Active learning of formal plant models for cyber-physical systems

Polina Ovsiannikova*, Daniil Chivilikhin*, Vladimir Ulyantsev*,
Andrey Stankevich*, Ilya Zakirzyanov*, Valeriy Vyatkin*†‡, and Anatoly Shalyto*
*Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia
Email: polina.ovsyannikova@corp.ifmo.ru, chivdan@rain.ifmo.ru, ulyantsev@rain.ifmo.ru, shalyto@mail.ifmo.ru
†Department of Electrical Engineering and Automation, Aalto University, Finland
Email: vyatkin@ieee.org
‡Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden

***Abstract*—As the world becomes more and more automated, the degree of cyber-physical systems involvement cannot be overestimated. A large part of them are safety-critical, thus, it is especially important to ensure their correctness before start of operation or reconfiguration. For this purpose the model checking approach should be used since it allows rigorously proving system correctness by checking all possible states. To ensure the compliance of controller-plant properties with system requirements, the closed-loop verification approach should be chosen, which requires not only a formal model of the controller, but also a formal model of the plant. In this paper we propose an approach for constructing formal models of context-free deterministic plants automatically using active learning algorithms. The case study shows its successful application to plant model generation for the elevator cyber-physical system.**

I. Introduction

Even if cyber-physical system (CPS) functionality does not include immediate interaction with humans, it still can influence them indirectly. Therefore, among other reasons, correct behavior of CPS is required to protect lives. The compliance of system implementation with its specification is commonly checked with simulation and testing. However, even automated testing is limited to checking only a limited number of behaviors defined in test cases. Formal verification using *model checking* [1] is a more comprehensive approach. It examines the entire state space of the system for errors, so that deviations that are inconspicuous during testing can be detected. In a number of works it was argued that in CPS applications closed-loop model checking [2], [3] is preferable to the open-loop approach, which is commonly used for computer programs verification.

In the closed-loop approach, a necessary part of the process is constructing a formal model of the plant. In most cases formal modeling is performed manually which may be very resource consuming. In order to reduce the influence of the human factor and keep the formal model consistent and up-to-date, approaches for automated plant model generation using behavior examples, or traces, were developed.

Approaches to formal model inference can be classified by the ability to interact with the modeled object (simulation model) during the construction process. *Passive learning* approaches build models from data collected before the learning process [4]–[6]. Thus, together with the model inference method, a way of gathering traces should be developed, that provides thorough coverage of system states. Without knowledge about coverage, measuring the similarity between the resulting formal model and the real system is complicated, although research has been done in this area [7]. The second approach to model inference is *active learning* [8] that can be introduced by the well-known L* algorithm [9] and its modifications, e.g. [10], [11]. The essence of active learning algorithms is constant communication of the algorithm with the simulation model during the inference process. This allows refining the model on every iteration and querying the system with the exact requests needed to determine its behavior.

In this paper we introduce the active learning approach for automatic formal models generation of deterministic discrete context-free plants in a black-box scenario. We also show why the most popular active learning algorithm L* and similar ones are not quite helpful when it comes to CPS plant models inferring.

II. Related work: active learning

The most widely known active learning algorithm L* [9] can be used to infer a Deterministic Finite Automaton (DFA) that accepts the same language as the source DFA. This approach is based on the *Nerode congruence theorem* [12], which states that two words u and u' belonging to the same language are equivalent, if and only if there are no *distinguishing suffixes* for them. Suffix v is distinguishing if, when concatenated with word u and u', it makes either uv or $u'v$ belong to language L, but not both.

The essence of the L* algorithm is iterative hypothesis automaton refinement based on counterexamples that are received after comparing the source automaton with the hypothesis one. Consequently, the algorithm consists of the following steps: (1) hypothesis automaton construction, (2) checking whether the hypothesis automaton is equivalent to the source automaton, (3) processing a counterexample if it exists. Having the source DFA `A` and hypothesis DFA `H`, let us describe each step in more detail.

