Buzhinsky, Igor; Vyatkin, Valeriy

Automatic Inference of Finite-State Plant Models From Traces and Temporal Properties

Published in:
IEEE Transactions on Industrial Informatics

DOI:
10.1109/TII.2017.2670146

Published: 01/08/2017

Document Version
Peer reviewed version

Please cite the original version:
Buzhinsky, I., & Vyatkin, V. (2017). Automatic Inference of Finite-State Plant Models From Traces and Temporal Properties. *IEEE Transactions on Industrial Informatics*, *13*(4), 1521-1530. [7857798].
https://doi.org/10.1109/TII.2017.2670146

Automatic Inference of Finite-State Plant Models from Traces and Temporal Properties

Igor Buzhinsky and Valeriy Vyatkin, *Senior Member, IEEE*

***Abstract*—Closed-loop model checking, a formal verification technique for industrial automation systems, increases the richness of specifications to be checked and reduces the state space to be verified compared to the open-loop case. To be applied, it needs the controller and the plant formal models to be coupled. There are approaches for controller synthesis, but little has been done regarding plant model construction. While manual plant modeling is time consuming and error-prone, discretizing a simulation model of the plant leads to state excess. This paper aims to solve the problem of automatic plant model construction from existing specification, which is represented in the form of plant behavior examples, or traces, and temporal properties. The proposed method, which is based on the translation of the problem to the Boolean satisfiability problem, is evaluated and shown to be applicable on several case study plant model synthesis tasks and on randomly generated problem instances.**

***Index Terms*—Model checking, closed-loop modeling, industrial automation software, automatic model synthesis, SAT.**

I. Introduction

INDUSTRIAL automation systems are often safety and/or mission critical, so their correctness must be ensured formally. Formal verification and specifically model checking [1] is one of the ways to address this problem. While model checking allows proving properties of the system which cannot be derived from software testing, it is often a time consuming process, in which the entire state space of the automation system is explored. The complexity issue is mitigated, but only partially, by modern symbolic model checkers.

Closed-loop model checking of control systems [2]–[4] enables proving not only controller but plant properties which are usually of more interest in the automation systems context. For that, the engineer is required to supplement the controller model with the corresponding plant model, which represents physical and mechatronic aspects of the industrial automation system. The inclusion of this model into the system not only allows specifying properties to be verified in terms of the plant, which is natural in control engineering, but also often reduces the state space of the entire system compared to the open-loop software model. This is achieved by omitting states unreachable in closed loop.

I. Buzhinsky is with Department of Electrical Engineering and Automation, Aalto University, Espoo 02150, Finland, and also with Computer Technology Department, ITMO University, St. Petersburg 197101, Russia (email: igor.buzhinskii@aalto.fi).

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

While manual work can be minimized during controller model preparation [5]–[7] even when its implementation is not entirely available, plant model development is an extra effort, which is not automated. Still, there is usually much information available to synthesize this model, such as plant interface, system specification and behavior examples. It can also be possible that the plant model is available, but it is continuous and comprehensive (such models, for example, can be created in the Apros software), which impedes its direct use in formal verification. This particular case leads to the problem of intelligent model simplification and discretization, which can be solved based on the data obtained from the comprehensive model (e.g. simulations, plant structure).

This paper presents a method to synthesize discrete plant models using behavior examples and temporal properties represented in linear temporal logic (LTL). Models are constructed as nondeterministic Moore automata and can further be translated into more convenient forms suitable for existing verifiers. As the means for plant model construction, the translation-to-SAT approach [8] is applied. This approach is appealing due to the recent increase of Boolean satisfiability problem (SAT) solver performance.

This work is the extended version of the conference paper [9]. Unlike the conference version, this work presents the actual translation of the plant model synthesis problem to the SAT problem, evaluates the proposed method more thoroughly, and compares it with other ones.

The paper is structured as follows. Section II reviews related work. Next, in Section III, the proposed plant model inference method is described. It is evaluated in Section IV on a number of plant construction problems. In Section V, the method is compared with other approaches. The results are discussed and concluded in Section VI.

II. Related Research

In this section, two topics are reviewed: the application of plant models in software engineering for industrial automation, and previous approaches to deterministic system inference.

A. Plant Models in Industrial Automation

Testing and verification are the main means to ensure the correctness of industrial automation software. Both can be done either in *open loop* or in *closed loop* [4]. The more straightforward open-loop approach requires only the control software to be available. However, since requirements are normally stated in terms of the plant, test cases are hard to

generate in this setting, and open-loop verification requires to check all possible input combinations, which is often infeasible.

In contrast, the plant model, which is mandatory in closed-loop approaches, allows to check the control software in an environment which resembles the devices with which the controller is supposed to operate in production. Modern software tools, such as Simulink, Modelica, Apros and Ptolemy II, allow modeling of complex physical objects and processes. The simplest way to use the plant model is to perform simulations, where the controller is faced with inputs possible for the real plant. However, to be used in formal verification, the plant model needs to be formal and often discrete. Finite-state machines, Petri nets [10], net condition/event systems (NCES) [11] and timed automata are among such models. Model checking in addition requires the model to have a reasonable size to be applicable in practice.

Closed-loop and open-loop model checking are compared in [12] on a case study. In closed-loop model checking, properties are only checked for model behaviors which are valid from the point of view of the plant. Thus, if plant models are generated from traces, this approach also has a limitation: some of the valid behaviors may be missed. On the other hand, the paper [12] discusses that these approaches are suitable for checking different sorts of properties. First, if a property involves plant variables which are not observable by the controller, then checking it in open loop is impossible since the required variables are missing. Second, even if these plant variables are observable, without the plant model their values are assumed to be arbitrary. For example, if such a variable refers to the water level in a tank which must be maintained around a setpoint, then this requirement would be simply false in the open-loop case. Consequently, the presence of the plant model gives more freedom in model checking.

In [4], a methodology for semi-automatic modular plant model generation is proposed. It is suggested to start from a 3D CAD drawing of the plant, then transform it into a simulation model, and finally to transform the simulation model into a formal one. Compared to [4], the approach from the present paper is not capable of constructing simulation models, but it is able to convert simulation models into formal ones in a more intelligent way. One more example of a modular approach to plant model construction is the work [13].

