Plant trace generation for formal plant model inference: methods and case study

Dmitry Avdyukhin, Daniil Chivilikhin, Georgiy Korneev, Vladimir Ulyantsev and Anatoly Shalyto
Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia
Email: {chivdan, ulyantsev}@rain.ifmo.ru, shalyto@mail.ifmo.ru

***Abstract*—Cyber-physical system correctness can be ensured by employing formal methods, including model inference. A rather new direction of research is using formal plant models for closed-loop verification and inferring these models from execution traces of the system. Since the quality of the resulting model heavily depends on trace coverage, in this paper we propose efficient methods for automatic plant trace generation. A method of checking the conformance between system and generated models is suggested and is used to analyze traces generation and model inference methods. Modifications of plant model inference approaches are proposed and comparison of the resulting models is performed.**

I. Introduction

Cyber-physical systems (CPS) are comprised of physical and software components interacting with each other. In many domains such as nuclear power plant engineering and aerospace systems their correct behavior is a critical requirement and its violation can result into grave consequences. To facilitate correctness of CPS such techniques as testing and formal verification are used. Whereas testing can only discover mistakes, verification, and in particular model checking [1], allows proving system correctness rigorously. However, verification requires the availability of the formal CPS model, which in practice is rarely developed and, even if created, is rarely maintained in actual state. This is due to the fact that development of formal CPS models and their support is associated with considerable effort and time costs.

Therefore, an alternative to manual creation of formal CPS models is required. Automated formal CPS model inference algorithm can use the real CPS itself for deriving its model. However, since a simulation model is often available during system development, it can be beneficial to use it instead. Hence, as an alternative to manual model creation, the problem of its generation using the original system or simulation model arises.

A CPS is usually comprised of two interconnected components: the *plant*, where all physical processes occur, and the *controller*, which receives information from the plant using various sensors and sends commands back to the plant. Often there is also human-machine interaction, but it can be considered as a part of the plant or the controller.

There are two main approaches to CPS verification: *open-loop* and *closed-loop* verification. Open-loop verification is applied to the controller model alone. In this case no assumptions about plant behavior are made, as if sensors can assume arbitrary values. This approach has several downsides, among which the *state explosion* problem and impossibility to prove some properties can be highlighted. It gives rise to closed-loop verification, which takes into account both plant and controller models and allows checking a wider class of properties. To apply it, a *formal plant model* is needed.

While there are effective methods of controller model generation [2], [3], existing approaches for plant model generation have certain limitations. They either impose limitations to development process and simulation model [4], are limited to certain class of systems [5], or require additional information about system [6]. As a result, existing plant model inference methods do not allow building a plant model for an arbitrary system automatically.

One of possible solutions is automatic model inference based on *execution traces* – records of system runs, which can be either deliberately generated for the purpose of model generation or can be available as log data. Methods proposed in works [7], [8] allow building a formal model of the plant automatically using execution traces. Correctness of models inferred by this method strongly depends on the traces. Informally, it must be possible to derive a correct plant model based on the given traces.

In this paper we propose three trace generation approaches for plant model inference. One of the proposed approaches called "semirandom" trace generation was shown to produce traces that are sufficient for inferring adequate plant models. Model adequacy is assessed using verification of temporal constraints and checking that the model conforms to the traces. To overcome discovered flaws in one of the model inference algorithms, its modifications are proposed.

II. Plant model inference and trace generation

Methods studied in this paper solve the problem of plant model inference based on execution traces. We consider a system consisting of a *plant* and a *controller* which interact in a cyclic way: on each cycle the controller first reads data from plant sensors (plant *outputs*) and then sends commands (*inputs*) to the plant.

Denote a set of all possible combinations of outputs as $v(O)$ and a set of all possible combinations of inputs as $v(I)$. An *execution trace* is defined as a sequence of the form $(O_1, I_1),\ (O_2, I_2), \ldots,\ (O_n, I_n)$, where $\forall k \in [1..n]\ O_k \in v(O)$ is a combination of outputs and $I_k \in v(I)$ is a combination of inputs. Internal variables also can and often need to be considered as outputs, assuming their logging is

978-1-5386-0836-4/17/$31.00 ©2017 IEEE

possible. This way the plant is considered as a black box, which means that only its external behavior is taken into consideration.