During the first step, a hypothesis automaton `H` is built according to the *Observation Table (OT)* that contains informa-

978-1-5386-4829-2/18/$31.00 ©2018 IEEE

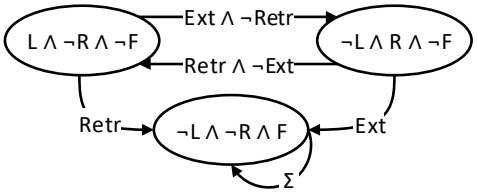


Fig. 1. Moving cylinder plant model

tion about acceptance of particular words by A. This knowledge is collected by sending *Membership Queries* (MQ) – requests to A that contain a sequence to check, on which A responds with True if the sequence belongs to the language or False in the opposite case. To proceed to hypothesis inference, *closedness* and *consistency* OT properties should be satisfied.

On the second step of the algorithm, H is compared to A by means of an *Equivalence Query* (EQ). In the original description of L*, this step is not detailed as the existence of an oracle is assumed: if H is found to be equivalent to A, the oracle reports "YES"; otherwise, it returns a *counterexample* word that is accepted by A, but not by H, or vice versa.

On the last step, a counterexample, if it exists, is processed. All its prefixes are added to the rows of the OT, and then the OT is filled with the results of new MQs and the the first step is repeated. If the oracle reports YES, then H is the resulting DFA, and the algorithm terminates.

In this work the L* algorithm was implemented and tested on the example of inferring a formal model of the plant for a moving cylinder (Fig. 1). The system has two Boolean input variables: Ext (Extend) and Retr (Retract), and three Boolean output variables: L (Left), R (Right) and F (Failure). Thus, in terms of the L* algorithm, there are 2^2 input symbols (all combinations of input variable values) and 2^3 output symbols. The plant is conveniently represented as a Moore machine. Hence, L* was adapted to learn deterministic Moore machines as follows: the cells of OT now store the output symbols generated by the source system in response to MQs instead of labels indicating existence of specific sequences in the source language.

Since the plant is considered as a black box and no oracle is provided, the EQ was implemented in the following way: the system was queried with random sequences that were not equal to the checked ones during OT construction, with their sizes up to the double number of states of the hypothesis automaton.

The constructed model was converted to the input format of the symbolic model checker NuSMV [13]. Once the model was constructed, system specification expressed with Linear Temporal Logic (LTL) properties was checked using NuSMV. The results of model checking are shown in Table I: the model of the system satisfies all considered LTL constraints.

However, since in a black box scenario an oracle is not available and EQ can only be approximated by heuristics, it is impossible to claim that the constructed formal model is equivalent to the source model. Also note that L* was developed for learning regular languages or context-dependent systems – behavior of such systems depends not only on the current input and output variables, but on the *history of interaction*, or the *context*. The absence of context and internal variables in a system means that regardless of the input sequence that has led the system to some particular state, future states will be determined by input symbols only.

Therefore, L* can be applied for learning models of plants with context, though facing the mentioned issue with EQ. Furthermore, additional difficulties are introduced by processing continuous variables – this would require serious modifications of L* while the result would still be inexact due to the heuristic nature of EQ.

However, it can be argued that in a CPS the plant should be independent of the context. Indeed, many systems comply with this assumption. Therefore, the method of inferring context-free systems was developed that does not require an oracle or EQ and where reliability of the resulting model is determined not via automata comparison but by the algorithm termination condition.

III. Proposed approach

Consider a context-free deterministic discrete system with no internal variables nor explicit time dependence. The combination of all input variable values passed to the system in a request is called an *Input Symbol* and is denoted as a tuple $(v(I_1), v(I_2), \ldots, v(I_n))$, where $v(I_1), \ldots, v(I_n)$ are input variable values. Similarly, the combination of all output variable values produced by the system as a response to a request is an *Output Symbol*: $(v(O_1), v(O_2), \ldots, v(O_n))$, where $v(O_1), \ldots, v(O_n)$ are output variable values. Note that since we consider only deterministic context-free plants, the output symbol is regarded as a *system state* since it contains values of all output variables.