Another work [14] is devoted to constructing plant simulation models using information contained in PLC programs. The authors rely only on the source code of the control application. While this is convenient (there is no need to prepare additional specification), such an approach can result only in very basic plant models, where each process in the plant is modeled by a state machine derived from the symbol table of the PLC program. This approach has some applicability in verification by simulation, but models generated this way lack dependencies between plant subcomponents.

In [15], instead of plant model identification, formal models of entire closed-loop systems are constructed using the data obtained from a running real-world controlled plant. Similarly to the present work, nondeterministic finite-state automata are applied in [15], but since the closed-loop system has no inputs, these automata are interpreted as generators of possible system input-output behaviors.

B. Approaches to Deterministic Automaton Synthesis

A large volume of publications deals with the issue of constructing software and automation system models in the form of deterministic finite automata (DFA), finite-state machines (FSMs), extended finite-state machines (EFSMs) and timed automata. Some of these approaches might serve as a basis for the method to solve the problem considered in this paper.

One of the approaches [16] identifies DFA from sets of positive and negative behavior examples. Such automata are only able to distinguish incorrect software behaviors from correct ones. Richer models of EFSMs are considered in [17]. Both approaches, [16] and [17], translate the problem of model synthesis to an instance of the SAT problem: the solution of the problem is represented as a set of Boolean variables whose values are constrained to make them represent an automaton which complies with the given specification. To solve these instances, third-party SAT solvers are applied, which have recently become very efficient. The usual workflow of such methods looks as follows:

1) represent the sought solution of the problem using Boolean variables;
2) generate the Boolean formula which constrains the variables to make them describe only correct solutions;
3) run the SAT solver;
4) if the solver has found a satisfying assignment to the formula, reconstruct the solution from this assignment;
5) otherwise, if the solver has spotted that the formula is unsatisfiable, no solution exists.

Another group of approaches is based on state merging [16], wherein traces are represented as a tree, and the nodes of this tree are then consequently merged guided by statistical data. In [18], modular plant models are constructed in the form of timed probabilistic automata based on behavior examples consisting of events and their timestamps, and then used for anomaly detection in a running automation plant. Another method [19] is able to construct software models from traces and LTL properties. It is based on iterative counterexample prohibition and repeated state merging execution. Finally, the method of deriving EFSMs from traces with real-valued data [20] is based on state merging and machine learning.

The next group of approaches exploits metaheuristic algorithms, which treat synthesis as an optimization problem. In the works [21] and [7], EFSMs are constructed from traces and temporal properties. A common shortcoming of such approaches is the large time required to find solutions.

Temporal formulae as the only type of specification are considered in the problem of LTL synthesis, where a piece of software needs to be synthesized. Approaches to solve this problem, such as the ones described in [6], [22], are less interesting in the context of this paper. First, they do not allow behavior examples to be used as input specification. Second, their nature is more comprehensive than the one of the approaches based on the SAT problem, state merging and metaheuristics, which makes removing the determinism restriction typical of LTL synthesis much harder.

III. Proposed Method

This section states the solved problem formally and outlines the proposed plant model synthesis method. The source code of the method, which is implemented in Java, is available online[1] as a module inside the EFSM construction project [17].

A. Plant Model Representation

The plant model to be constructed will be represented using the formalism of finite automata, or finite-state machines, which is widely applied in formal verification and software engineering in general. Moore-type automata, which incorporate outputs in their states, are most suitable since this mapping of states into outputs clearly represents the usual situation when the plant's state is partially visible to the controller via sensors. Next, plant models can be thought to be nondeterministic: this accounts for possible human intervention into the plant operation as well as helps to tackle value precision issues, for example, by rounding a real value to both neighboring integers nondeterministically.

Other options for a formalism to represent discrete plant models include Petri nets, NCES and timed automata. While the first two options are more suitable for representing distributed plants, the third option captures continuous time aspects (timed automata are defined in terms of continuous time, but it can still be processed as a finite set of intervals). Finite automata are chosen instead since they are more basic, widespread, and can be converted into other formalisms without loss of information.

Formally, a nondeterministic Moore automaton is a sextuple $(S, S_0, I, O, T, \lambda)$, where S is the finite set of *states*, $S_0 \subset S$ is the non-empty set of *initial states*, I and O are the finite sets of *inputs* and *outputs* respectively, $T \subset S \times I \times S$ is the *transition relation*, and $\lambda : S \to 2^O$ is the *output function*. Inputs are regarded in a generic sense: they can embody any discrete information from the controller, including events and variable values. As for outputs, the chosen form of the output function suggests that they should mainly be regarded as Boolean variables, which are nevertheless sufficient to represent any discrete value. Each state is labeled with a set of outputs from O, which means that each output in each state can be present or not independently of others. The automaton must also satisfy the *completeness* requirement: for each input and state there is at least one transition to some other state.

The interaction of the plant and the control software (controller) is assumed to be cycle-based, but time intervals between cycles are irrelevant and are not obliged to be constant. In the beginning, the plant nondeterministically selects its initial state from the set S_0. Then, on each cycle, the controller reads the plant's output and chooses an input, after which the plant nondeterministically chooses a transition matching its current state and the provided input, and changes its state.

Fig. 1 shows an example of an automaton which represents the model of a pneumatic cylinder (Fig. 2) with three inputs (`extend`, `retract`, `wait`), two outputs (`home`, `end`) and three positions (home, end and intermediate). Note the set notation

[1] https://github.com/ulyantsev/EFSM-tools/

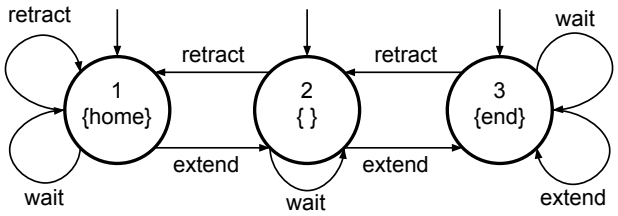

Fig. 1. Automaton representing the model of the pneumatic cylinder

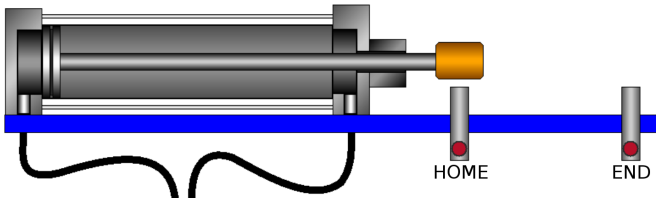

Fig. 2. Pneumatic cylinder. Home and end sensors identify the position of the piston rod

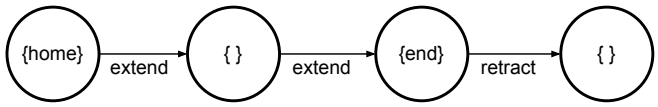

Fig. 3. Trace example for the pneumatic cylinder

for outputs. All the states of this model are initial, which is indicated by incoming arrows without source states. No other sources of nondeterminism are present in this model.

