so only one solution had to be examined.

Algorithm 1: SAT-based synthesis of a set of monolithic state machines from noisy PLC traces

```
Data: noisy PLC execution traces $\mathbb{T}$
$N \leftarrow 1$
while findModel($\mathbb{T}$, $N$) $= \varnothing$ do
    $N \leftarrow N + 1$
$K \leftarrow 1$
while findModel($\mathbb{T}$, $N$, $K$) $= \varnothing$ do
    $K \leftarrow K + 1$
$R \leftarrow 1$
while findModel($\mathbb{T}$, $N$, $K$, $R$) $= \varnothing$ do
    $R \leftarrow R + 1$
/* find all minimal solutions
   consistent with traces, increasing
   the number of errors from zero     */
$S \leftarrow list()$, $\varepsilon \leftarrow 0$
while True do
    $A \leftarrow$ findModel($\mathbb{T}$, $N$, $K$, $R$, $S$, $\varepsilon$)
    if $A \neq \varnothing$ then $S$.add($A$)
    else
        $\varepsilon \leftarrow \varepsilon + 1$
    if $\varepsilon \geq |E^{alternative}|$ then
        break
return $S$
```

V. Modular controller synthesis

The proposed modular controller synthesis algorithm allows distributing control logic across several function blocks: it solves the problem of generating an IEC 61499 composite function block and underlying basic function blocks with corresponding ECCs, such that the composite function block repeats the behaviors represented by traces. A simple modular decomposition is employed: control is distributed across M modules, each module controls a subset of all output variables without intersections with other modules. Input variables are distributed arbitrarily: each module may use any input variables.

Main input data for the method are behavior traces $\mathbb{T}$, the number of modules M, and the number N which is here treated as the maximal number of states in each module. It is assumed that the traces do not contain errors. If initial traces have been gathered from a legacy PLC, this can be achieved by means of the algorithm described in Section IV, since the by-product of this algorithm is a set of error-free traces.

A trace tree $(V, E = E^{\text{active}} \cup E^{\text{passive}})$ is constructed using the traces. The main idea of the approach is to color the trace tree with N colors for each of M modules – thus, each vertex of the scenario tree has M colors, one for each module. The color $i \in [0..N)$ of node $v \in V$ for module $m \in [0..M)$ is represented by variable $c_{m,v,i}$. It is similar to variable $c_{v,i}$ used in Section IV, so we use the same letter; below we will use the same approach for other variables introduced in Section IV.

A. Modular decomposition, transitions, and guard conditions

The modular decomposition for output variables is encoded by mapping each output variable to a single module: variable $\theta_{m,i}$ denotes that output variable i is controlled by module m.

Transitions of the state machine for each module are encoded differently than in Section IV. Instead of labeling a transition with an input tuple (values of all input variables), a transition is here labeled with a guard condition of the form $\bigwedge_{i_l \in [0..|X|)} \xi_{i_l}$ where $\xi_{i_l} \in \{x_{i_l}, \neg x_{i_l}\}$ – conjunction of input variable literals, each positive or negative. This is done in order to enable modular decomposition of input variables. Each state of each module's state machine is allowed to have at most K transitions. Thus, transitions are encoded with three types of variables. Variable y_{m,n_1,k,n_2} denotes that the k-th transition from state n_1 of module m leads to state n_2. Variable $\alpha_{m,i}$ encodes the modular decomposition of input variables and denotes that input variable i is used in module m. Variable $\beta_{m,n,k,i}$ encodes the literal value of input variable i in the guard condition of the k-th transition from state n of module m.

Each guard condition can evaluate to `True` or `False` depending on inputs G. We say that a transition *fires* for some input from G if its guard condition evaluates to `True` on this input. To encode whether the k-th transition from state n in module m fires for some input $g \in [0..|G|)$ we introduce variable $f_{m,n,k,g}$ defined as follows:

$$f_{m,n,k,g} \iff \bigwedge_{\{i | G_{g,i}=1\}} (\alpha_{m,i} \implies \beta_{m,n,k,i}) \\ \wedge \bigwedge_{\{i | G_{g,i}=0\}} (\alpha_{m,i} \implies \neg\beta_{m,n,k,i}).$$

In other words, if for each variable i used in module m its literal in a guard condition is equal to the value of this variable for some input g, then this guard condition evaluates to `True`, and the transition fires for input g.