Every variable in the model should be discrete. If the system has continuous variables, they can be discretized by splitting their allowed range into a number of contiguous intervals. For example, the real output variable $\mathtt{O} \in [0; 15]$ can be discretized as follows: $\mathtt{O}' \in \{[0; 5), [5; 10), [10; 15]\}$. Then, each continuous value of a variable can be replaced with the index of the interval it belongs to. Since discretization is case-dependent, it is assumed to be provided by the user.

Now note the two following observations. First, the maximum number of states of the resulting automaton is bounded from above by the number of unique output symbols. Second, if the system is in some state and all transitions from this state have been checked with MQs, there is no need to check them again. With these observations in mind, the plant model can be inferred using an active algorithm that resembles classical breadth-first search (BFS), which produces automata of the form shown in Fig. 2.

The core idea of the algorithm is to explore every transition labeled by every input symbol reachable from the chosen initial state. During the learning process only newly discovered states are added to the queue of next states to make transitions from, hence, if a state already exists in the model, it is not

TABLE I
TEMPORAL PROPERTIES FOR THE CYLINDER PLANT MODEL

Name	Temporal property	Comment	Correct value	Obtained value
φ_1	$\mathbf{G}(\mathtt{F} \rightarrow \mathbf{G}\,\mathbf{F}(\mathtt{R} = 0 \ \wedge \ \mathtt{L} = 0))$	If the system experiences failure, it stops working forever	+	+
φ_2	$\mathbf{G}(\mathtt{L} \ \wedge \ \mathtt{R} \ \wedge \ \neg\mathtt{Ext} \rightarrow \mathbf{X}(\mathtt{L}))$	When the cylinder is on the left side and the command is to retract, it will stay on the left	–	–
φ_3	$\mathbf{G}\,\mathbf{F}(\mathtt{R} \ \wedge \ \mathtt{L})$	Infinitely often the cylinder will appear in both sides simultaneously	–	–
φ_4	$\mathbf{G}(\mathtt{R} \ \wedge \ \mathtt{Ext} \rightarrow \mathbf{X}(\mathtt{F}))$	When the cylinder is on the right side and the command is to extend, the failure signal will be produced	+	+
φ_5	$\mathbf{G}(\mathtt{R} \ \wedge \ \neg\mathtt{Ext} \ \wedge \ \neg\mathtt{Retr} \rightarrow \mathbf{X}(\mathtt{R}))$	If the cylinder is on the right side and there are no commands to move anywhere, it does not move (the same for the left side)	+	+
φ_6	$\mathbf{G}(\mathtt{L} \ \wedge \ \mathtt{Ext} \ \wedge \ \neg\mathtt{Retr} \rightarrow \mathbf{X}(\mathtt{R}))$	If the cylinder is on the left side and the command is to extend, it will appear on the right	+	+

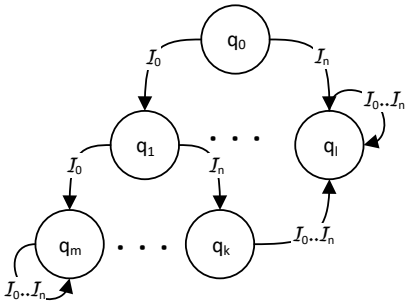


Fig. 2. The example form of the automaton that can be inferred using the proposed approach, where q_l, q_m, and q_k are distinct and unknown in advance states

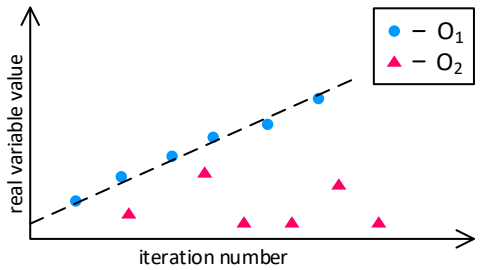


Fig. 3. Real variable $\mathtt{O}_1$ value monotonically increases, correlation coefficient > 0.5, whereas $\mathtt{O}_2$ value fluctuates, correlation coefficient < 0.5

