Scalable methods of discrete plant model generation for closed-loop model checking

Buzhinsky, Igor; Pakonen, Antti; Vyatkin, Valeriy

Published in:
Proceedings IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society

DOI:
10.1109/IECON.2017.8216949

Published: 15/12/2017

Document Version
Peer reviewed version

Scalable Methods of Discrete Plant Model Generation for Closed-Loop Model Checking

Igor Buzhinsky[1, 2], Antti Pakonen[3], Valeriy Vyatkin[1, 4]
[1] Department of Electrical Engineering and Automation, Aalto University, Finland
[2] Computer Technology Department, ITMO University, St. Petersburg, Russia
[3] VTT Technical Research Centre of Finland Ltd, Finland
[4] Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, Sweden
igor.buzhinskii@aalto.fi, antti.pakonen@vtt.fi, vyatkin@ieee.org

***Abstract*—To facilitate correctness and safety of mission-critical automation systems, formal methods should be applied in addition to simulation and testing. One of such formal methods is model checking, which is capable of verifying complex requirements for the system's model. If both the controller and the controlled plant are formally modeled, then the variant of this technique called closed-loop model checking can be applied. Recently, a technique of automatic plant model generation has been proposed which is applicable in this scenario. This paper continues the work in this direction by presenting two plant model construction approaches which are much more scalable with respect to the previous one, and puts this work into a more practical context. The approaches are evaluated on a case study from the nuclear automation domain.**

I. Introduction

Reliability is a crucial requirement for industrial automation systems. As a traditional approach of verification and validation (V&V), testing is widely applied. However, certain mission-critical systems such as aerospace and nuclear instrumentation and control (I&C) ones require stronger guarantees of correctness and safety. Among well-known techniques offering deeper analysis is model checking [1], a formal verification technique which can prove system correctness (or, more precisely, the correctness of the system's formal model) for the entire range of its possible behaviors.

In the industrial automation context, at least two separate formal models are often considered: plant process model (later referred to as the plant model) and the automation system model (from now on, the controller model). Having only the controller model, one can verify systems in open loop [2]: the controller's correctness is checked without considering the environment where it is operating. If the plant model is also available, the more natural closed-loop modeling and verification [3]–[6] become possible.

Apart from the actual formal verification, the challenge of formal model construction is crucial: it often requires manual effort, making the prerequisites of model checking harder to achieve and introducing the so-called human factor, a major source of errors. For plant models in particular, a model constructed manually might be too detailed and thus difficult to verify, and adjusting it to make verification feasible would require even more effort. Thus, approaches have been proposed to construct plant models automatically [6], [7].

This paper builds on top of the work [6] and presents two plant model construction methods whose main advantage over [6] is the degree of scalability, which was limited in [6] due to the use of SAT solvers. As input data, we use system behavior traces collected with the help of a simulation model of the system. This choice puts plant model construction in a more practical context, which is demonstrated by evaluating the proposed methods on a case study based on a nuclear power plant (NPP) simulation model. Then, tuning model generation parameters such as the size of the trace set and the strength of discretization enables finding the right trade-off between the precision of the model and the computational complexity of its verification.

The paper is structured as follows. Section II introduces concepts used throughout the paper. Then, Section III describes the proposed plant model generation techniques. In Section IV, the techniques are evaluated on a case study. The results are discussed in Section V.

II. Preliminaries

A. Simulation environments

Specialized simulation environments aid the construction of automation systems and allow simulating them. Typically, these environments present means of constructing both the controller model (that is, the entity whose correctness must be ensured), and the plant model, with which the controller operates. In this paper, the Apros[1] continuous process simulator is used. In Apros, the automation system can be represented as a number of function block diagrams, either expressing control logic or the plant's mechatronic aspects.

B. Discrete formal models

While simulation models are created for simulation and testing, *formal models* are intended to be applicable for formal verification. They are often represented in finite-state formalisms, such as finite-state machines [8], timed automata [9] and Petri nets [10]. These kinds of formal models are applicable for verification by means of model checking, which is described in Section II-C. Simulation models, in contrast, cannot be directly applied in model checking: they typically contain

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

continuous parameters which are not adequately supported by modern model checkers.