According to IEC 61499 the ECC follows the first transition for which the guard condition evaluates to `True` [2]. We introduce variable $\mathit{ff}_{m,n,k,g}$ that represents that the corresponding transition *fires first*: the guard condition for this k-th transition evaluates to `True`, while guard conditions of previous transitions $(0..(k-1))$ must evaluate to `False`. To encode this efficiently we introduce auxiliary variables $\mathit{nf}_{m,n,k,g}$ that denote that the transition does *not fire*:

$$\mathit{nf}_{m,n,0,g} \iff \neg f_{m,n,0,g}$$
$$\bigwedge_{1 \le k < K} \mathit{nf}_{m,n,k,g} \iff \neg f_{m,n,k,g} \wedge \mathit{nf}_{m,n,k-1,g}.$$

Then, first fired variables $\mathit{ff}_{m,n,k,g}$ are defined as follows:

$$\mathit{ff}_{m,n,0,g} \iff f_{m,n,0,g}$$
$$\bigwedge_{1 \le k < K} \mathit{ff}_{m,n,k,g} \iff f_{m,n,k,g} \wedge \mathit{nf}_{m,n,k-1,g}.$$

B. Correspondence between trace tree and controller behavior: transition firing, output algorithms

For each active tree edge (u, v, g) some transition of some module should fire, while for passive edges no transitions should be fired. We also want to minimize the number of modules that make a transition. For that we require that a transition is made by modules which are associated with output variables that have different values in u and v:

$$\bigwedge_{(u,v,g) \in E^{\text{active}} : z_{u,j} \neq z_{v,j}} \theta_{m,j} \wedge c_{m,u,n_1} \wedge c_{m,u,n_2} \implies \\ \implies \bigvee_{k} y_{m,n_1,k,n_2} \wedge \mathit{ff}_{m,n_1,k,g};$$

$$\bigwedge_{(u,v,g) \in E^{\text{passive}}} c_{m,u,n} \implies c_{m,v,n} \wedge \bigwedge_{k} \neg f_{m,n,k,g}.$$

In other words, for each active edge and each output variable that changes its value there exists a module that has a corresponding transition that fires; for passive edges no transitions are fired, and the child vertex is colored the same as the parent.

Next, consider the case when for an active edge some module m does not make a transition. The child node of the edge must be colored with the parent color:

$$\bigwedge_{(u,v,g) \in E^{\text{active}}} c_{m,u,n} \wedge \bigwedge_{k} \neg f_{m,n,k,g} \implies c_{m,v,n}.$$

Further, if there is a transition that fires for an input, then the child node of an edge is colored according to the transition:

$$\bigwedge_{(u,v,g) \in E^{\text{active}}} y_{m,n_1,k,n_2} \wedge \mathit{ff}_{m,n_1,k,g} \wedge c_{m,u,n_1} \implies c_{m,v,n_2}.$$

Transitions of each state machine are forced to be covered by the traces. Constraints ensure that each transition corresponds to at least one edge for which the transition fires:

$$y_{m,n_1,k,m_2} \iff \bigvee_{(u,v,g) \in E^{\text{active}}} c_{m,u,n_1} \wedge c_{m,v,n_2} \wedge \mathit{ff}_{m,n,k,g}.$$

Output algorithms attributed to states are encoded similar to the way employed in Section IV, with the exception that the module index is added to the corresponding variables $d^0_{m,n,i}$ and $d^1_{m,n,i}$, and that the left part of implication is augmented with the output mapping correspondence, together with the transition existence and firing conditions:

$$\bigwedge_{(u,v,g) \in E^{\text{active}}} c_{m,u,n_1} \wedge c_{m,v,n_2} \wedge \theta_{m,j} \wedge y_{m,n_1,k,n_2} \wedge \\ \wedge \mathit{ff}_{m,n_1,k,g} \implies \begin{cases} \neg d^1_{n_2,j}, & \text{if } \neg z_{u,j} \wedge \neg z_{v,j} \\ d^1_{n_2,j}, & \text{if } \neg z_{u,j} \wedge z_{v,j} \\ d^0_{n_2,i}, & \text{if } z_{u,i} \wedge \neg z_{v,i} \\ \neg d^0_{n_2,i}, & \text{if } z_{u,i} \wedge z_{v,i} \end{cases}.$$

C. Parameter minimization

We would like our modular controller to be as simple as possible, thus its parameters need to be minimized. First, minimization of the maximal number of states N in all modules is done the same way as in Algorithm 1. Second, we would like to minimize the total number of input variables used by the modules by limiting the total number of `True` $\alpha_{m,i}$ variables to constant A: $\texttt{card}(\{\alpha_{m,i}\}, A)$.