B. Specification Representation and Problem Statement

The first considered type of specification is a finite set of *traces*, or behavior examples. A trace of length l is a directed path $v_1 e_1 ... v_l e_l v_{l+1}$, where each node is represented by plant model outputs ($v_j \in 2^O, 1 \le j \le l+1$), and each edge is represented by a plant model input ($e_j \in I, 1 \le j \le l$). Traces can be *positive* (sometimes referred to as just traces) or *negative*. Only positive traces are used as input specification, and negative traces are employed as a means of supporting temporal properties. The synthesized automaton is required to have each positive trace in its transition graph starting from an initial state. In contrast, negative traces must not correspond to any paths in the transition graph. An example of a positive trace for the cylinder example is shown in Fig. 3.

The remaining type of specification is a finite set of *LTL formulae*, or *LTL properties*, which the synthesized model is obliged to satisfy. LTL properties contain usual Boolean operators ($\wedge$, $\vee$, $\neg$, $\rightarrow$), temporal operators ($\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$, $\mathbf{U}$), and atomic propositions:

- `input`$(i), i \in I$: input i was received by the plant on the current cycle of the plant-controller interaction;
- `output`$(o), o \in O$: output o was produced by the plant on the current cycle of the plant-controller interaction.

Since the plant-controller interaction is initiated by the plant, which selects an initial state with the first produced subset of outputs, on the first step neither of the `input` propositions are

satisfied. Below are some examples of LTL properties for the cylinder model synthesis problem:

- $\mathbf{G}\,\neg(\texttt{output}(\texttt{home}) \wedge \texttt{output}(\texttt{end}))$: "the cylinder cannot report being in home and end positions simultaneously";
- $\mathbf{G}(\mathbf{G}(\texttt{input}(\texttt{extend})) \rightarrow \mathbf{F}\,\mathbf{G}(\texttt{output}(\texttt{end})))$: "if the cylinder receives the extension request eternally, then in the future it will start reporting being in the end position eternally".

Finally, the problem solved in this paper is to construct a nondeterministic Moore automaton, such that:

- the number of its states $|S|$ is fixed and equal to the supplied number;
- each supplied positive trace is a valid example of its behavior;
- the automaton complies with all given LTL properties;
- the automaton satisfies the completeness requirement.

C. Trivial Solution for the Case of Positive Traces Only

The assumption that the specification is represented only by positive traces, and the fact that the synthesized automata are nondeterministic leads to the trivial solution of the considered problem. To find the minimum automaton consistent with the traces, one needs to merge all trace nodes with identical outputs. When nodes are merged, their incoming and outgoing transition sets are simply united (no nondeterminism resolution is performed), and the resulting state is initial if and only if there was a root among the merged trace nodes. To satisfy the completeness requirement, missing transitions are added as self-loops. The described procedure is linear of the total length of traces. Note that, despite that states are merged, it should not be confused with state merging [16]. In the described procedure, no recursive merging is done and no statistical approaches are employed. This simplicity is explained by the absence of the need to make the resulting model deterministic.

D. General Scheme of the Method

From now on, assume that the LTL specification is nonempty. The plant model generation method was decided to be based on one of the existing methods for software model generation, which were briefly reviewed in Section II-B. While software models are often constructed in the form of deterministic Mealy automata, the desired approach must be capable of constructing nondeterministic Moore machines. Two kinds of existing methods almost do not depend on these model properties: the ones based on the SAT problem and on metaheuristic algorithms. Among them, a SAT-based approach was chosen to be developed, using the work [17] as the basis, since such an approach would be able to not only find the solution, but also prove that one does not exist. This allows applying such methods to find minimum models consistent with the given specification, which may decrease the size of the state space during model checking. The authors have also found that state merging can be adapted for nondeterministic state machine construction, but such an adaptation does not perform well according to the criteria listed in the end of Section III-B. More details about the problem solution based on state merging will be presented in Section V-A.

The workflow of the proposed method is slightly more complex than the one of a typical SAT-based approach (see Section II-B). Positive traces are encoded into the Boolean formulae directly, but LTL properties are not: instead, when the constructed model violates them, negative traces are added to the specification and encoded into new Boolean constraints. The idea of using counterexamples was adopted from the works [19], [23]. Instead of restarting the solver with the new, longer formula, the benefits of incremental SAT solving are exploited. In incremental SAT solving, a solver can handle multiple SAT instances in a single run, and each new instance to be solved is obtained by appending new constraints (possibly using more variables) to the previous Boolean formula. Thus, on each iteration, only new constraints are passed to the solver, and it is requested to produce a new solution. The incremental process ends when the identified model is finally correct. Briefly speaking, the stages of the proposed method are as follows:

1) translate the model inference problem instance, excluding LTL property compliance, into a Boolean formula;
2) run the SAT solver in the incremental mode;
3) if the solver spotted the absence of the solution, then either the specification is unrealizable or the supplied number of states is insufficient;
4) otherwise, reconstruct the solution from the found Boolean assignment;
5) if the solution violates LTL properties, generate counterexamples as negative traces, encode them into new constraints, pass them to the solver, request it to find a new solution and return to stage 3;
6) otherwise, apply post-processing of the obtained automaton.

Stages 1, 4, 5 and 6 are clarified in more detail in subsequent subsections. As a SAT solver, *cryptominisat*[2] was chosen. This solver is the winner of the SAT-Race 2015[3] incremental track.