The resulting plant model must conform to the following requirements. First, the model must accept all traces it was generated from – for each trace $(O_1, I_1), \ldots, (O_n, I_n)$, if the model receives the sequence of input combinations $I_1, \ldots, I_n$, it must be able to produce the corresponding sequence of output combinations $O_1, \ldots, O_n$. Second, the model should not have *deadlocks*, which means that it has to accept any possible input in any state. Third, an *adequacy* requirement should be satisfied, which states that the model should have a behavior similar to the one of the original system. This requirement is formalized by means of verifying temporal specifications describing the behavior of the original system and also checking if the model can accept traces generated by different methods (this topic will be covered in more details below). Intuitively, verification of temporal constraints, Linear Temporal Logic (LTL) [1] constraints in particular, allows checking whether the generated model is specific enough and does not allow untypical behavior. On the other hand, acceptance of different traces allows checking that the model is general enough to support a behavior not covered in traces it was built from.

Since an adequate model cannot be inferred from incomplete traces, the stated requirements on the plant models consequently result into requirements for the traces and a problem of generating "good" traces arises. In this paper we propose trace generation methods that produce traces sufficient for inferring adequate formal plant models.

III. Methods under study

We focus on plant model inference methods that produce models in the form of nondeterministic Moore machine [9] with transitions labeled with elements from $v(I)$. The methods establish an injection from the set of automaton states to the set of all possible output combinations. Therefore, a unique element in $v(O)$ denoted as $o(s)$ corresponds to each state s. This implies the following constraints. First, $v(O)$ has to completely specify and distinguish all states of the system. If two essentially different system states share the same element from $v(O)$, the resulting model will be overgeneralized. Second, since the number of model states must be finite, $v(O)$ must also be finite, therefore a *discretization* of continuous outputs is required, which is a finite partition of an infinite or large set. For example, if an object can have a position from the interval $[L; R]$, this set can be discretized into three subsets: $\{L\}$, $\{R\}$, and $(L; R)$, denoting left, right, and intermediate positions respectively.

We consider two recently proposed approaches for plant model inference: explicit-state and constraint-based [7], [8], [10] creating models in the form of NuSMV [11] modules. Discretization of each continuous output is implemented as follows: its range of values is splitted into k intervals, and each continuous value in the traces is substituted by the index of the interval it belongs to.

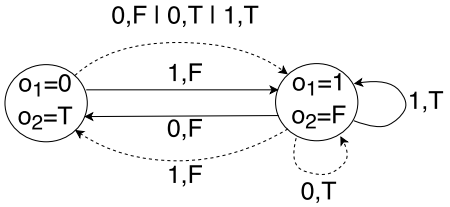

Fig. 1. Explicit-state automaton built from the example trace (*)

Consider as an example a system with two outputs $o_1 \in [0; 30]$, $o_2 \in \{T, F\}$ and two inputs $i_1 \in [-100; 100]$, $i_2 \in \{T, F\}$ with the following discretization:

$$o_1 \in [0; 0] \cup (0; 30], i_1 \in [-100; 0] \cup (0; 100].$$

Consider the following execution trace, where each element is in format (o_1, o_2, i_1, i_2):

$$(0, T,\ 10, F), (10, F,\ 20, T), (30, F,\ -30, F), (0, T,\ 0, T).$$

After discretization it becomes as follows:

$$(0, T,\ 1, F), (1, F,\ 1, T), (1, F,\ 0, F), (0, T,\ 0, T). \quad (*)$$

A. Explicit-state approach

The *explicit-state* approach generates an automaton with a set of states S where $s \in S$ iff $o(s)$ (the corresponding combination of outputs) is encountered in execution traces. The model contains a transition from state s_1 to state s_2 with label l iff there is a trace in which $O_k = o(s_1)$, $I_k = l$ and $O_{k+1} = o(s_2)$. A state s is the initial state iff there is a trace starting with $o(s)$. An automaton built from the discretized trace (*) is shown in Fig. 1.

To prevent deadlocks, *unsupported* transitions are introduced. An unsupported transition leads to the same state as a transition which is closest to it with respect to the amount of different inputs and starts in the same state. Ambiguities are resolved arbitrarily. In Fig. 1 unsupported transitions are denoted by dotted lines.