Third, we would also like to minimize the total number of states used in all modules, thus we add constraints that limit this number to constant $N' < M \times N$. In order to do this, we first introduce variables tr_{m,n_1,n_2} that denote that in module m there is a transition from state n_1 to state n_2:

$$\bigwedge_{m}\bigwedge_{n_1,n_2} tr_{m,n_1,n_2} \iff \bigvee_{k} y_{m,n_1,k,n_2}.$$

Then, we introduce variables $uc_{m,n}$ indicating that the m-th module *uses color* n. By definition of uc we state that the color is used whenever there is a tree vertex colored with it:

$$uc_{m,n} \iff \bigvee_{v} c_{m,v,n}.$$

If the color n_1 is used in module m and there is a transition in this module to state n_2, then color n_2 is also used:

$$uc_{m,n_1} \wedge tr_{m,n_1,n_2} \implies uc_{m,n_2}.$$

If color $n_1 > 0$ is used, then there must be a transition in the automaton that ends in state n_1:

$$uc_{m,n_1} \implies \bigvee_{n_2 \neq n_1} tr_{m,n_2,n_1} (n_1 > 0).$$

If color n_1 is not used, then larger colors are not used either:

$$\neg uc_{m,n} \implies \neg uc_{m,n+1}.$$

If either of colors n_1 or n_2 are not used, no transition may connect corresponding states:

$$\neg uc_{m,n_1} \vee \neg uc_{m,n_2} \implies \neg y_{m,n_1,k,n_2}.$$

Finally, if a color is not used, then output algorithms for the corresponding unused states are fixed:

$$\neg uc_{m,n} \implies \neg d^0_{m,n,j} \wedge \neg d^1_{m,n,j}.$$

The total number of states in the modular controller is limited to constant N' by constraints: $\texttt{card}(\{uc_{m,n}\}, N')$.

D. Modification of BFS symmetry breaking constraints [16] for modular controller synthesis

As in Section IV, we also apply the symmetry breaking constraints that enforce the states of the automaton to be enumerated in the ordered defined by the BFS algorithm [16]. However, direct application of these constraints leads to an unsatisfiability result in the case when we try to find a modular controller with a total number of states that is less than $M \times N$. The reason is that one of the constraints from [16] demands that each state has a defined parent in the BFS tree (BFS parent) with a smaller number:

$$\bigwedge_{1 \leq n < N} p_{n,1} \vee p_{n,2} \vee \ldots \vee p_{n,n-1},$$

where p_{n_1,n_2} denotes that n_2 is the BFS parent of n_1.

If a state is not used, then it, obviously, does not have a BFS parent. Therefore, we rewrite the above constraint from [16] by adding an additional index for the module and a condition that the state must be used in order to have a BFS parent:

$$\bigwedge_{m}\bigwedge_{1 \leq n < N} uc_{m,n} \implies p_{m,n,1} \vee p_{m,n,2} \vee \ldots \vee p_{m,n,n-1},$$

and also state that an unused state does not have a BFS parent:

$$\bigwedge_{m}\bigwedge_{1 \leq n_1 < N}\bigwedge_{0 \leq n_2 < N} \neg uc_{m,n_1} \implies \neg p_{m,n_1,n_2}.$$

The monolithic and modular controller synthesis algorithms are implemented as stand-alone Java tools. The generated controller is exported as a set of IEC 61499 XML function block files: one composite function block that determines the modular decomposition and M basic function blocks representing the modules of the controller. The generated controller can be directly used in NxtStudio.

VI. Case Study: Distribution Station

A. Physical system description

As a case study in this paper we use a Festo didactics' distribution station from our lab. It is comprised of two main units (Fig. 4): a Stack Magazine Module (*SMM*) and a Rotating Arm Changer (RAC) module. The station has six digital inputs and five digital outputs: *SMM* has three inputs and one output, whereas the *RAC* has three inputs and four outputs.