E. Translation to SAT: the Case of Positive Traces Only

Assume that all positive traces form a directed graph $G = (V_{\text{tr}}, E_{\text{tr}})$ which includes each trace as a separate connected component. The following primary Boolean variables are used in the translation:

- $x_{v,s}$: whether node $v \in V_{\text{tr}}$ of the trace graph is passed by state s $(1 \leq s \leq |S|)$ of the automaton;
- $y_{s_1,s_2,i}$: whether there is a transition from state s_1 to s_2 $(1 \leq s_1, s_2 \leq |S|)$ triggered by input $i \in I$, i.e. whether $(s_1, i, s_2) \in T$.
- $z_{s,o}$: whether state s is labeled with output o, i.e. whether $o \in \lambda(s)$.

During the solution reconstruction stage of the method, y and z variables are sufficient to restore the automaton, except its initial states. For example, state 1 of the automaton from Fig. 1 corresponds to the following variables assigned to

[2]http://www.msoos.org/cryptominisat4/
[3]http://baldur.iti.kit.edu/sat-race-2015/

true: $y_{1,1,\texttt{retract}}, y_{1,1,\texttt{wait}}, y_{1,2,\texttt{extend}}, z_{1,\texttt{home}}$. Initial states of the automaton are encoded implicitly and are restored based on z variables and the original traces. For example, state 1 from Fig. 1 is marked initial based on the true value of $z_{1,\texttt{home}}$ and on at least one trace starting from `home` (for example, the one from Fig. 3).

Each node of the trace graph must be passed by a certain automaton state, which is required to be unique by the following at-least-one and at-most-one constraints:

$$\bigwedge_{v \in V_{\text{tr}}} \bigvee_{1 \le s \le |S|} x_{v,s};$$

$$\bigwedge_{v \in V_{\text{tr}}} \bigwedge_{1 \le s_1 < s_2 \le |S|} \neg \left(x_{v,s_1} \wedge x_{v,s_2} \right).$$

Next, each edge in the trace graph requires the presence of a proper transition in the automaton:

$$\bigwedge_{(v_1,i,v_2) \in E_{\text{tr}}} \bigwedge_{1 \le s_1, s_2 \le |S|} \left(x_{v_1,s_1} \wedge x_{v_2,s_2} \rightarrow y_{s_1,s_2,i} \right).$$

Assume that the output of the trace node v is given by $o(v)$. Each trace node also constrains the output of the corresponding state:

$$\bigwedge_{v \in V_{\text{tr}}} \bigwedge_{1 \le s \le |S|} \left(x_{v,s} \rightarrow E_{s,v} \right),$$

$$\text{where } E_{s,v} = \bigwedge_{o \in o(v)} z_{s,o} \wedge \bigwedge_{o \in O \setminus o(v)} \neg z_{s,o}.$$

The last requirement enforces the completeness of the automaton by specifying that for each input and each state there is at least one outgoing transition:

$$\bigwedge_{i \in I} \bigwedge_{1 \le s_1 \le |S|} \bigvee_{1 \le s_2 \le |S|} y_{s_1,s_2,i}.$$

F. Translation to SAT: Negative Traces

The information about negative traces emerging from counterexamples to unsatisfied LTL properties also needs to be encoded. In general, these counterexamples are infinite paths in the Kripke structure of the automaton [1], but they can nonetheless be translated into finite negative traces, since the situation is simplified by the following facts:

- It is sufficient to consider only so-called lasso-shaped counterexamples, which consist of a finite prefix and a loop.
- There is a subset of LTL properties called LTL *safety* properties: counterexamples to such properties always have a finite prefix such that every possible infinite continuation of this prefix is also a counterexample [1]. Such finite prefixes can be interpreted as finite counterexamples.
- In case of the absence of such a finite counterexample, the fact that the identified models are bounded in size becomes useful: a looping counterexample can be transformed into a finite one by repeating its loop $|S|$ times. This makes such counterexamples up to $|S|$ times more expensive to encode into a Boolean formula, though.

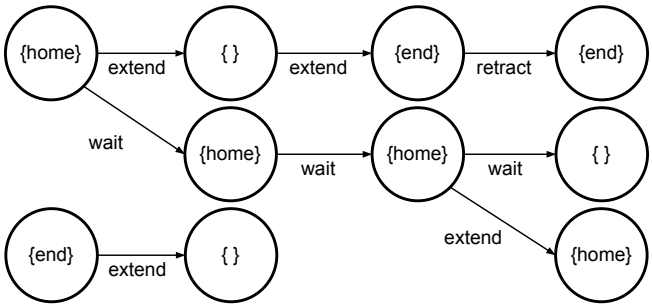

Fig. 4. Example of a negative trace graph for the cylinder example

Negative traces are treated somewhat analogously to the positive ones. Assume that they form a directed graph $\overline{G} = (\overline{V}_{\text{tr}}, \overline{E}_{\text{tr}})$. It is possible to use a separate connected component for each negative trace, but to reduce the number of nodes (and, consequently, the number of produced constraints), equal prefixes of negative traces are merged, which transforms the negative trace graph into a forest. Note that this simplification is impossible for positive traces, since equal prefixes of different positive traces can be passed by different states. An example of a negative trace graph for the cylinder example is shown in Fig. 4.

To specify states which can be used to pass negative nodes, variables $\overline{x}_{v,s}$ ($v \in \overline{V}_{\text{tr}}, 1 \le s \le |S|$) are employed. Each negative node can now be passed by possibly multiple states, but the terminal ones (the set of such nodes is denoted as $\overline{T}_{\text{tr}} \subset \overline{V}_{\text{tr}}$) are enforced to be impossible to pass:

$$\bigwedge_{v \in \overline{T}_{\text{tr}}} \bigwedge_{1 \le s \le |S|} \neg \overline{x}_{v,s}.$$

The next idea is, given an x, y and z assignment, to require negative nodes to be passed by all possible states. Denote the root nodes of the positive and negative trace graphs by R_{tr} and $\overline{R}_{\text{tr}}$, respectively. If a positive root is passed by a certain state, then so are all the negative roots labeled with the same output:

$$\bigwedge_{v_1 \in R_{\text{tr}}} \bigwedge_{v_2 \in \overline{R}_{\text{tr}}:\, o(v_2) = o(v_1)} \bigwedge_{1 \le s \le |S|} \left(x_{v_1,s} \rightarrow \overline{x}_{v_2,s} \right).$$

Finally, proper transitions and outputs allow the propagation of $\overline{x}$ values along the edges of the negative trace graph:

$$\bigwedge_{(v_1,i,v_2) \in \overline{E}_{\text{tr}}} \bigwedge_{1 \le s_1, s_2 \le |S|} \left(\overline{x}_{v_1,s_1} \wedge y_{s_1,s_2,e} \wedge E_{s_2,v_2} \rightarrow \overline{x}_{v_2,s_2} \right).$$

On each iteration of the method, a counterexample trace is added to the negative trace graph, and missing constraints are passed to the solver.