Both the controller and the plant can be represented not only by simulation models, but also by formal models. According to [11], formal plant models are subdivided into *detailed* and *abstract* ones: while detailed models preserve the internal structure and parameters of the plant, abstract models are more simple and only convey its external behavior. Plant model construction in the present paper is based on traces showing external behavior, so the inferred models are abstract.

C. Model checking

Model checking [1] is a formal verification technique which exhaustively explores the state space of the system's formal model to check certain requirements, which may involve statements over the model's state at different moments of time. The most common formal languages used to formulate such *temporal properties* are the linear temporal logic (LTL) [12] and the computation tree logic (CTL) [13]. They operate with the simplest, logical model of time: it is measured as the number of executed state transitions, or discrete steps. The problem of model checking is, given a discrete finite-state model and a temporal requirement, to determine whether this requirement is satisfied for this model, and, if it is not, optionally provide a counterexample trace.

In CTL, constraints over the system's formal model are expressed with Boolean connectives and *temporal operators*. For example, operators $\mathbf{AG}$, $\mathbf{AF}$ and $\mathbf{AX}$ express that their argument is satisfied always, eventually or on the next step of the behavior respectively for all future behaviors of the model. Corresponding operators $\mathbf{EG}$, $\mathbf{EF}$ and $\mathbf{EX}$ formulate the same requirements for at least one future behavior.

D. Model checking of automation systems

Model checking of automation systems can be performed in either open loop or closed loop [3]–[6]. The work [14] references several concrete open-loop and closed-loop approaches. While the closed-loop approach explicitly considers the model of the plant, in open-loop model checking, a more traditional and simple approach, the plant model is reduced to plant sensor measurements which are assumed to have arbitrary values and are independent from each other. This can result in counterexample behaviors that demonstrate scenarios that are not possible in the real world. Such scenarios can, for example, involve measurements radically changing their values in an instant, which is impossible for physically slow processes. Ruling out such counterexamples manually costs effort.

On the other hand, nuclear automation systems are designed to be fault tolerant, and an analyst should not automatically disregard a counterexample that shows "impossible" plant response since it may be relevant due to possible failures of plant sensors. Thus, from the safety point of view, closed-loop modeling represents a challenge. By closing the loop, the state space of the model is usually reduced [3], and it is possible that relevant behaviors (that would reveal a design issue) are accidentally filtered out [15].

A comparison of open-loop and closed-loop model checking is provided in [16]. According to this work, these approaches are complementary to each other. Then, in closed-loop model checking it is possible to check temporal properties which involve plant variables unobservable by the controller. Model checking results of other properties may be different in the open-loop and closed-loop cases.

E. Model checking tools

Two model checking tools are involved in the present study. The first tool is the symbolic model checker NuSMV [17]. It accepts the model and the specification to be checked in the form of a text file. The model has a number of variables with values from finite ranges. The initial state of the model and allowed state changes are expressed with constraints which form two formulas, `INIT` and `TRANS`. While `INIT` simply defines which value combinations of state variables form valid initial states, `TRANS` is formulated over two variable set: the set of current variables, denoted by variables names, and the set of next-step variables, whose names are additionally marked by the keyword `next`.

By means of such constraints, expressing finite-state machines becomes straightforward in NuSMV. The second involved tool, called MODCHK, is the enhancement of the toolset presented in [18] and can be viewed as a user-friendly interface to NuSMV which allows to specify and verify automation system as a number of function block networks. MODCHK is also partially integrated with Apros.

F. Related work

In [7], simplified simulation plant models are synthesized based on source code of PLC programs. Formal modeling is not considered in [7], and the construction method works only for manufacturing systems. In [19], formal discrete-state models of the entire closed-loop system are constructed as nondeterministic finite-state event generators using the data obtained from a running real-world controlled plant. This approach, however, is not suitable for mission-critical systems such as NPPs, which must be verified prior to operation. The technique [20] constructs automation system models to detect anomalies, but not for formal verification. In [5] and [21], plant models are built based on 3D CAD drawings, but the construction process is not fully automated. Another field of research is construction of software models [22].