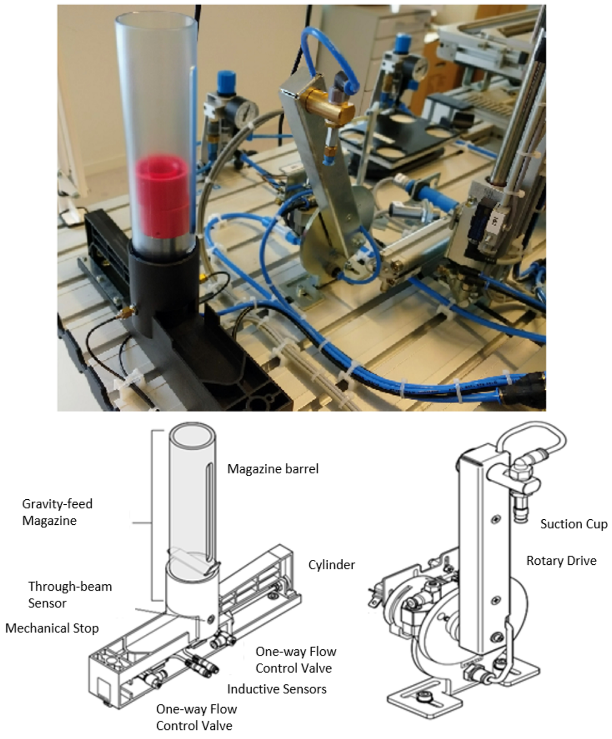

Fig. 4. The Distribution Station: physical implementation (top), Stack Magazine (left), and Rotating Arm (right)

The *SMM* consists of three parts: a gravity-feed magazine, a mechanical stop, and a single-acting cylinder (needs a single actuation to extend and retract). When the actuation is `True` it extends, and retracts when the actuation is `False`. The feed magazine holds a maximum of eight work pieces. Table I summarizes the I/O's and their actions for *SMM*.

TABLE I
SMM I/O's and their actions

Name	*Type*	*Description*
empty	input	`True` if there are no work pieces in the magazine, `False` otherwise
retracted	input	`True` if cylinder is retracted, `False` otherwise
extended	input	`True` if cylinder is extended[a], `False` otherwise
retract	output	`True` when cylinder needs to be retracted, `False` otherwise

[a]The default/reset position of the cylinder is extended.

The *RAC* is the transport system that moves the work pieces from magazine physical stop position to another side (downstream) using the suction cup. The arm behaves as a double-acting cylinder, instead of extending and retracting, it swivels from the magazine (SMM) side (*to_magazine*) downstream (*to_client*). The arm has two stop positions, *at_magazine* and *at_client*. Table II summarizes the I/O's and their actions for *RAC*. A simulation model of the system developed in NxtStudio was used to test the generated controller models.

TABLE II
RAC I/O'S AND THEIR ACTIONS

Name	***Type***	***Description***
at_magazine	input	`True` when arm is at the magazine side, `False` otherwise
at_client	input	`True` when arm is at the client(downstream) side, `False` otherwise
vac_on	input	`True` if vacuum (suction), `False` otherwise
vacuum_on	output	`True` if vacuum needs to be turned on, `False` otherwise
vacuum_off	output	`True` if vacuum needs to be turned off, `False` otherwise
to_magazine	output	`True` if arms needs to be moved towards magazine side, `False` otherwise
to_client	output	`True` if arms needs to be moved towards client (downstream) side[a], `False` otherwise

[a]The default/reset position of the arm is *at_client* position.

B. Data collection

A total of six logs were generated from the physical system for six different use cases. The use cases consisted of varying complexity and varying length of runs (from use case transferring just one work piece to use cases collecting up to eight work pieces). Use cases differed in intermediate pushes of buttons on the machine control panel. Apart from I/O's of the legacy controller for SMM and RAC, human-machine interface (HMI) signals are also logged: input signals `start_but`, `stop_but`, `aut_man`, `emerg_but` indicating clicks of buttons, and output signals `reset_led` and `start_led` indicating turning on lamps of corresponding buttons. This way, the generated state machine will also incorporate logics of the HMI. The source code of the legacy PLC is written using IEC 61131-3 languages.

C. Monolithic state machine synthesis from noisy PLC traces

We used the method proposed in Section IV to find state machine models for the collected PLC traces. The algorithm found no solution for the number of errors $\varepsilon = 0$, and found one solution for $\varepsilon = 1$. The generated state machine was checked using simulation testing in NxtStudio.

In contrast, the preliminary algorithm published in conference proceedings [7] found 63 state machines with different numbers of errors, since no constraints on ε were used at that time. Out of these 63 solutions, only one demonstrated fully correct behavior during simulation. Thus, use of additional constraints (3) allowed to reduce the amount of manual work required for checking generated solutions. The constructed state machine with 8 states and 10 transitions (not counting the default `START` and `INIT` states and two corresponding transitions) is shown in Fig. 5 in the form of a basic function block ECC. Due to long guard conditions, we display each of them with a binary string in which the i-th character corresponds to the i-th input variable.