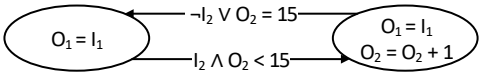


Fig. 4. Example system model with Boolean inputs $\mathtt{I}_1$ and $\mathtt{I}_2$, Boolean output $\mathtt{O}_1$ and continuous output $\mathtt{O}_2$

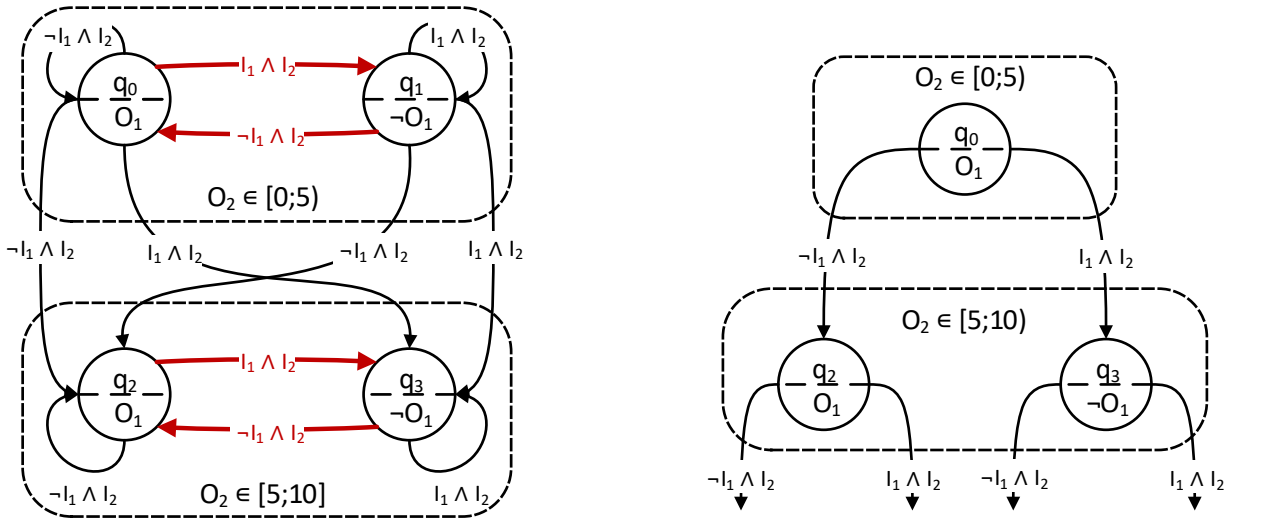
processed again. The algorithm terminates when the queue is empty and there are no new states to process.

The simple algorithm described above is sufficient in the rare case when the plant only uses Boolean variables. In practice, the plant almost always has continuous dynamics and is described with real variables, which makes formal modeling more complicated. Discretization of each real variable into a set of contiguous intervals makes the model discrete, however, as seen on the following example, the above algorithm will not produce a correct model.

Consider a system with one Boolean input variable $\mathtt{I}_1$ and one real output variable discretized with intervals $\mathtt{O}_1' \in \{[0;5), [5;15)\}$. The constructed model automaton will have two states, one for each value of $\mathtt{O}_1$ intervals. In the initial state $\mathtt{O}_1 \in [0;5)$; after a transition triggered by $\mathtt{I}_1 = \mathtt{True}$, the value of $\mathtt{O}_1$ is increased by 1. Following the logic described above, two transitions will be checked, one for $\mathtt{I}_1 = \mathtt{True}$ and one for $\mathtt{I}_1 = \mathtt{False}$. Both queries will result in self-loops in the model, no new states will be generated, and the algorithm will terminate leaving the state in which $\mathtt{O}_1 \in [5;15)$ undiscovered. Still, taking a closer look at the value of $\mathtt{O}_1$ after the first transition may indicate that the variable value is moving towards the next interval and the query simply needs to be repeated several times to reach it. The situation when after some transition a continuous variable changes its value but stays inside the current discretization interval will be called *movement inside an interval*, which means that the continuous variable value changes monotonically.