Finally, the work [6] deals with automatic formal plant model construction from behavior examples and LTL specification of the plant. Using such a plant model, it is potentially possible to verify multiple controllers prior to their deployment. However, the method proposed in [6] has limited scalability and cannot construct plant models with hundreds or thousands of states given hundreds or thousands of behavior examples. The present paper develops the work [6], mitigating the scalability issue by focusing on a more specific problem where only behavior examples, or traces, are available.

III. Plant model construction

Following some preliminary definitions, two plant model construction methods are proposed in this section. Both approaches are currently implemented to produce NuSMV models, but can be modified to support other formats. Their implementation is available online as a part of the extended finite-state machine construction toolset.[2]

A. Definitions

The source of data for automatic plant model construction is a set of *traces*, or input-output behavior examples of the plant. Since the plant model is the part to be constructed automatically, traces will be viewed in terms of its interface. First, the plant model has a set of *inputs* $\mathcal{I} = \{i_1, ..., i_k\}$, each of which can be either Boolean or real-valued (continuous). Inputs describe the state of plant actuators, which receive information from the controller. Second, the plant model has a set of Boolean or real-valued *outputs* $\mathcal{O} = \{o_1, ..., o_m\}$. Together, plant model inputs and outputs will be referred to as plant model *parameters*. Plant and controller model interaction is cycle-based: on each cycle the controller model first reads plant model outputs and then produces inputs to the plant model. The duration of each cycle is fixed.

A *trace element* is a pair (O, I), where O is a list of m outputs and I is a list of k inputs. A *raw trace* of length ℓ is a finite sequence of trace elements $((O_1, I_1), ..., (O_\ell, I_\ell))$. Since model checking usually operates with discrete values, continuous parameters must be discretized. This is done by distributing the values of each continuous parameter into a finite number of intervals, the indices of which will be referred to as *discrete levels*. A *discretized trace* is a sequence $((O'_1, I'_1), ..., (O'_\ell, I'_\ell))$ obtained from the raw trace by replacing each real value with the corresponding discrete level. Boolean values remain unchanged (i.e. they have two discrete levels). Fig. 1 shows an example of a trace and illustrates the process of trace discretization.

Assuming that traces are available and discretized according to the selected thresholds, the problem of plant model construction can be defined. Let $v(\mathcal{I})$ and $v(\mathcal{O})$ be the sets of all discrete level combinations of plant model inputs and outputs respectively. A *plant model* is a tuple $(S, S_0, v(\mathcal{I}), v(\mathcal{O}), T, \lambda)$, where S is the finite set of states, $S_0 \subset S$ is the non-empty set of initial states, $v(\mathcal{I})$ and $v(\mathcal{O})$ are the input and output sets, $T \subset S \times v(\mathcal{I}) \times S$ is the transition relation, and $\lambda : S \to v(\mathcal{O})$ is the output function. This definition is a variant of the definition of a nondeterministic Moore machine [8], which has been partially adopted from [6]. Assuming that the plant model interacts with the controller model in the cycle-based way, the following requirements must be satisfied:

1) the plant model must have the given discretized traces among its valid finite input-output behaviors (i.e the ones which start in S_0 and proceed according to T);
2) in each state, the behavior of the plant model must be defined for each possible input combination (states violating this condition, or *deadlocks*, are undesired in model checking);
3) some input-output behaviors must be impossible for the plant model (this requirement is not formalized, but the choice of these behaviors must depend on traces).

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

B. Explicit-state method

In the *explicit-state method*, the formal plant model is sought according to its definition, as a nondeterministic Moore machine, where each state corresponds to certain discrete levels of plant model outputs. To construct the state machine, the following procedure, partially adopted from [6], is applied. For each element of $v(\mathcal{O})$ found in discretized traces, a state is created. This state is included into S_0 if and only if it corresponds to the beginning of some trace. For each pair of contiguous discretized trace elements $((O'_1, I'_1), (O'_2, I'_2))$, a transition labeled with I'_1 is added from O'_1 to O'_2.