G. Further Improvements

The translation outlined above was additionally improved to provide better support for particular types of LTL properties. First, properties of the form $\mathbf{G}\, f$, where f is a Boolean formula over solely `output` atomic propositions, are included into the overall Boolean formula when it is assembled at the first time without any further counterexamples. This is achieved

by directly converting f into a set of Boolean constraints over $z_{s,o}$ variables for each s.

Second, properties of the form $\mathbf{G}\, f$, where f is an arbitrary temporal formula not matching the restriction stated in the previous paragraph, are handled with a special negative trace graph. The constraints generated from this graph are slightly altered to allow counterexamples to start in any state, not only in an initial one. This modification reduces the average size of counterexamples for such properties.

The final improvement concerns the way counterexamples are added on each iteration of the method. Only minimum counterexamples (in terms of the number of negative trace nodes required to represent them) are considered for each LTL property, and only the minimum negative trace among all unsatisfied LTL properties is added.

H. Post-Processing

On the post-processing stage, the generated model is converted into the convenient output format. The following output formats are currently supported:

- DOT language, which is suitable for graph visualization and is supported by the Graphviz software;
- NCES [11] modular graphical language, which is used in representing IEC 61131 and IEC 61499 compliant software;
- the input format of the popular symbolic verifier NuSMV.

All these conversions are straightforward. In particular, finite-state machines can be represented [10] with Petri nets, a subclass of NCES. The last two formats are applicable for combining the plant model and the control software model into a closed loop and further verification, which is the primary use case of the proposed method.

IV. Experimental Evaluation

In this section, the introduced technique is applied to construct several case study plant models and evaluated on randomly generated data. All experiments are performed on the Intel Core i7-4510U CPU with the clock rate of 2GHz.

A. Pneumatic Cylinder

This section starts with a very simple example – the discrete model of the pneumatic cylinder which has been previously shown in Fig. 2. Such cylinders are common, for instance, for pick-and-place robots like the one considered in [24]. The problem was to construct the model which would be able to report three positions – home, end and the intermediate one, and to switch positions based on controller's commands to extend or retract. To do this, three positive traces and six LTL properties were prepared. The examples of traces and LTL properties have already been given in Section III-B.

The resulting automaton is exactly the one which has been shown in Fig. 1. The method processed this example with the iteration number of three and the execution time of less than a second. Despite the successful result, the cylinder example is only suitable for the purposes of illustration and smoke testing of the method as the target model can be easily constructed by hand. More comprehensive plants are considered in the following subsections.

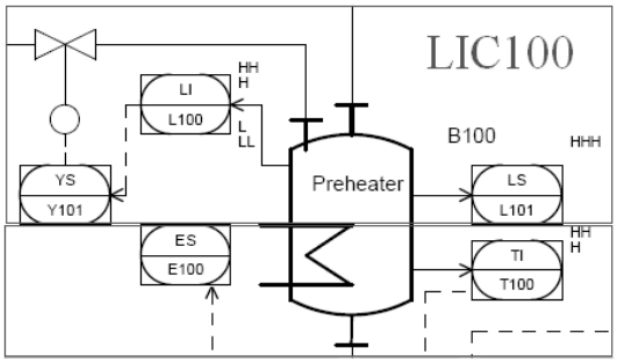

Fig. 5. Preheater tank and the LIC100 control loop (upper rectangle). L100 and L101 are the water level and overflow sensors respectively, Y101 is the valve actuating function.

B. Water Level

The next case example originates from a lab-scale plant called heat production plant (HPP) [25], which consists of a number of tanks, pipes, valves, pumps, water level and pressure sensors. The control application for this plant maintains proper water levels and pressures in the tanks and allows the HPP to supply pressurized heated water to a customer's process. This application is implemented as a number of subcontrollers for distinct control loops.

Specifically, the LIC100 control loop was chosen for the case study. It maintains the water level in the preheater tank. The part of the HPP's P&I diagram associated with this tank is shown in Fig. 5. The controller can access two level sensors, the continuous and the binary ones, and one binary valve which regulates the presence of the incoming water flow.

Previously, the NCES model of this control loop had been prepared. In particular, the plant model was obtained by discretizing the continuous water level with a given precision. Now the aim was to reconstruct water level models of this kind. Reference plant models with $|S| = 8k, k = 1..10$ states were generated. These numbers are multiples of 8, since this is the number of discrete water levels visible for the controller in the NCES model of the loop. These models included nondeterminism in the incoming and outgoing flows.

For each reference model, $4k$ positive traces were recorded. Each trace represents the situation of starting from the medium water level, reaching the extreme level (either the minimum or the maximum one), and then reaching the opposite extreme level. For each trace element, the concrete transition modeling a particular filling/emptying rate was chosen randomly. The described form of traces was chosen to make all plant model outputs covered: had random walks starting from normal water levels been used instead, critical levels would have been difficult to reach. Such traces ensure that problem instances are sufficiently complex, but do not guarantee transition coverage of the reference automata, which requires the synthesis method to restore missing transitions using LTL properties. The absence of coverage also mimics the situation where traces are obtained from plant simulations, which are not always capable of capturing the whole range of its behaviors. As for LTL properties, 66 of them were specified for each of the models.

TABLE I
RESULTS OF THE EVALUATION ON LIC100 INSTANCES

k	States	Traces	Iterations	Execution time (s)
2	16	8	75	3.9
4	32	16	99	13.7
6	48	24	118	34.6
8	64	32	128	78.8
10	80	40	223	339.4

Execution times and medium iteration numbers of the proposed method applied to the water level instances are presented in Table I (data only for even k is presented). As visible from the table, the method is able to synthesize models with up to 80 states in several minutes. For $k \geq 2$, the generated models were not isomorphic to the reference ones. However, almost all LTL properties in the specification were safety ones (i.e. always had finite counterexamples), which are easier for the method to process, as described in Section III-F. Later, in Section IV-D, the proposed method will be applied on instances with harder temporal formulae.