To detect "movement inside an interval" behavior the same MQ is sent to the system C times and the results are saved. After C repetitions, the Pearson correlation coefficient between the number of iterations and variable value is calculated. If the correlation coefficient is greater than 0.5, we can conclude that the value is changing almost monotonically – in this case we keep querying the system with the same MQ until the variable takes a value from the next interval. If the correlation coefficient is less than 0.5, then there is no change in the system and there is no sense in repeating a particular MQ (see Fig. 3).

However, in the situation when there are several discrete and discretized output variables in the system, even using additional logic for self-loops processing, the resulting model can be wrong – unnecessary cycles can be formed in the resulting model which will cause problems during verification. This problem is illustrated on an example system (Fig. 4) with two Boolean input variables $\mathtt{I}_1$ and $\mathtt{I}_2$ and two output



(a) A part of the model that contains cycles (marked bold red) in the first and in second intervals of variable O_2

(b) A part of the model where cycles from the first and second intervals of O_2 are removed

Fig. 5. Parts of the model before and after removing cycles

variables: Boolean variable O_1 and real variable $O_2 \in [0; 15]$ with discretization $O_2' \in \{[0; 5), [5; 10), [10; 15]\}$.

A part of the resulting automaton is shown in Fig. 5a. Here only transitions with $I_2 = \texttt{True}$ are shown to make the graph easier to read. Having such a model, false negative verification results are possible due to the cycle between q_0 and q_1. For example, the property "when O_2 is in the first interval and I_2 is always true, the system will eventually move to the state where O_2 is in the second interval" is false, while it is satisfied for the original system in Fig. 4.

To resolve this issue, the previous mechanism of detecting "movement inside intervals" situations was enhanced as follows. If during processing of a transition from q_n to q_k induced by some request (MQ) R a "movement inside an interval" situation is detected for some variable, state q_k is saved as a temporary state. Then, firstly, it is necessary to check whether the transition induced by request R from state q_k is a self-loop as it is described above. If so, then no other requests are queried from q_k and request R is repeated until the continuous variable reaches its next interval. Then q_k is registered as a new state. Otherwise, state q_k is marked as a state in which continuous variables will be compared by their concrete values, not by discrete intervals. If a state is marked this way, all states that will be generated after any transition from it will be marked as well if they stay in the same intervals of continuous variables.

Using the described strategy, the part of the model given in Fig. 5a can be redrawn as shown in Fig. 5b. The full pseudocode of the proposed plant model construction algorithm is given in Algorithm 1, where the aforementioned comparison of concrete continuous variable values is implemented via `enableConcreteComparison(state)` function (see line 32). After its execution it becomes impossible to limit the final number of states of the model from above. Also it is worth mentioning that if there are no discretized variables in the system, the loop starting at line 17 of the algorithm can be omitted.

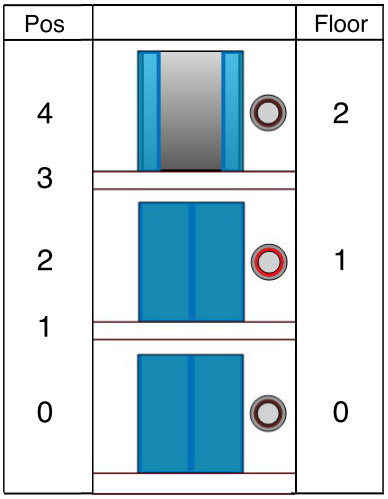


Fig. 6. Interface of the elevator model [7]

IV. Evaluation on a Case Study

The proposed approach was implemented in Java and is available as an open-source tool [14]. The approach was tested on the example of a three-floor elevator simulation model developed in NxtStudio [15] with user interface shown in Fig. 6. This model is almost identical to the one considered in [7] with a difference in buttons logic, which was moved to the controller. The aim of the experiments was to generate the plant model with the proposed method and compare verification results with [7].