To prevent deadlocks, if some transitions are missing after the described procedure is applied, they must be added. Differently from [6], the unknown input combination is directed to the same states as the closest (averaged over all inputs) combination for which a transition sourcing in the considered state exists. Depending on whether a transition has been added based on this completion procedure or traces, it will be called either an *unsupported* or a *supported* one, respectively. Transitions triggered by input combinations absent in traces (*unknown* input combinations) are added in a different way: if ordinary transitions are present from O'_i to $O'_{j_1}, ..., O'_{j_n}$, then a transition group is added for each unknown input combination to each of $O'_{j_1}, ..., O'_{j_n}$. Fig. 2 shows an example of a state machine constructed using the discretized trace from Fig. 1.

Finally, the finite-state machine is explicitly encoded in NuSMV, which is a straightforward process. Discrete levels of real-valued parameters are mapped to corresponding numeric intervals (only integer values are allowed in NuSMV). During the encoding, additional *fairness constraints* are added to prevent self-loops from being executed eternally. Self-loops are ubiquitous in plant models due to the common situation when none of the plant model outputs change their discrete levels between consequent trace elements. Paths eternally taking these loops often prevent target plant states from being reached (since the state of the plant remains unchanged forever), which causes violations of liveness requirements. Fairness constraints remove such behaviors from consideration. In NuSMV, these constraints are introduced by defining a variable indicating that the state of the plant has changed during the last step, and adding a `FAIRNESS` declaration for this variable, obliging it to be true infinitely often.

C. Constraint-based method

The rationale behind the *constraint-based*, or the *symbolic method*, was to produce condensed models with shorter NuSMV representations which are easier for NuSMV to process. Instead of explicitly defining states and transitions, this method constrains plant model parameters and their changes between two adjacent steps. In the NuSMV model, a separate

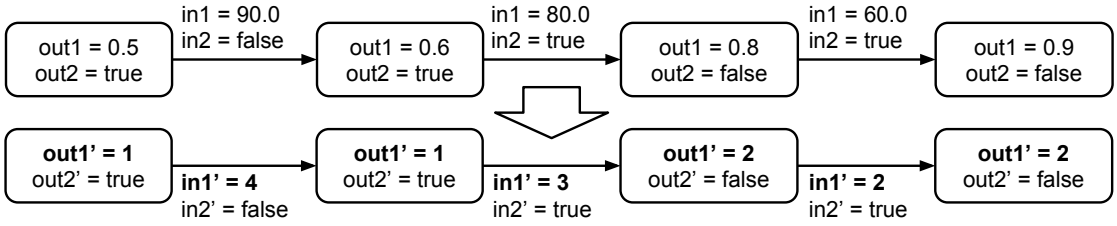

Fig. 1. Trace discretization. Real-valued input `in1` is discretized into intervals $[0.0, 25.0), [25.0, 50.0), [50.0, 75.0), [75.0, 100.0)$ indexed from 1 to 4. Real-valued output `out1` is discretized into intervals $[0.0, 0.75), [0.75, 1.5)$ indexed from 1 to 2.

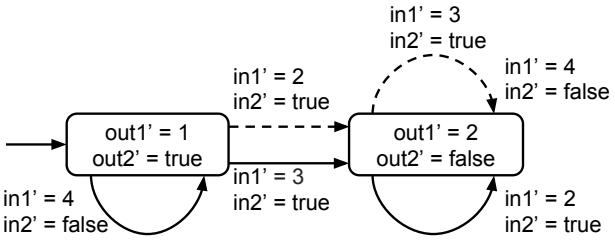

Fig. 2. Example of a plant model constructed by the explicit-state method given the trace from Fig. 1. Unsupported transitions are shown in broken lines. The initial state is indicated by the arrow without a source state. Transitions for unknown input combinations are not shown and are assumed to lead to both states from the left state and only to the right state from the right state.

state variable is declared for each of m plant model outputs. The values of these variables encode discrete levels of the outputs. Then, using the data from the traces, the following constraints are added to the model:

1) each output may only have values found in the traces;
2) for each pair of outputs, only value pairs found in some trace element are possible;
3) for each output, only its value transitions found in some pair of contiguous trace elements are possible;
4) for each pair of an input and an output, only output values are possible which occur after the given input value in some pair of contiguous trace elements.