The main disadvantage of the explicit-state approach is that it is capable of generating models only with states which are present in some trace from the training set. This restricts the predictive ability of the model – to build an adequate model, each state and each transition have to be covered at least once. This is an unrealistic requirement for large real-world systems due to the state explosion problem.

B. Constraint-based approach

The *constraint-based* approach proposes the following generalization of traces to deal with the state explosion problem. For each input i_k and each output o_k a corresponding variable is created, and the following constraints on evolution of each pair of variables are imposed (below we present the generated NuSMV constraints for the example *).

Each pair of output variables o_1 and o_2 can simultaneously have values v_1 and v_2 iff there exists a trace element in which the value of o_1 is v_1 and the value of o_2 is v_2:

```
o1=0 & o2=T  |  o1=1 & o2=F.
```

An output variable o_k can have consequent values x and y iff it has these values in adjacent trace elements:

```
(o1=0 -> next(o1)=1) &
(o1=1 -> next(o1) in {0,1}) &
(o2=T -> next(o2)=F) &
(o2=F -> next(o2) in {T,F}).
```

Output variable o_j can take value x immediately after input variable i_k takes value y iff there are two consequent trace elements for which this situation occurs:

```
(i1=0 -> next(o1)=0 & next(o2)=T) &
(i1=1 -> next(o1)=1 & next(o2)=F) &
(i2=T -> next(o1)=1 & next(o2)=F) &
(i2=F -> next(o1) in {0,1}
         & next(o2) in {T,F})
```

These constraints are specified in a symbolic form, which avoids explicit state representation and allows symbolic verification. It is possible to use constraints for more than two variables, but it will lead to significant increase of their number – for n variables the number of constraints on k variables can reach $\binom{n}{k}$.

Obtained models can be modified in various ways, amongst which the introduction of the *changeability constraint* can be especially emphasized. Self-loops are common in generated models and arise from the situation when some continuous input or output changes, but its discretized value does not. In this case verification can conclude that system can be "stuck" in such a state, which is not the case in the original system. The changeability constraint has been proposed as a solution for this eternal self-loops problem. It is specified in the form of a fairness constraint – condition which should be true infinitely often on any infinite path:

$$\mathbf{G}\,\mathbf{F} \neg \bigwedge_{k=1}^{m} (o_k = \mathbf{next}(o_k)).$$

The changeability constraint implies that at least one output must eventually change.

An important question is the adequacy of models generated by described methods. In this paper we try to address it by performing a case study of described methods on the example of the simulation model of the elevator control system, described in the next section.

IV. Case study: elevator model

As a case study for this paper we consider an elevator moving between three floors. Its simulation model is implemented using IEC 61499 [12] function blocks in the NxtStudio [13] environment. Its user interface is shown in Fig. 2. Onwards, the simulation model is called just "system". The system consists of a controller, transmitting commands to open/close doors and move the elevator up or down, and a plant, handling specified commands and responding to button clicks and door opening. The plant has the following Boolean input variables.

- `Up`, `Down` – move the car up or down respectively. In case when both variables are true, the car does not move.
- `OpenDoor0..2` – open a door at the respective floor. In case when the variable is false and the door is opened, the door closes.

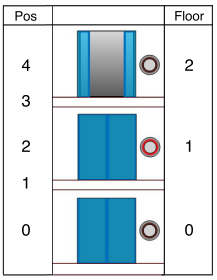

Fig. 2. Left: discretized positions. Middle: elevator user interface. Right: floor numeration

As outputs, real value `Position` (of the car) and the following Boolean variables are used:

- `Button0..2` – whether the button on the corresponding floor is pressed.
- `Floor0..2` – whether the car is on the corresponding floor.
- `DoorClosed0..2` – whether the door on the corresponding floor is fully closed.

In case when a door is fully opened and the car is on the same floor, the corresponding button turns off. Buttons are activated by the user who presses them.

The controller acts as follows. The car can change its direction only when it is located at a floor. When some button becomes active, the car moves to the corresponding floor and then the door starts to open. When the door is fully opened, the button turns off. An ambiguity arises when the car is on the first floor and buttons on floors 0 and 2 are active; in this case the floor 0 is chosen.

In order to apply the described model inference methods, only the continuous parameter `Position` of the system should be discretized. This is implemented in a natural way: for each floor and each position between floors there is a separate discrete value of variable `Pos`. Value 0 corresponds to floor 0, value 4 – to floor 2, as shown in Fig. 2.