The plant has the following Boolean variables:

- input: `moveUp`/`moveDown` – move the car up or down;

Algorithm 1: Proposed plant model synthesis algorithm

Data: set I of input variables, set O of output variables
Result: set of transitions $\mathcal{T}$

$Q_{proc} \leftarrow \varnothing$;
$Q_{next} \leftarrow$ `sendMQ(`$\varnothing$`)`;
$\mathcal{T} \leftarrow \varnothing$;
$\mathcal{I} \leftarrow \prod I_i$;
$\mathcal{O} \leftarrow \prod O_i$;
while $Q_{next} \neq \varnothing$ **do**
  $T \leftarrow \varnothing$;
  for $q \in Q_{next}$ **do**
    for $input \in \mathcal{I}$ **do**
      $(q_s, q_e, s) \leftarrow$ `sendMQ(`$q, input$`)`;
      $T \leftarrow T \cup \{(q_s, q_e, s)\}$;
    end
  end
  $T_{inInterval} \leftarrow \varnothing$;
  for $(q_s, q_e, s) \in T$ **do**
    if `movingInsideInterval(`q_s, q_e`)` **then**
      $T_{inInterval} \leftarrow T_{inInterval} \cup \{(q_s, q_e, s)\}$;
    end
  end
  for $(q_s, q_e, s) \in T_{inInterval}$ **do**
    $q'_e \leftarrow$ `sendMQ(`q_e, s`)`$.end$;
    $c \leftarrow$ `correlation(`q_s, q_e, s`)`;
    if $q'_e = q_e \ \wedge \ c > 0.5$ **then**
      while $q'_e = q_e$ **do**
        $q_e \leftarrow q'_e$;
        $q'_e \leftarrow$ `sendMQ(`q_e, s`)`$.end$;
      end
      $q_e \leftarrow q'_e$;
    else if $q'_e = q_e \ \wedge \ c < 0.5$ **then**
      $q_e \leftarrow q'_e$;
    else
      `enableConcreteComparison(`q_e`)`;
    end
  end
  $\mathcal{T} \leftarrow \mathcal{T} \cup T$;
  $Q_{proc} \leftarrow Q_{proc} \cup \{q_s \mid (q_s, q_e, s) \in T\}$;
  $Q_{next} \leftarrow \{q_e \mid (q_s, q_e, s) \in T, q_e \notin Q_{proc}\}$;
end
return $\mathcal{T}$

- input: `openDoors0..2` – open the doors at the respective floor;
- output: `carAtFloor0..2` – elevator car is at the respective floor;
- output: `doorClosed0..2` – the doors at the respective floor are completely closed.

Also there is a Real output variable `carPos` $\in [30; 419.5)$ discretized with intervals: [30; 30.5), [30.5; 224.5), [224.5; 225.5), [225.5; 418.5), [418.5; 419.5). The `carPos` output variable represents the concrete position of the car, its initial value is 30 (interval [30; 30.5)) that corresponds to the second floor. After each `moveDown`/`moveUp` command the position value increases/decreases by 1, respectively.

Plant model inputs correspond to controller outputs, controller inputs include all plant model outputs except `carPos`, and also include additional Boolean inputs `buttonPressed0..2` – whether the "request elevator" button is pressed at the respective floor. After the car reaches the floor where the button is pressed, the `buttonPressedN` value is set to `False`. To test the proposed algorithm, a special functionality of setting the plant to an arbitrarily chosen state was added to the simulation model.