D. Modular controller synthesis

We then used the modular controller synthesis method to generate a two-module controller where the first module operates HMI signals (`start_led`, `reset_led` and `mag_empty`), and the second module operates all other output signals. The architecture of the generated modular controller is shown in Fig. 6. The first (HMI) module operates three output variables using three input variables, has four states and three transitions. The second module operates five output variables using seven input variables, has eight states and nine transitions.

VII. Discussion & threats to validity

We have shown that it is possible to identify a state machine controller from noisy traces collected from the legacy PLC that is correct in the sense that it copies the behavior of the PLC controller for situations represented by the traces. However, the question of ensuring that the generated state machine is indeed in some sense equivalent to the original controller still stands: in our black-box case it is impossible to formally guarantee that all possible input-output combinations are featured by collected behavior traces.

As an example, consider Fig. 5: one can notice that the first input variable (`start_but`) value in all guard conditions is "0", meaning that there is no transition for which `start_but` = `True`. Consider the following consecutive elements of one of the preprocessed behavior traces.

1) $\langle \texttt{REQ}[000000100100], \texttt{CNF}[10000010]\rangle$
2) $\langle \texttt{REQ}[100000100100], [10000010]\rangle$
3) $\langle \texttt{REQ}[000000100100], \texttt{CNF}[00000001]\rangle$

In the second element the first input variable is `True` (1), it is the `start_but` variable. We can see that the output variable values in elements 1 and 2 are the same, so our algorithm treats element 2 as a passive one. In element 3 `start_but` is already `False`, and the output variables have changed (the RAC started moving). This is due to the behavior of the original PLC code: the `start_but` variable remains `True` only while the user keeps the button pressed down. This time interval is very small, so when the plant reacts to the button press, the `start_but` variable is already back to `False`. So from one or even several traces it is unclear which inputs (e.g. from element 2 or 3) trigger the corresponding output. Nevertheless, the state machine works correctly in NxtStudio simulation: the ECC waits for a `REQ` event so it does not move the arm until the start button is pressed; though ideally it should wait for a `REQ` event with `start_but` = `True`).

Some possible ways of countering such issues may include: (1) use of integration tests from the production environment if any; (2) use of formal verification (e.g. model checking) to ensure compliance of the generated state machine controller with formal specification (if any); (3) use of a testing period: e.g. one week data is collected, then a state machine generated,