In order to apply the described plant model inference methods, a few modifications of the system were performed. First, it was necessary to create conditions for fully automatic trace recording. For this purpose buttons were modified in a way that allows any of them to become active with probability $\frac{1}{3}$ while the car moves between adjacent floors. This way a random behavior of the elevator user(s) is modeled, allowing coverage of the large number of different system states.

Second, we needed to provide a way to record traces of plant behavior. For this purpose, a TCP connection with an external program receiving all data between the controller and the plant was created. This approach allows not only data recording, but also its modification in any way. For example, we can substitute the controller (or the plant) with a program constituting a different behavior. As a result, the plant can be tested even if the controller is not implemented. The described architecture is depicted in Fig. 3. This potential is actively utilized in the present work, because it allows using different controllers and, therefore, studying different ways of trace generation.

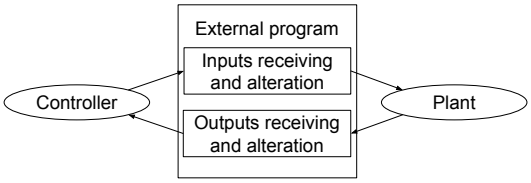

Fig. 3. System architecture after modification

V. Proposed trace generation methods

A natural way to generate traces is to use the *original controller*. Since system is restricted to "normal" behavior, the plant cannot visit some of its states, e.g. two doors cannot be opened simultaneously. This can greatly reduce the number of states and therefore speed up verification. On the other hand, the behavior of the model in unusual (for example, emergency) situations can be unpredicted or even undefined. While plant outputs are not altered by the external program, the possibility to modify the controller commands is used. In the present work the following ways of trace generation are proposed and analyzed.

1) *Random controller*. On each cycle controller commands are generated randomly.
2) *"Semirandom" controller*. Inputs are first generated randomly, but then do not change until either Ccycles passes or some output changes. This approach allows visiting relatively rare states: with the random controller it is unlikely to visit some floor with closed door, because there are hundreds of possible positions for both the door and the car, and only one combination corresponds to the described situation. With the semirandom controller this becomes possible, because after visiting the floor the outputs change, and there is a chance that there will be a command to close the door and stay on the floor, leading after a few cycles to the described state.
3) *Uniform inputs coverage*. This is a modification of the semirandom approach. The probability that each input will take a certain value is influenced by the frequency with which this value appears in traces: if values $v_1, \ldots, v_k$ occurred in traces $c_1, \ldots, c_k$ times respectively, then the probability to take value v_i is inversely proportional to c_i.

Models generated with the described methods are onwards called "original model", "random model", "semirandom model", and "uniform model" respectively. They are relatively simple and can potentially be applied to any system, without any knowledge of its internal structure or behavior.

VI. Plant and system model generation

In this section, the process of building formal models of the plant and the entire system is described.

A. Model building and test set generation

The recorded traces are partitioned into two sets: the training set used for model generation and the test set used to evaluate model quality. Each simulation results into one trace with length $2 \cdot 10^5$ elements where each element corresponds to outputs and inputs on one cycle. The first 90% of entries form the training set, and the last 10% are splitted into small traces used to check that the model conforms to the traces.

Here and onwards we will say that a model *conforms* to a trace if there is a state in the model starting from which there is a path equivalent to the given trace. The path can start in any reachable state, not only the initial one. For each trace from the test set a Computation Tree Logic (CTL) [1] *conformance* property was generated, satisfaction of which is equivalent to model conformance to the trace. The property has the form:

$$\mathbf{EF}(O_1 \wedge I_1 \wedge \mathbf{EX}(O_2 \wedge I_2 \wedge \mathbf{EX}(O_3 \wedge \ldots))).$$

It means that there is a path ($\mathbf{EF}$) from the initial state to a state in which the outputs equal to O_1 and from which there is a transition with label I_1 into a state for which the outputs equal to O_2 and so on. Conformance properties are checked using NuSMV.

Since states and output combinations are equivalent in generated models, traces of form $(O_k, I_k),\ (O_{k+1}, ...)$, checking each transition separately, are sufficient. But the large number of such conditions leads to increased verification time and memory consumption, so it is required to consider longer traces. As a compromise, traces of length 8 were chosen as a rather small value with the best performance results.