The transition relation of the model permits all possible transitions satisfying these constraints. Similarly to the explicit-state method, fairness constraints preventing eternal self-loop executions are added (a self-loop is assumed to be executed when none of discrete levels of plant model outputs change).

Nontrivial constraints generated from the discretized trace in Fig. 1 are expressed in NuSMV in Table I. Constraints of types 1–2 are expressed twice: for the initial states (with the `INIT` keyword) and for each consecutive state (with the `TRANS` and `next` keywords). Note that primes are not allowed in NuSMV, but they are shown in the table for convenience. The same plant model is also illustrated in Fig. 3. Note the difference between this model and the one from Fig. 2.

IV. Experimental evaluation

In this section, the proposed plant model generation methods are evaluated on a case study. All experiments were performed on the Intel Core i7-4510U CPU with the clock rate of 2 GHz.

TABLE I
NuSMV example of a constraint-based plant model

Type	Constraints
1	`INIT out1' in {1, 2}`
	`TRANS next(out1') in {1, 2}`
2	`INIT out1' = 1 & out2' | out1' = 2 & !out2'`
	`TRANS next(out1') = 1 & next(out2') | next(out1') = 2 & !next(out2')`
3	`TRANS (out1' = 1 & next(out1') in {1, 2} | out1' = 2 & next(out1') = 2) & (!out2' & !next(out2') | out2')`
4	`TRANS (in1' = 4 -> next(out1') = 1 & next(out2')) & (in1' = 3 -> next(out1') = 2 & !next(out2')) & (in1' = 2 -> next(out1') = 2 & !next(out2')) & (!in2' -> next(out1') = 1 & next(out2')) & (in2' -> next(out1') = 2 & !next(out2'))`

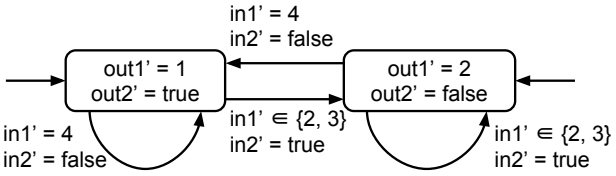

Fig. 3. Plant model constructed by the constraint-based method given the trace from Fig. 1 and represented as a state machine. The figure corresponds to the constraints from Table I. Transitions for unknown input combinations are not shown and are assumed to lead to both states.

A. Case study

For the case study, a generic simulation model of an NPP with a pressurized water reactor (PWR), hereafter referred to as the generic PWR model, was used. This Apros model was provided by Fortum Power and Heat Oy,[3] a power utility with NPP operation license in Finland, and includes the most important process, mechatronic and control components of an NPP, and corresponding automation logic. Due to its large size, the case study considered only eight automation block networks out of 44 networks modeled in Apros. The names of these eight subsystems were masked and are indicated as S1, ..., S8. Subsystems S1, ..., S5 are responsible for activating protection functions, and the rest control pressures and liquid levels in tanks. Controller NuSMV models for the selected subsystems were constructed using MODCHK.