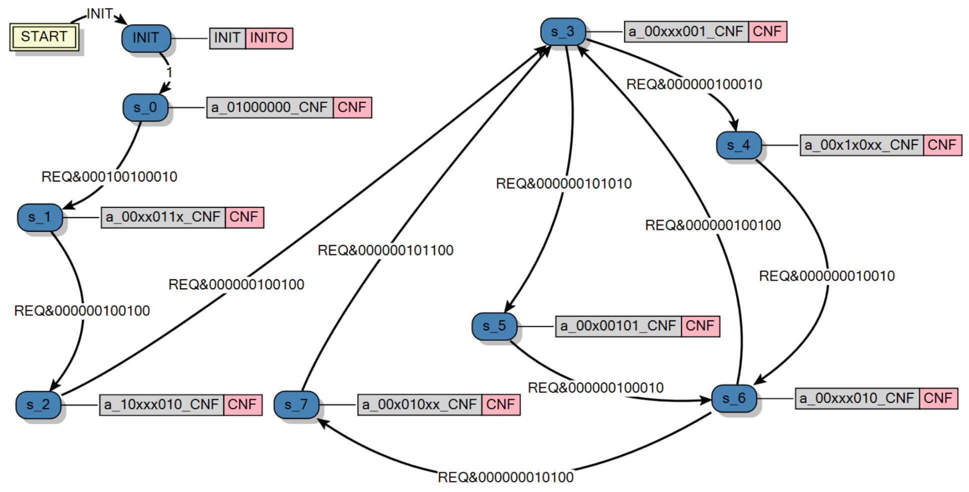

Fig. 5. ECC derived from the synthesized monolithic automaton

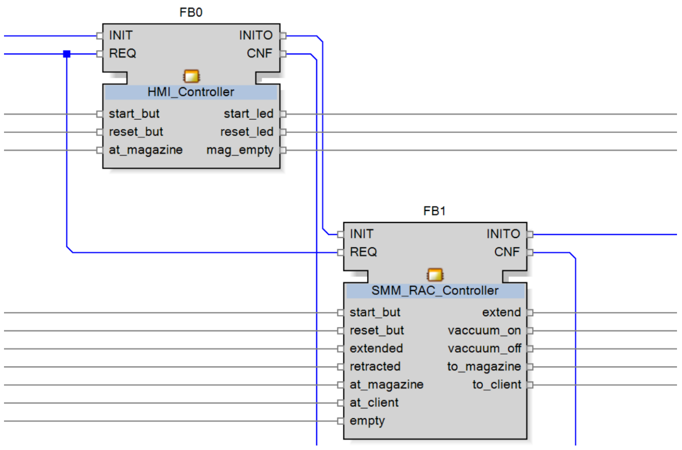

Fig. 6. Architecture of the generated modular controller

then new data is collected and checked for compliance with state machine. The third approach appears to be the most practically applicable one and is a variation of counterexample-guided inductive synthesis (CEGIS) [30], [31]. It would feature the following loop.

1) Collect initial behavior traces from the PLC.
2) Generate state machine(s), upload to new controller that works in monitoring mode; real work is still done with the legacy controller.
3) Monitor differences in legacy PLC and new controller behavior; if a difference is detected, form a new trace and go to step 2.
4) Deploy the generated state machines to the new controller and use it in production.

In this paper we only make several steps towards this goal by developing the state machine synthesis methods capable of dealing with noisy PLC traces and modular decomposition. Implementation of the CEGIS loop is part of future work.

The proposed modular controller synthesis algorithm may be used in various scenarios. First, the method can be used to find *some* modular decomposition, given only the behavior traces and the number of modules. This is useful when little is known about the synthesized system.

Second, the user may specify the modular decomposition represented by variables $\alpha_{m,i}$ and $\theta_{m,j}$, partially or completely. For example, the user may request variables z_1, z_3 and z_7 to be controlled by module 1, for which we will add constraints $\theta_{1,1}$, $\theta_{1,3}$ and $\theta_{1,7}$; other parts of the modular decomposition will be determined automatically. This is useful when the user has specific requests about the design of the modular system.

Two algorithms developed in this paper – (1) monolithic controller synthesis from noisy PLC traces and (2) modular controller synthesis from error-free traces – may be used either separately or in a single toolchain. In the latter case the monolithic algorithm is used to prepare error-free traces, while the modular algorithm allows distributing control logic across several function blocks. Such distribution should allow making synthesized controller more user-friendly and easily understandable to humans, especially if the modular decomposition was partially selected by the user and if all parameters of the synthesized controller have been minimized.

Note that the proposed modular controller synthesis method only allows to distribute sequential control logic (e.g. described by traces from one physical PLC) across several modules which can be attributed to several physical devices. The method is, however, inapplicable for data collected from several points in a distributed automation system. This is a more challenging problem and will be subject of future work.

VIII. Conclusion and Future Work

Factory floors always need upgrading to stay competitive due to new technologies being introduced. The proposed methodology is well-suited for upgrading legacy PLCs so that current advanced technologies can be applied in an existing production environment while also allowing vendors of other more traditional automation platforms to "test" the effectiveness of modern approaches without risking high investments from the beginning. This allows the industry to study the effects and make a seamless transition.

This paper only covers some initial steps towards this goal. Several directions of further development and improvement are envisaged. First, data could be logged into the cloud directly instead of files locally stored on the PLCs. The second possible improvement is to run the learning algorithm in the cloud too and constantly check with automated testing to see if the generated model behaves the same way as the legacy system. This process may continue until all test cases are satisfied, and no new test cases are generated. The third idea would be to collect data through distributed data collection points with time synchronization in order to extend the modular controller synthesis method proposed in this paper and to be able to generate distributed control software based on traces collected from a distributed system.

Another direction of future research is increasing readability of automatically synthesized state machines. Partly the readability is increased with our modular controller synthesis method, which can decompose control logic over several independent function blocks. Readability of each individual ECC is increased by minimization of the number of states and transitions, as well as guard conditions size. Future work may target the readability issue specifically, e.g., by inferring invariants for each state of each ECC and deriving state annotations automatically.