B. Closed-loop model checking

Two sets of temporal properties for the considered elevator system were prepared: plant-only and closed-loop properties. Plant-only properties correspond to the situation when we try to check model behavior considering specific input commands, which can have no relation with the ones of the original controller. Closed-loop properties assume that the model acts in presence of the original controller, and so they allow checking the behavior of the entire system. In order to check them we need to build a closed-loop model, and for that purpose a controller model was built.

VII. Evaluation of inferred models and trace generation methods

The main evaluation criterion of generated models is their adequacy in terms of temporal properties, trace conformance and their size. No notable difference was found between semirandom and uniform models with the same value of C. This is probably due to the fact that the random method ensures sufficiently uniform input coverage. Therefore the uniform method will not be considered further.

First we consider an LTL property which shows a major issue of the constraint-based approach:

$$\mathbf{G}(\texttt{Pos} = 2 \ \wedge \ \texttt{Down} \rightarrow \mathbf{X}(\texttt{Pos} = 1)).$$

This property is violated because generated constraints are weak: generated constraints impose that $\texttt{Pos} = 2$ is followed by $\texttt{Pos} \in [1..3]$ and `Down` is followed by $\texttt{Pos} \in [0..3]$. However, this is not enough to prove that the car moves down after the `Down` command is issued, it can only be

TABLE I

RESULTS OF TEMPORAL PROPERTIES VERIFICATION: "+" CORRESPONDS TO VERIFIED FORMULAS, "–" CORRESPONDS TO FALSIFIED ONES. LABEL "(C)" DENOTES RESULTS OF VERIFICATION WHICH ARE AS GIVEN IN CASE OF ACTIVE CHANGEABILITY CONSTRAINT AND ARE OPPOSITE OTHERWISE.

Name	Temporal property	Comment	Correct value	Original constraint	Original explicit with unsupported	Original explicit	(Semi)random constraint	(Semi)random explicit	(Semi)random constraint (modified)
		Plant-only model checking							
φ_1	$\mathbf{G}(\texttt{Floor1} \wedge \mathbf{G}\ \neg\texttt{Up} \wedge \mathbf{G}(\texttt{Down} \vee \texttt{Floor0}) \rightarrow \mathbf{G}\,\neg\texttt{Floor2})$	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	$\mathbf{G}(\mathbf{G}\,\neg\texttt{Up} \wedge \mathbf{G}(\texttt{Down} \vee \texttt{Floor0}) \rightarrow \mathbf{F}\,\texttt{Floor0})$	With similar conditions the car will reach floor 0	+	+(C)	–	+(C)	–	–	+
φ_3	$\mathbf{G}(\mathbf{G}\,\neg\texttt{Down} \wedge \mathbf{G}(\texttt{Up} \vee \texttt{Floor2}) \rightarrow \mathbf{F}\,\texttt{Floor2})$	Analogously, if we the car moves up, it will reach floor 2	+	+(C)	–	+(C)	–	–	+
φ_4	$\mathbf{G}\,\mathbf{F}\,\neg\texttt{Down}$	Controller cannot always send the "Down" command	–	+(C)	–	+(C)	–	–	–
φ_5	$\mathbf{G}(\texttt{Floor1} \wedge \mathbf{G}\ \neg\texttt{Up} \wedge \mathbf{G}\,\texttt{Down} \rightarrow \mathbf{F}\,\texttt{Floor2})$	Similar to condition 1, but $\mathbf{G}\,\texttt{Down}$ is used instead of $\mathbf{G}(\texttt{Down} \vee \texttt{Floor0})$	–	+(C)	–	+(C)	–	–	–
φ_6	$\mathbf{G}\,\neg(\texttt{Down} \wedge \texttt{Up})$	Commands to move up and down are never issued simultaneously	–	+	–	–	–	–	–
φ_7	$\mathbf{G}(\texttt{Pos} = 4 \wedge \texttt{Down} \wedge \neg\texttt{Up} \rightarrow \mathbf{X}\,\texttt{Pos} = 3)$	If the car stays on floor 2 and moves down, it will be between floors 1 and 2	+	+	–	–	–	–	+
φ_8	$\mathbf{G}(\texttt{Pos} = 2 \wedge \texttt{Down} \wedge \neg\texttt{Up} \rightarrow \mathbf{X}\,\texttt{Pos} = 1)$	Similarly, floor 1, Down	+	+	–	+	–	+	+
φ_9	$\mathbf{G}(\texttt{Pos} = 2 \wedge \neg\texttt{Down} \wedge \texttt{Up} \rightarrow \mathbf{X}\,\texttt{Pos} = 3)$	Similarly, floor 1, Up	+	+	–	+	–	+	+
φ_{10}	$\mathbf{G}(\texttt{Pos} = 0 \wedge \neg\texttt{Down} \wedge \texttt{Up} \rightarrow \mathbf{X}\,\texttt{Pos} = 1)$	Similarly, floor 0, Up	+	+	–	+	–	+	+
		Closed-loop model checking							
φ_{11}	$\forall k \in [0..2]\ \mathbf{G}(\texttt{Button}_\texttt{k} \rightarrow \mathbf{F}\,\texttt{Floor}_\texttt{k})$	If the car is called, it will arrive at the specified floor	–	–	–	–	–	+	–
φ_{12}	$\mathbf{G}(\texttt{Button2} \wedge (\text{not always at some floor}) \rightarrow \mathbf{F}\,\texttt{Floor2})$	If the car is called to floor 2 and is not stuck at some floor it will arrive at floor 2	+	+(C)	+	+(C)	–	+	+
φ_{13}	$\mathbf{G}(\texttt{Button1} \wedge (\text{not always at some floor}) \rightarrow \mathbf{F}\,\texttt{Floor1})$	The same for floor 1	+	+(C)	+	+(C)	–	+	+
φ_{14}	$\mathbf{G}(\texttt{Button0} \wedge (\text{not always at some floor}) \rightarrow \mathbf{F}\,\texttt{Floor0})$	The same for floor 0. Because of controller choice the result differs	–	–	–	–	–	+	–
φ_{15}	$\mathbf{G}(\texttt{Pos} \in \{1,3\} \rightarrow \texttt{DoorClosed0} \wedge \texttt{DoorClosed1} \wedge \texttt{DoorClosed2})$	When the car is between floors, all doors are closed	+	+	+	+	–	+	+