The proposed algorithm generated the plant model in the form of an automaton with 40 states, which required a total of 135 seconds. Most of this time was spent on processing self-loops and cycles. The generated model was exported in the NuSMV format and composed with the manually prepared controller model to perform closed-loop verification. LTL properties that were verified are the same as the ones checked in [7] and are listed in Table II. All verification results are correct and no additional changes were applied to the resulting formal model.

V. Discussion and conclusion

Active learning approaches to plant model generation are quite perspective since their core idea is to refine the hypothesis model on every iteration and gather only those behavior examples that are required to build a consistent model. In this work a method for generating formal context-free deterministic plant models in a black-box scenario was introduced. It does not require an oracle or equivalence checks between source and hypothesis automata and generates reliable models.

On the other hand, note that model construction starts in some initial state, which means that the algorithm only discovers states reachable from the initial one. Thus, if it is guaranteed that the entire model state space is reachable from the chosen initial state, then the generated formal model for context-free deterministic plant will be reliable. Otherwise, there is a risk to discover only a sub-automaton or, in case of a disconnected state space, to fail to detect some of its parts. Meanwhile, the solution is quite straightforward if the set of possible initial states is known in advance – running the algorithm from each initial state and merging resulting automata will solve the issue. However, if the plant simulation model can be initialized in every possible state, all such states should be explored.

Also it should be noted that the suggested approach relies in its efficiency on the possibility to quickly reset the simulation model to the required state. If for some system this is impossible, at least a reset to the initial state must be available: in this case the method will still work, but each reset will take some time, depending on the simulation model implementation.

Another thing to mention is the time complexity of the algorithm. Since the proposed algorithm resembles BFS, all transitions must be executed for all input symbols from every new state, and the number of transitions will grow exponentially with the number of plant input variables. But, in fact,

TABLE II
ELEVATOR SYSTEM TEMPORAL PROPERTIES VERIFICATION RESULTS

	Temporal property	Comment	Correct value	Obtained value
	Plant temporal properties			
φ_1	$\mathbf{G}(\texttt{carAtFloor1} \wedge \mathbf{G}\ \neg\texttt{motorUp} \wedge \mathbf{G}(\texttt{motorDown} \vee \texttt{carAtFloor0}) \rightarrow \mathbf{G}\ \neg\texttt{carAtFloor2})$	If the car is on the first floor and never moves up and always moves down or stays on floor 0, it will never reach floor 2	+	+
φ_2	$\mathbf{G}(\mathbf{G}\ \neg\texttt{motorUp} \wedge \mathbf{G}(\texttt{motorDown} \vee \texttt{carAtFloor0}) \rightarrow \mathbf{F}\,\texttt{carAtFloor0})$	With similar conditions the car will reach floor 0	+	+
φ_3	$\mathbf{G}(\mathbf{G}\,\neg\texttt{motorDown} \wedge \mathbf{G}(\texttt{motorUp} \vee \texttt{carAtFloor2}) \rightarrow \mathbf{F}\,\texttt{carAtFloor2})$	Analogously, if we the car moves up, it will reach floor 2	+	+
φ_4	$\mathbf{G}\,\mathbf{F}\ \neg\texttt{motorDown}$	Controller cannot always send the "Down" command	−	−
φ_5	$\mathbf{G}(\texttt{carAtFloor1} \wedge \mathbf{G}\ \neg\texttt{motorUp} \wedge \mathbf{G}\,\texttt{motorDown} \rightarrow \mathbf{F}\,\texttt{carAtFloor2})$	Similar to condition 1, but $\mathbf{G}\,\texttt{motorDown}$ is used instead of $\mathbf{G}(\texttt{motorDown} \vee \texttt{carAtFloor0})$	−	−
φ_7	$\mathbf{G}(\texttt{carPos} = 4 \wedge \texttt{motorDown} \wedge \neg\texttt{motorUp} \rightarrow \mathbf{X}\,\texttt{carPos} = 3)$	If the car is on floor 2 and moves down, it will be between floors 1 and 2	+	+
φ_9	$\mathbf{G}(\texttt{carPos} = 2 \wedge \neg\texttt{motorDown} \wedge \texttt{motorUp} \rightarrow \mathbf{X}\,\texttt{carPos} = 3)$	Similarly, floor 1, motorUp	+	+
	Closed-loop model checking			
φ_{11}	$\forall k \in [0..2]\ \mathbf{G}(\texttt{buttonPressed}_\texttt{k} \rightarrow \mathbf{F}\,\texttt{carAtFloor}_\texttt{k})$	If the car is called, it will arrive to the specified floor	−	−
φ_{12}	$\mathbf{G}(\texttt{buttonPressed2} \wedge$ (not always at some floor) $\rightarrow \mathbf{F}\,\texttt{carAtFloor2})$	If the car is called to floor 2 and is not stuck at some floor it will arrive to floor 2	+	+
φ_{14}	$\mathbf{G}(\texttt{buttonPressed0} \wedge$ (not always at some floor) $\rightarrow \mathbf{F}\,\texttt{carAtFloor0})$	The same for floor 0. Because of controller choice the result differs	−	−
φ_{15}	$\mathbf{G}(\texttt{carPos} \in \{1,3\} \rightarrow \texttt{doorClosed0} \wedge \texttt{doorClosed1} \wedge \texttt{doorClosed2})$	When the car is between floors, all doors are closed	+	+