To show that the produced models fulfilled their purpose, they were transformed into NCES and combined with the LIC100 controller in a closed loop. Then, these composite models were subject to model checking against the requirements to have no overflows and underflows expressed in computation tree logic (CTL):

- $\mathbf{AG}\,\neg(\texttt{above\_h} \vee \texttt{above\_hh})$: "water level never reaches overflow alarm H or HH levels";
- $\mathbf{AG}\,\neg(\texttt{below\_l} \vee \texttt{below\_ll})$: "water level never reaches underflow alarm L or LL levels".

These requirements, which were satisfied for the reference models, also appeared to be fulfilled for the generated models.

C. Nuclear Power Plant Model

For the last case study, a model of a generic nuclear power plant (NPP) with a pressurized water reactor (PWR), hereafter referred to as the generic PWR model, was used. It was provided by Fortum Power and Heat Oy, a power utility with NPP operation license in Finland. This model, implemented in the Apros[4] modeling environment, includes the most important process, mechatronic and control components of an NPP. The subsystem responsible for activating emergency pumps was focused on, and a discrete plant model was synthesized for the related plant parameters: water level in the pressurizer (continuous output), two pressures (continuous output), the voltage supplied for the emergency pumps (continuous output) and the activation states of these pumps (a binary input for each of three pumps).

To prepare the data for plant model construction, 250 simulations were executed in Apros with the total length of 16 hours 40 minutes (measured in simulated time; in reality it took around 5 hours to record them). Inputs and outputs were measured each simulated second, resulting in the total trace length of 60 thousand. Due to the complexity of the system, even this amount of traces may not be sufficient to represent some of its behaviors, but for the considered amount the method already consumes almost 4GB of RAM. Each output parameter was discretized into several levels. The prepared problem instance had 3 inputs (for each combination of pump states found in the traces) and 12 outputs (several outputs for each parameter). As temporal properties, the validity of outputs (i.e. if a parameter has several discrete levels, then exactly one of them must be generated on each step) and the smoothness of their changes (a discrete level can only change to a contiguous level) were specified.

The plant model construction method was executed with various numbers of states, and the value of 12 was found to be the minimum possible one. For this number of states, the execution time of the method was 252 seconds and the number of iterations was 52. The constructed plant model was then coupled with the NuSMV model of control logic of the generic PWR model, which was obtained using the tool presented in [26]. Four CTL properties expressing activation of protection functions given certain inputs, as well as their nonactivation when such inputs never arrive, were successfully checked for the composite model using the NuSMV verifier.

D. Random Instances

The purpose of the evaluation on random instances was to test the method on a broader set of input specifications, including the ones which contain non-safety LTL properties, which are harder for the method to process (see Section III-F). For each $|S| = 5k, k = 2..7$, 50 random reference automata were generated, each with five inputs, five outputs (each state was labeled with exactly one output), one or two transitions for each source state and input. Each state of each generated automaton was checked to be reachable, and all these checks were successful. The chosen number of inputs is reasonable, since a larger number would reduce $|S|$ for which the method is able to terminate in reasonable time. A larger number of outputs would make states distinguishable based on them, which would simplify the learning task. The choice of transition density introduces nondeterminism but is sufficiently small to make reference automata easy to visualize.

Then, for each reference model, $|S|$ positive traces, each of length $|S|$, were recorded. These traces were generated by random walks and did not ensure any coverage: unlike Section IV-B, randomized generation does not create parts of the models which are hard to reach with such walks, and the reasons not to ensure coverage are the same as in Section IV-B. LTL properties, 50 for each automaton, were created with the help of the *randltl* tool [27]. They appeared to be mostly non-safety ones. All properties generated by *randltl* were ensured to comply with the reference automaton by repeating the generation if this requirement was not met. The large number of properties generated for each automaton is explained by the observation that many generated properties were satisfied by other randomly generated automata, and hence a means to make instances harder was required.

The results of applying the presented method to the data described above are summarized in Table II. The method was

[4]http://www.apros.fi/en/

TABLE II
RESULTS OF THE EVALUATION ON RANDOM INSTANCES

States	Time, s		Iterations		Solved within 1 hour
	Q_2	Q_3	Q_2	Q_3	
10	0.6	0.7	2	6	50
15	1.7	3.5	9	26	50
20	4.3	8.1	18	47	50
25	7.5	39.3	21	109	48
30	20.3	133.5	48	201	48
35	62.3	368.5	92	281	43

given one hour to process each instance, and the numbers of runs which did not violate the time limit are shown in the rightmost column. Other columns show the second and the third quartiles of the run time and the number of iterations. As visible from the table, the number of states which the method is able to handle in one hour is lower than the one presented in Table I. The reason for this is the presence of non-safety LTL properties, whose looping counterexamples require long Boolean formulae to be encoded.

Since the data in this experiment were generated artificially, the associated threats to validity should be discussed. First, the instances might have been too easy for the method, or they might not have exercised the support of LTL properties enough. However, according to Table II, there was a sufficient number of instances that were not solved instantly and which required many iterations to be solved. Second, automaton parameters except the number of states were fixed in this experiment. While this might have concealed more profound findings concerning the performance of the method, the most crucial Boolean formula parameters, namely the number of variables in it and its length, were variant for different $|S|$. Third, the generated instances might not be representative of model construction tasks for plants typical in industry. This threat is compensated by other experiments in Section IV.

V. COMPARISON WITH OTHER APPROACHES

This section compares the approach presented in this paper with other approaches, which were mentioned in Section II-B.

A. Qualitative Comparison

The techniques based on Boolean satisfiability [16], [17] are the closest ones to the approach presented in this paper. The EFSM synthesis technique [17] differs from the one presented in this paper by the following aspects: deterministic Mealy state machines are synthesised instead of nondeterministic Moore ones, and neither LTL properties nor the completeness requirement are supported. Considering this, and the fact that the translation from [17] was used as the basis for the translation in the present paper, the detailed comparison with this approach is not performed. The paper [16] presents a hybrid technique combining translation-to-SAT and heuristic approaches, including state merging. This technique synthesizes DFA and supports only traces as input data, so a direct comparison with this work is also impossible.