concluded that $\texttt{Pos} \in [1..3]$. This issue arises since the next output value often depends on both the input and the current output value – this is not captured in generated constraints. It is possible to solve this problem by considering triples of values instead of pairs, but, as said before, this would lead to a significant increase of model size. An alternative solution employed in this paper is to use additional constraints $O_i \wedge I_j \rightarrow \mathbf{next}(O_i)$. As a result, the new value of the output depends both on the previous input and output.

Verification results for other properties are shown in Table I. It can be concluded from the table that the explicit model with unsupported transitions is not adequate, which can be due to the somewhat arbitrary choice of unsupported transitions. Therefore, unsupported transitions were excluded from the following analysis. Other observations are considered in detail below.

A. Variable grouping

Consider verification of properties φ_7–φ_{10} for the random and semirandom constraint-based models. Commands Up and Down together can be viewed as a variable indicating the direction of movement. Recall that only interactions between pairs of variables, such as $(\texttt{Up}, \texttt{Pos})$ and $(\texttt{Down}, \texttt{Pos})$, are constrained in the model, ant therefore triples of variables $(\texttt{Down}, \texttt{Up}, \texttt{Pos})$ are not affected. As a result it only can be concluded that if $\texttt{Pos} = 2 \wedge \texttt{Down} \wedge \neg\texttt{Up}$, then $\texttt{Pos} \in \{1, 2\}$, because of the following constraints:

- $\texttt{Pos} = 2 \wedge \texttt{Down} \rightarrow \mathbf{next}(\texttt{Pos} \in \{1, 2\})$;
- $\texttt{Pos} = 2 \wedge \neg\texttt{Up} \rightarrow \mathbf{next}(\texttt{Pos} \in \{1, 2\})$.

This discrepancy between the plant and plant model behavior can be solved by considering Down and Up together, which means to consider all their combinations paired with other variables: $\texttt{Pos} = 2 \wedge (\texttt{Down} \wedge \neg\texttt{Up}) \rightarrow \mathbf{next}(\texttt{Pos} = 1)$.