not every transition ends in a new state. Commonly, some rule exists in input variable changes, therefore, detection of such rules makes it possible not to check all input symbols. Hence, future work will tackle reduction of the number of transitions to be checked during model construction. Another direction of future work is automating discretization intervals construction, which could be done on the basis of inferred model properties analysis and temporal properties verification results.

ACKNOWLEDGEMENTS

This work was supported 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] S. Preusse, *Technologies for Engineering Manufacturing Systems Control in Closed Loop*. Logos Verlag Berlin GmbH, 2013, vol. 10.

[3] V. Vyatkin, H.-M. Hanisch, C. Pang, and C.-H. Yang, "Closed-loop modeling in future automation system engineering and validation," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 39, no. 1, pp. 17–28, 2009.

[4] A. Maier, "Online passive learning of timed automata for cyber-physical production systems," in *12th IEEE International Conference on Industrial Informatics*, 2014, pp. 60–66.

[5] 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, 2017.

[6] G. Giantamidis and S. Tripakis, "Learning moore machines from input-output traces," in *FM 2016: Formal Methods*. Cham: Springer International Publishing, 2016, pp. 291–309.

[7] D. Avdyukhin, D. Chivilikhin, G. Korneev, V. Ulyantsev, and A. Shalyto, "Plant trace generation for formal plant model inference: Methods and case study," in *IEEE 15th International Conference on Industrial Informatics*, 2017, pp. 746–752.

[8] C. de la Higuera, *Grammatical Inference. Learning automata and grammars*. Cambridge University Press, 2010.

[9] D. Angluin, "Queries and concept learning," *Machine Learning*, vol. 2, no. 4, pp. 319–342, 1988.

[10] B. Steffen, F. Howar, and M. Merten, "Introduction to automata learning from a practical perspective," *Formal Methods for Eternal Networked Software Systems*, pp. 256–296, 2011.

[11] R. Rivest and R. Schapire, "Inference of finite automata using homing sequences," *Information and Computation*, vol. 103, pp. 299–347, 1993.

[12] A. Nerode, "Linear automaton transformations," *Proceedings of the American Mathematical Society*, vol. 9, no. 4, pp. 541–544, 1958.

[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. International Conference on Computer-Aided Verification*, ser. LNCS, vol. 2404. Copenhagen, Denmark: Springer, 2002.

[14] Proposed algorithm implementation. [Online]. Available: https://github.com/ShakeAnApple/active-learning

[15] NxtControl. [Online]. Available: http://www.nxtcontrol.com