Acknowledgment

We thank Vladimir Ulyantsev for fruitful discussions, and the anonymous reviewers for their comments that helped us improve the paper. This work was supported by the Government of Russian Federation (Grant 08-08) and by European project DAEDALUS (H2020 Grant Agreement n°: 723248), and also by RFBR, project number 19-07-01195 A.

References

[1] Y. Shimanuki, "OLE for process control (OPC) for new industrial automation systems," in *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 6, 1999, pp. 1048–1050.

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

[3] V. Vyatkin, *IEC 61499 function blocks for embedded and distributed control systems design*, 3rd ed. ISA-Instrumentation, Systems, and Automation Society, 2015.

[4] "Programmable Logic Controllers - Part 3: Programming Languages, IEC Standard 61131-3," 2013.

[5] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Trans. Ind. Inform.*, vol. 7, no. 4, pp. 768–781, 2011.

[6] ——, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. Ind. Inform.*, vol. 9, no. 3, pp. 1234–1249, 2013.

[7] D. Chivilikhin, S. Patil, A. Cordonnier, and V. Vyatkin, "Towards automatic state machine reconstruction from legacy plc using data collection," in *17th IEEE International Conference on Industrial Informatics*, 2019, pp. 147–151.

[8] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[9] J. Marques-Silva and K. A. Sakallah, "GRASP—a New Search Algorithm for Satisfiability," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '96. USA: IEEE Computer Society, 1997, p. 220–227.

[10] M. J. H. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Grammatical Inference: Theoretical Results and Applications*, 2010, pp. 66–79.

[11] ——, "Software model synthesis using satisfiability solvers," *Empirical Softw. Eng.*, vol. 18, no. 4, pp. 825–856, 2013.

[12] I. Zakirzyanov, A. Morgado, A. Ignatiev, V. Ulyantsev, and J. Marques-Silva, "Efficient symmetry breaking for SAT-based minimum DFA inference," in *LATA*, 2019, pp. 159–173.

[13] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 811–853, 2016.

[14] G. Giantamidis and S. Tripakis, "Learning moore machines from input-output traces," in *Formal Methods*, 2016, pp. 291–309.

[15] S. M. Lucas and T. J. Reynolds, "Learning deterministic finite automata with a smart state labeling evolutionary algorithm," *IEEE Trans. Pat. Anal. Mach. Int.*, vol. 27, no. 7, pp. 1063–1074, 2005.

[16] V. Ulyantsev, I. Zakirzyanov, and A. Shalyto, "BFS-based symmetry breaking predicates for DFA identification," in *Language and Automata Theory and Applications*. Springer International Publishing, 2015, vol. 8977, pp. 611–622.

[17] I. Buzhinsky and V. Vyatkin, "Modular plant model synthesis from behavior traces and temporal properties," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation*, 2017, pp. 1–7.

[18] J. S. Mitchell, *An introduction to machinery analysis and monitoring*. Pennwell Publishing, 1981.

[19] A. K. Jardine, D. Lin, and D. Banjevic, "A review on machinery diagnostics and prognostics implementing condition-based maintenance," *Mechanical systems and signal processing*, vol. 20, no. 7, pp. 1483–1510, 2006.

[20] A. G. Rehorn, J. Jiang, and P. E. Orban, "State-of-the-art methods and results in tool condition monitoring: a review," *Intern. J. Adv. Manufact. Techn.*, vol. 26, no. 7-8, pp. 693–710, 2005.

[21] L. Wang, "Machine availability monitoring and machining process planning towards cloud manufacturing," *CIRP Journal of Manufacturing Science and Technology*, vol. 6, no. 4, pp. 263–273, 2013.

[22] V. C. Gungor and G. P. Hancke, "Industrial wireless sensor networks: Challenges, design principles, and technical approaches," *IEEE Trans. Ind. Electr.*, vol. 56, no. 10, pp. 4258–4265, 2009.

[23] L. Zhuang, K. M. Goh, and J.-B. Zhang, "The wireless sensor networks for factory automation: Issues and challenges," in *IEEE Conference on Emerging Technologies and Factory Automation*, 2007, pp. 141–148.

[24] D. Chivilikhin, V. Ulyantsev, A. Shalyto, and V. Vyatkin, "Function block finite-state model identification using SAT and CSP solvers," *IEEE Trans. Ind. Inform.*, 2019.

[25] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *Int. J. Softw. Tools Technol. Transfer*, vol. 20, no. 1, pp. 35–55, 2018.

[26] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Trans. Ind. Inform.*, vol. 13, no. 4, pp. 1521–1530, 2017.