Thus, the algorithm can be modified in the following way: give the user an opportunity to manually group variables which are needed to be considered together. This approach will not lead to a large increase of the number of constraints, because only a limited number of variables are grouped. Also it is not a laborious task, because only trivially connected variables such as `Up` and `Down` should be grouped.

The described problem does not arise in the original model because in the original controller commands `Down` and `Up` are mutually exclusive. However, as a result, the resulting model can not receive arbitrary inputs, as shown by property φ_6, because this combination of inputs was never encountered in traces.

B. Changeability constraint problem

Property φ_4 ($\mathbf{G}\,\mathbf{F}\,\neg$`Down`) is satisfied in both explicit-state and constraint-based original models when the changeability constraint is enabled, which means that there is a possibility of deadlocks – some input sequences are prohibited. Aside from deadlocks, the situation when outputs do not change is prohibited in the model, but can be acceptable by the system. Moreover, in the case of random and semirandom models the changeability constraint does not solve its original task – the car can still be stuck between floors if other outputs change.

An alternative solution for eternal self-loops problem, applied to the elevator system, is to add fairness constraints of the form $\neg($`Up` $\wedge$ $\neg$`Down` $\wedge$ `Pos` $= 3)$ – if the car moves in the same direction between floors, it must eventually reach the end point. If these constraints and input grouping are applied, verification results of all properties for random and semirandom models are correct. The downside of the approach is that no general algorithm is known to generate such constraints automatically. On the other hand, there is a common situation when an output representing a position of a moving object is controlled by only one (possibly grouped) parameter, representing movement direction. For example, in the considered model `Pos` is controlled by grouped parameter (`Up`, `Down`). In such cases it is often possible to prove properties of the form "if an object moves from one end of a segment to another, it will eventually reach the latter".

One possible approach is to discover monotonicity: if some input takes certain value, then the output value, if it is not the maximum one, is always increased in the following trace element. For example, `Up=True` $\wedge$ `Down=False` results into `Position` increase. The same holds for output value decrease. A drawback of this approach is the necessity of continuous output value, which is not always available.

An alternative solution is to consider even more special case: when the object can move only with a constant speed, so it can take only values $-c,\ 0,\ c$. In this case, during movement from one end of a segment to the other one, the difference between numbers of c and $-c$ must be equal. Thus, we can iterate through all pairs of values (assuming that one of them is c and the other one is $-c$) and check whether this condition holds for them.

In the case of the elevator system, both approaches allow obtaining the required constraints. Also it should be noted that this constraints could not be derived without variable grouping.

C. Trace conformance

All models built in experiments conform to all traces they were generated from. Table II presents results of conformance of different inferred models to test sets, obtained with different trace generation methods. It can be seen that the explicit-state models never totally conform to traces, because not every logical state is reached, and the model becomes undergeneralized. While the constraint-based models conform to all traces from the training set they were built from, the explicit-state models conform to a vast majority of traces, but, because of undergeneralization, not to all of them.

As expected, the original model does not conform to traces generated by the other methods, because they often reach states that are unreachable in the original model. It can be seen that constraint-based semirandom models become more general with the increase of C, as they conform to semirandom traces with smaller C as well as original traces (random traces can be considered as semirandom(1)).

D. Performance: model sizes and verification time

Metrics of constraint-based models do not vary notably across different trace generation method. Their sizes are equal to 44 ± 1 KB and trace verification requires 34 ± 1 seconds – such a tight distribution is not surprising, since their sizes are mainly determined by the number of variables, which is fixed.

On the other hand, the metrics of explicit-state models vary greatly and are presented in Table III It can be seen that verification time increases with the growth of C, which can be explained by the increase of the number of states visited during simulation. This can indicate that for large-scale systems with a big set of states the explicit-state approach is not acceptable, emphasizing the advantage of the constraint-based approach.

VIII. Discussion

The main issue of the applied solutions is that they often require manual manipulation, whereas the changeability constraint proposed in the original method has an advantage of being simple to generate. The positive side is that modifications are relatively simple and the number of modifications is at most linear of the number of variables.