Among the automaton synthesis approaches based on state merging, the works [18]–[20] were mentioned. The general difference between state merging and translation-to-SAT approaches is that the first ones solve the synthesis problem inprecisely, while the second ones are exact but often more slow. The technique [18] constructs automation system models for the purpose of anomaly detection. The following aspects indicate that this approach is not applicable to the problem of plant model construction for closed-loop model checking:

- *Probabilistic deterministic timed automata* are synthesized in [18]. Probabilistic model checking mostly concerns quantitative properties (e.g. a fault does not happen with a certain probability), while qualitative ones (e.g. a fault does not happen) are considered in the present work.
- Only traces are supported as input data in [18]. Unlike the present paper, these traces consist of events annotated with timestamps.

In [19], automata construction is performed based on traces and LTL properties of the safety subtype (see Section III-F). Although the traces do not differentiate between inputs and outputs, and the constructed state machines are of the Mealy type, it was found that this approach can be applied for plant model construction. The empirical comparison of the proposed approach with [19] is provided in the next subsection.

The work [20], which combines state merging with supervised classifier learning, synthesizes deterministic software models as EFSMs. These models are inferred from traces represented as events labeled with real-valued data, and the support of these arguments is the essential part of the approach [20]. In the context of the present work, these real values can be imagined to represent plant parameters. This technique is evaluated on the problem of plant model construction in Section V-C.

Among solutions based on metaheuristics [21] and [7], the EFSM constuction approach [7] supports both traces and LTL properties. This approach is possible to adapt to generate nondeterministic Moore automata, but the performance of the method [7] (although measured on a different type of automata) is reported to be worse: constructing models with 10 states takes 15 minutes or more, while the method from the present paper needs only 1 second for such a task.

B. Empirical Comparison with State Merging

The approach from the work [19] has been adapted to identify automata of the type considered in Section III-A. This approach iteratively runs state merging, restarting the procedure with more data when the automaton under construction does not comply with LTL properties. It has been implemented with the following adjustments:

- In [19], traces are composed of events. In the implemented modification of the method, an event is formed of an input and a set of outputs. A special "empty" event is used for first trace elements.
- The resulting Mealy state machines are converted to Moore ones by considering transitions of the Mealy machine as states of the Moore machine. Transitions labeled with "empty" events become initial states.

TABLE III
STATE MERGING PERFORMANCE ON RANDOM INSTANCES

Initial $\|S\|$	Time, s		Median	Complied with all
	Q_2	Q_3	generated $\|S\|$	LTL properties?
10	1.2	1.6	28	13/50
15	2.4	4.2	47	7/50
20	3.8	10.4	68	1/50
25	6.9	12.5	90	1/50
30	8.8	27.6	108	0/50
35	7.2	21.5	108	0/50

The implemented technique has the following limitations:

- The number of states of the resulting automaton is often larger than the minimum possible one.
- The completeness requirement is not supported.
- Only safety LTL properties are supported.

Applying the state merging approach to the cylinder example resulted in an automaton with 10 states (instead of 3), which violated the completeness requirement. The results of applying this approach to the data from Section IV-D are shown in Table III. By comparing it with Table II, it is visible that state merging is faster, especially for larger initial automata. However, it generates automata roughly three times larger than the initial ones. What is more important, since the randomly generated data contained LTL properties not belonging to the safety class, such properties had to be ignored, and thus were often not satisfied for the resulting automata. Thus, despite being fast, the approach [19] loses in quality.

C. Comparison with Synthesis Involving Real-Valued Data

The goal of the second empirical comparison was to investigate whether existing synthesis techniques can infer plant models with real-valued data values. This might be relevant in the following context. Suppose that the plant's state variables are real-valued. To apply the approach proposed in the present paper, the values of these variables must be first discretized. Instead, if the plant model synthesis method is able to produce models with real-valued values, this would make the model easier to comprehend. Then, the obtained model can be verified as a hybrid system [28], or, alternatively, discretized and model-checked by conventional approaches.

The work [20] presents a tool capable of constructing such rich finite-state models as Mealy machines. This tool implements the approach proposed in the same paper [20], and also another approach called GK-tails [29]. To perform the comparison, both the approaches have been tried on continuous versions of traces for the water level and for the generic PWR model. In these traces, plant model inputs were represented as events, and plant model outputs were recorded as real values annotating the events.

Unfortunately, the approach from [20] produced oversimplified models, which only indicated the possibility of receiving various input events, without any data annotations. As for GK-tails, the produced models had less states than the reference one for the water level and more states (40) for the generic PWR model. The completeness requirement was not satisfied. In particular, around one half of states lacked any outgoing transitions. Thus, applying these models in model checking is problematic.

VI. DISCUSSION AND CONCLUSIONS

In this paper, a method of discrete plant model construction was proposed and evaluated on a number of case studies and random instances. This method was shown to be applicable in aiding closed-loop model checking, which is an important safety and correctness assurance technique for automation systems. Still, several issues should be discussed.

Firstly, the method is intended to construct simplified plant models whose behavior may only resemble the behavior of the original plant. The sources of this imprecision are the small number of states, discretization of continuous parameters and possible incompleteness of the input specification. But even such inaccurate models are useful in model checking since they can find faults in the verified system. The application of the method is, however, limited for plants with complex dynamics: in theory, discretization can potentially be applied to model the plant with any precision, but the computational complexity of the problem will quickly become intractable.

Second, in addition to constructing simplified plant models, the method can also be employed to find minimum models consistent with the given specification, and a smaller number of states means faster verification. Although plant models with fixed numbers of states were often sought, this number can be interpreted as an additional unknown parameter. For example, starting from one and increasing the number of states $|S|$ by one until the problem instance becomes satisfiable allows to find the minimum model.

Next, if one needs to apply the method for complex, modular plants, the main recommendation is to create a separate model for each component of the plant, and then verify each component independently. As for the components, continuous parameters must be discretized.

An alternative approach to handling complex plants is to use only traces as input specification to gain the benefits of the trivial solution described in Section III-C. This solution places little restriction on plant complexity, but temporal properties, which may specify important restrictions on the plant behavior, cannot be supported this way.