Functional CTL requirements were prepared for each sub-

[3]http://www.fortum.com/

system using the documentation of the generic PWR model, its actual implementation and our own understanding of the model. *Controller-only requirements* are the ones which are sensible to be verified without a plant model (they do not depend on feedback between the plant and the controller). The most common examples of such requirements are of the request-response type: they require some controller outputs (e.g. an activation of a protection function) given some conditions over inputs. In contrast, *plant-and-controller requirements* are not intended for open-loop verification as they restrict plant model outputs which are assumed to be unrestricted in open-loop verification. For example, the following plant-and-controller requirements were formulated for S7:

1) $\mathbf{AG}(p < p_1 \rightarrow \mathbf{AF}\, p > p_1)$: "always, if pressure p goes below p_1, eventually it will be above p_1" for several low pressures p_1;
2) $\mathbf{AG}(p > p_2 \rightarrow \mathbf{AF}\, p < p_2)$: "always, if pressure p goes above p_2, eventually it will be below p_2" for several high pressures p_2.

Simulating the generic PWR model, we collected 3000 traces, each with 240 elements corresponding to a four minute long simulation with the sampling time of one second.

B. Plant model construction

Maximum construction times of explicit-state and constraint-based plant models were 4.7 and 3.1 minutes respectively. The numbers of states in the explicit-state model and constraints in the constraint-based model are shown in Table II. The diversity of these numbers indicates that models of various complexities are included in the case study. A more interesting property of plant models is the percentage of supported transitions (also shown in Table II), which are more reliable than unsupported ones. This metrics characterizes the sufficiency of traces and the adequacy of plant parameter discretization given the traces. From the data we can see that the case study may be further improved, but, on the other hand, we are able to investigate plant models with the majority of transitions created according to our heuristic completion procedure.

C. Closed-loop model checking

Table II shows the results of closed-loop verification with generated plant models: the numbers of satisfied requirements (out of the maximum possible numbers) are presented together with model checking time. Each model checking run was limited to 48 hours. As visible from the table, verification times are smaller for constraint-based models. Still, there are subsystems for which verification even with constraint-based models is infeasible in realistic time. The supposed reason for this is the complexity of controller models.

There are cases of requirement violations. On one hand, violations (which are limited to the case of plant-and-controller requirements) may be explained by the wrong understanding of the generic PWR model by the authors. On the other hand, a smaller, simpler case study may be beneficial to analyze the relation of such violations with plant model reliability.

D. Example

Below, we focus on S7, a rather simple subsystem compared to others. Responsible for pressurizer pressure control, S7 controls spray valves and the pressurizer heater (9 real-valued plant model inputs) given the pressure p and the liquid level l in the pressurizer (2 real-valued plant model outputs). When p is too high, closing signals for pressurizer spray valves are generated. Then, the pressurizer heater power depends on l, and the pressurizer heater is turned off if l is too low.

All controller-only requirements, which related controller's inputs and outputs, were satisfied in verification with both plant models. However, such a result is not surprising since they were also satisfied in open-loop model checking. A more interesting situation was observed for plant-and-controller requirements given in Section IV-A. Despite S7 being responsible for pressurizer pressure control, none of the requirements were satisfied for the constraint-based model and only requirements of the second type were satisfied for the explicit-state model. By examining the traces, we were able to see that such requirements were already violated in them, i.e. the generic PWR model may show behaviors violating the requirements, and the constructed constraint-based model is able to point this. Then, the examination of the explicit-state model revealed that some of the requirements were satisfied due to the inability of the model to leave some of its states given the existing controller – paths leading to such states were excluded from consideration due to fairness constraints.

V. Discussion and conclusions

We have presented two methods of plant model synthesis (the explicit-state and the constraint-based ones) for closed-loop model checking and evaluated them on a case study involving an NPP simulation model. Both methods generalize simulation runs of the closed-loop system: while the explicit-state method treats occurrences of all plant model parameters as separate states, the constraint-based method infers dependencies between limited subsets of these parameters. According to Table II, the constraint-based method is superior over the explicit-state one in terms of time required for model checking. On the other hand, tight integration of MODCHK with NuSMV prevented us from exploring the case of explicit-state model checking (e.g. using the verifier SPIN) – it may potentially speed up verification of explicit-state models due to the small number of states in them.