The resulting constraint-based model probably will be overgeneralized, and then it is possible to refine it using counterexamples. First, the model is built automatically, and then modifications can be applied if necessary – therefore construction of a complete correct model can be avoided, if it does not serve the user's needs. If some temporal property is violated, a counterexample can be examined, and either it is a correct one, or it can be concluded that model behavior does not correspond to the system behavior. Therefore additional constraints can be manually formulated to prohibit this behavior. Additional conditions can only restrict it, so an error can be only one-sided in the assumption that original traces are

TABLE II
CONFORMANCE OF MODELS TO TRACES. FOR EACH TRACE GENERATION METHOD, THE RELATIVE NUMBER OF THE GENERATED TRACES ACCEPTED BY THE GIVEN MODEL IS SPECIFIED. "CONSTRAINT" DENOTES CONSTRAINT-BASED APPROACH, "EXPLICIT" — EXPLICIT-STATE APPROACH. SEMIRANDOM(k) DENOTES THE SEMIRANDOM METHOD WITH $C = k$

Trace generation method	Model							
	Original		Random		Semirandom(10)		Semirandom(100)	
	Constraint	Explicit	Constraint	Explicit	Constraint	Explicit	Constraint	Explicit
original	100%	99.96%	31%	14%	76.3%	14%	100%	24%
random	0%	0%	100%	98.7%	100%	96.5%	100%	79%
semirandom(10)	0.07%	0.07%	99.9%	94%	100%	96%	100%	93.5%
semirandom(100)	1.3%	2.5%	67%	63%	99.4%	82%	100%	93.5%

TABLE III
METRICS OF EXPLICIT-STATE MODELS. "VERIFICATION TIME" DENOTES TIME REQUIRED TO VERIFY 2860 TRACES OF LENGTH 8 USING NUSMV

Parameter	Trace generation method			
	Original	Random	Semirandom(10)	Semirandom(100)
Size (KB)	19	46	109	204
The number of states	62	46	127	246
Number of supported transitions	186	1315	2522	3500
Verification time (sec)	16	22	205	3028

complete. Such model refinement can be performed several times and, as a result, either counterexamples will converge to correct ones or the property will become satisfied.

IX. CONCLUSION

In this work we proposed three plant model trace generation methods and studied their performance in the context of automatic formal plant model inference. Methods for quality assessment of the inferred formal models were also suggested. It was shown that the original constraint-based approach suffers from some major flaws: inability to capture changes of the specific output even when it depends only on one input; insufficient constraints in the case when several parameters are strongly binded; over-restriction of model behavior because of the changeability constraint.

Methods to deal with discovered flaws were proposed, such as: additional constraints, manual variable grouping, fairness constraints which prohibit eternal moving in the same direction within some segment and can be automatically derived in some cases.

It was shown that semirandom models with reasonably large C are adequate considering both trace acceptance and temporal properties verification. Considering its simplicity, the proposed semirandom trace generation method can be applied to a wide range of systems. Also the constraint-based approach with the suggested modifications was shown to be superior to the explicit-state one in terms of the number of correctly verified LTL properties and trace conformance.

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

[3] I. P. Buzhinsky, S. V. Kazakov, V. I. Ulyantsev, F. N. Tsarev, and A. A. Shalyto, "Modification of the method of generation of control finite-state machines with continuous actions based on training examples," *Journal of Computer and Systems Sciences International*, vol. 54, no. 6, pp. 853–865, 2015.

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

[5] 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.

[6] J. Machado, B. Denis, and J.-J. Lesage, "A generic approach to build plant models for DES verification purposes," in *Discrete Event Systems, 2006 8th International Workshop on*. IEEE, 2006, pp. 407–412.

[7] 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)*, 2016, pp. 736–739.

[8] ——, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Transactions on Industrial Informatics*, 2017.

[9] E. F. Moore, "Gedanken-experiments on sequential machines," *Automata studies*, vol. 34, pp. 129–153, 1956.

[10] I. Buzhinsky, A. Sandru, A. Pakonen, D. Chivilikhin, V. Ulyantsev, A. Shalyto, and V. Vyatkin, "Generation of formal plant models based on simulation environments." [Online]. Available: http://eurosim2016.automaatioseura.fi/images/sas/posters/Eurosim2016_poster_Buzhinsky.pdf

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model verifier," in *International Conference on computer aided verification*. Springer, 1999, pp. 495–499.

[12] IEC 61499-1:2012. Function blocks — part 1: Architecture. [Online]. Available: https://webstore.iec.ch/publication/5506

[13] NxtStudio homepage. [Online]. Available: http://www.nxtcontrol.com/en/engineering/