The last concern is the source of input specification. Ideally, traces should be obtained based on a more complex, often continuous simulation model (e.g. a hybrid automaton, a set of differential equations, an Apros model), which is in turn obtained by other means. LTL properties can be derived from system requirements and common sense. Ensuring that these properties are satisfied for the model which generated the traces is the responsibility of the user of the method.

Future work may include the evaluation of the method on larger plants and more extensive application of the produced plant models in model checking. Next, the translation-to-SAT technique can be improved, for example by providing better support for commonly used LTL properties and by considering timed automata synthesis. Another direction for improvement is the degree of automation of the method.

ACKNOWLEDGMENT

This work was financially supported, in part, by the SAUNA project (funded by the Finnish Nuclear Waste Management Fund VYR as a part of research program SAFIR2018) and by the Government of Russian Federation, Grant 074-U01.

We thank Antti Pakonen from VTT Technical Research Centre of Finland for formal modeling of the control logic of the generic PWR model.

REFERENCES

[1] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[2] S. Preuße, H. Lapp, and H. Hanisch, "Closed-loop system modeling, validation, and verification," in *17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2012, pp. 1–8.

[3] C. Gerber, S. Preuße, and H.-M. Hanisch, "A complete framework for controller verification in manufacturing," in *15th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2010, pp. 1–9.

[4] S. Preuße, *Technologies for Engineering Manufacturing Systems Control in Closed Loop*. Logos Verlag Berlin GmbH, 2013, vol. 10.

[5] C. Pang and V. Vyatkin, "Automatic model generation of IEC 61499 function block using net condition/event systems," in *6th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2008, pp. 1133–1138.

[6] C.-H. Cheng, C.-H. Huang, H. Ruess, and S. Stattelmann, "G4LTL-ST: Automatic generation of PLC programs," in *Computer Aided Verification*. Springer, 2014, pp. 541–549.

[7] D. Chivilikhin, V. Ulyantsev, and A. Shalyto, "Combining exact and metaheuristic techniques for learning extended finite-state machines from test scenarios and temporal properties," in *13th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2014, pp. 350–355.

[8] A. Biere, M. Heule, and H. van Maaren, Eds., *Handbook of satisfiability*. IOS press, 2009.

[9] I. Buzhinsky and V. Vyatkin, "Plant model inference for closed-loop verification of control systems: Initial explorations," in *14th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2016, pp. 736–739.

[10] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[11] M. Rausch and H.-M. Hanisch, "Net condition/event systems with multiple condition outputs," in *INRIA/IEEE Symposium on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 1995, pp. 592–600.

[12] J. Machado, B. Denis, and J.-J. Lesage, "Formal verification of industrial controllers: with or without a plant model?" in *7th Portuguese Conference on Automatic Control (CONTROLO)*, 2006.

[13] ——, "A generic approach to build plant models for DES verification purposes," in *8th International Workshop on Discrete Event Systems (WODES)*. IEEE, 2006, pp. 407–412.

[14] H.-T. Park, J.-G. Kwak, G.-N. Wang, and S. C. Park, "Plant model generation for PLC simulation," *International Journal of Production Research*, vol. 48, no. 5, pp. 1517–1529, 2010.

[15] M. Roth, L. Litz, and J.-J. Lesage, "Identification of discrete event systems: Implementation issues and model completeness," in *7th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, vol. 3, 2010, pp. 73–80.

[16] M. J. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering, Springer*, vol. 18, no. 4, pp. 825–856, 2013.

[17] V. Ulyantsev and F. Tsarev, "Extended finite-state machine induction using SAT-solver," in *14th IFAC Symposium "Information Control Problems in Manufacturing (INCOM)"*. IFAC, 2012, pp. 512–517.

[18] A. Maier, A. Vodencarevic, O. Niggemann, R. Just, and M. Jaeger, "Anomaly detection in production plants using timed automata," in *8th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, 2011, pp. 363–369.

[19] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2008, pp. 248–257.

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

[21] D. Chivilikhin and V. Ulyantsev, "MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines," in *15th Annual Conference on Genetic and Evolutionary Computation (GECCO)*. ACM, 2013, pp. 511–518.

[22] R. Ehlers, "Symbolic bounded synthesis," *Formal Methods in System Design, Springer*, vol. 40, no. 2, pp. 232–262, 2012.

[23] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing finite-state protocols from scenarios and requirements," in *Hardware and Software: Verification and Testing*. Springer, 2014, pp. 75–91.

[24] S. Patil, V. Vyatkin, and M. Sorouri, "Formal verification of intelligent mechatronic systems with decentralized control logic," in *17th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2012, pp. 1–7.

[25] J. Peltola, S. Sierla, P. Aarnio, and K. Koskinen, "Industrial evaluation of functional model-based testing for process control applications using CAEX," in *18th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2013, pp. 1–8.

[26] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A toolset for model checking of PLC software," in *18th IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, 2013, pp. 1–6.

[27] A. Duret-Lutz, "Manipulating LTL formulas using Spot 1.0," in *Automated Technology for Verification and Analysis*. Springer, 2013, pp. 442–445.

[28] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical computer science*, vol. 138, no. 1, pp. 3–34, 1995.

[29] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 501–510.

Igor Buzhinsky received the B.Sc. and M.Sc. degrees in applied mathematics and informatics from ITMO University, St. Petersburg, Russia, in 2013 and 2015, and the M.Sc. degree in software engineering and service design from University of Jyväskylä, Finland, in 2015. He is currently a doctoral candidate with Aalto University and a software engineer with ITMO University. His research interests include software testing, verification and formal model synthesis applied to industrial automation systems.

Valeriy Vyatkin (M'03, SM'04) received Ph.D. degree from the State University of Radio Engineering, Taganrog, Russia, in 1992. He is on joint appointment as Chaired Professor (Ämnesföreträdare) of Dependable Computation and Communication Systems, Luleå University of Technology, Luleå, Sweden, and Professor of Information and Computer Engineering in Automation at Aalto University, Helsinki, Finland. Previously, he was a Visiting Scholar at Cambridge University, U.K., and had permanent academic appointments with the University of Auckland, Auckland, New Zealand; Martin Luther University of Halle-Wittenberg, Halle, 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 applied in various industry sectors, including smart grid, material handling, building management systems, data centres and reconfigurable manufacturing.

Dr. Vyatkin was awarded the Andrew P. Sage Award for the best IEEE Transactions paper in 2012.