Compared to the previous method of the authors [6], the proposed methods are to a large extent more scalable. While the method [6] generated models with up to 80 states from traces with up to 1000 elements in total, the proposed explicit-state method was able to handle 720000 trace elements and construct plant models with up to 4906 states. Supporting the same input data complexity, the constraint-based method constructs more concise models with more states. On the other hand, the proposed methods do not support LTL properties for the plant model as input data, but such properties may be difficult to obtain in practical cases, while the demand to construct larger models from larger trace sets does exist.

TABLE II
PROPERTIES OF GENERATED MODELS AND RESULTS OF CLOSED-LOOP MODEL CHECKING. TIME LIMIT IS INDICATED AS "TL"

Subsystem		S1	S2	S3	S4	S5	S6	S7	S8
Explicit-state method	Number of states	14	1355	1206	192	4906	1904	20	100
	Fraction of supported transitions	0.754	0.062	0.025	0.411	0.360	0.303	0.598	0.714
	Model checking time (min)	387.2	TL	278.6	TL	687.1	TL	2.1	1.0
	Requirements satisfied	5 / 9	– / 30	38 / 38	– / 17	16 / 16	– / 30	18 / 21	8 / 20
Constraint-based method	Number of constraints	36	576	1275	144	357	234	26	20
	Model checking time (min)	0.3	TL	278.6	616.3	0.1	TL	0.1	0.1
	Requirements satisfied	5 / 9	– / 30	38 / 38	13 / 17	16 / 16	– / 30	11 / 21	8 / 20

Our approach has a number of limitations. First, correctness of the simulation model, which is usually constructed manually and hence is also influenced by the human factor, is not addressed. Then, generated models do not exhaustively represent possible plant behaviors since the input set of simulation traces is incomplete. On the other hand, no model can guarantee exhaustive analysis, and finding the trade-off between the richness of the model and the feasibility of verification is inevitable.

Considering the mentioned limitations, the issue of reliability of generated models is still important – otherwise, the methods will not be adopted in industrial practice. In Section IV-D, we have found that fairness constraints may cause some requirements to be satisfied while they must be violated. The issue of model reliability is also connected with the preparation of data for the methods. To address it, future work may involve adjustment of fairness constraints, new case studies, evaluation of the sufficiency of available traces, means to record them in order to maximize their utility in terms of model quality, explorations of strategies of parameter discretization, and selection of proper cycle duration.

ACKNOWLEDGMENTS

This work was financially supported by the SAUNA project (funded by the Finnish Nuclear Waste Management Fund VYR as a part of research program SAFIR2018) and by the Ministry of Education and Science of the Russian Federation, project RFMEFI58716X0032.

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[2] J. Lahtinen, T. Kuismin, and K. Heljanko, "Verifying large modular systems using iterative abstraction refinement," *Reliability Engineering & System Safety*, vol. 139, pp. 120–130, 2015.

[3] 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.

[4] 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.

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

[6] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1521–1530, Aug 2017.

[7] 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.

[8] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011.

[9] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on concurrency and Petri nets*. Springer, 2004, pp. 87–124.

[10] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[11] M. Perin and J.-M. Faure, "Comparing detailed and abstract timed models of automated discrete manufacturing systems," in *2013 IEEE International Conference on Automation Science and Engineering (CASE)*. IEEE, 2013, pp. 916–923.

[12] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977, pp. 46–57.

[13] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*. Springer, 1981, pp. 52–71.

[14] G. Frey and L. Litz, "Formal methods in PLC programming," in *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, vol. 4. IEEE, 2000, pp. 2431–2436.

[15] A. Pakonen, J. Valkonen, S. Matinaho, and M. Hartikainen, "Model checking for licensing support in the Finnish nuclear industry," in *International Symposium on Future I&C for Nuclear Power Plants (ISOFIC)*. Korean Nuclear Society, 2014.

[16] 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.

[17] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[18] 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.

[19] 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.

[20] 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.

[21] E. Lobo, J. Fertuzinhos, J. P. Silva, and J. Machado, "Obtaining plant models for formal verification tasks from 3D CAD models: Which is the best approach?" in *Advanced Materials Research*, vol. 630. Trans Tech Publ, 2013, pp. 283–290.

[22] 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.