[27] F. Avellaneda and A. Petrenko, "FSM inference from long traces," in *Formal Methods*. Springer, 2018, pp. 93–109.

[28] I. Zakirzyanov, A. Shalyto, and V. Ulyantsev, "Finding all minimum-size DFA consistent with given examples: SAT-based approach," in *Software Engineering and Formal Methods*, 2018, pp. 117–131.

[29] O. Bailleux and Y. Boufkhad, "Efficient CNF encoding of boolean cardinality constraints," in *Principles and Practice of Constraint Programming*, F. Rossi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 108–122.

[30] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, 2006, p. 404–415.

[31] A. Solar-Lezama, "Program synthesis by sketching," 2008, PhD thesis.

Daniil Chivilikhin received the Bachelor's and Master's degrees in applied mathematics and informatics from ITMO University, Saint Petersburg, Russia, in 2011 and 2013, respectively; and the Ph.D. degree in technical sciences (mathematics and software for computing systems) from the same university, in 2015.

He is currently a Research Associate with the Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia. His research interests include program synthesis and verification, industrial informatics, evolutionary algorithms, and SAT solver applications.

Sandeep Patil (S'11–M'19) received the Bachelor's degree in computer science engineering from the CMR Institute of Technology, Bangalore, India, in 2005; the Master of computer science (software engineering) degree from the Illinois Institute of Technology, Chicago, IL, USA, in 2010; the Master of Engineering Studies (computer systems) degree from the University of Auckland, Auckland, New Zealand, in 2011; and Ph.D. degree in formal verification of cyber physical systems from the Lulea University of Technology, Lulea, Sweden in 2018.

His research interests include software engineering principles and methodologies in distributed industrial automation, especially using the IEC 61499 paradigm. He also works with formal verification techniques in the same field. He is an accomplished software engineering professional with over 14 years of research and development experience in systems and application software, including four years at Motorola India Pvt. Ltd., India, as a Senior Software Engineer.

Anthony Cordonnier received the Baccalaureate (equivalent of an A-level or high school diploma) in sciences and technologies of industry and of sustainable development option energies and environment from the Louis Majorelle High School, Toul, France, in 2014; the "Brevet de Technicien Supérieur en Electrotechnique" (equivalent of a Higher National Diploma in Electrotechnics) from the Pôle des Industries de Lorraine, Maxeville, France, in 2016; and the Master's degree in electrical engineering from Institut National des Sciences Appliquées (INSA), Strasbourg, France, in 2019.

He has completed his last five years of studies in sandwich course: he worked as a maintenance technician apprentice between 2014 and 2016, and market researcher engineer apprentice between 2016-2019 at RTE company. He is currently working at ENEDIS company as an in-company trainer and course developer in the field of electrical network management.

Konstantin Chukharev received the Bachelor's degree in control systems and informatics from ITMO University, Saint Petersburg, Russia, in 2018. Additionally, he finished the one-year program "Algorithmic Bioinformatics" at Bioinformatics Institute, Saint Petersburg, Russia, in 2017.

He is currently is a Junior Research Associate with the Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia. He studies formal methods, finite-state automata, SAT and phylogenetics while teaching students and writing his Master's thesis in applied mathematics and informatics devoted to modular automata synthesis.

Valeriy Vyatkin (M'03–SM'04) received Ph.D. in 1992 and Dr. Sc. in 1999 in applied computer science from Taganrog Radio Engineering Institute, Russia, in 1992, Dr. Eng. degree from Nagoya Institute of Technology, Japan in 1999, and Habilitation degree in Germany in 2002. He is on joint appointment as Chair of Dependable Computations and Communications, Luleå University of Technology, Sweden, and Professor of Information Technology in Automation, Aalto University, Finland. He is also co-director of the international research laboratory "Computer Technologies", ITMO University, Saint Petersburg, Russia.

Previously, he was a visiting scholar at Cambridge University, UK, and had permanent appointments with the University of Auckland, New Zealand; Martin Luther University, Germany, as well as in Japan and Russia.

His research interests include dependable distributed automation and industrial informatics; software engineering for industrial automation systems; artificial intelligence, distributed architectures and multi-agent systems in various industries: smart grid, material handling, building management systems, datacentres and reconfigurable manufacturing.

Dr. Vyatkin received the Andrew P. Sage award for the best IEEE Transactions paper in 2012. He is the Chair of IEEE IES Technical Committee on Industrial Informatics.