Function block finite-state model identification using SAT and CSP solvers

Daniil Chivilikhin, Vladimir Ulyantsev, Anatoly Shalyto and Valeriy Vyatkin

Citation: Chivilikhin D., Ulyantsev V., Shalyto A., Vyatkin V. Function block finite-state model identification using SAT and CSP solvers / To appear IEEE Transactions on Industrial Informatics, 2019

IEEE-published preprint: https://doi.org/10.1109/TII.2019.2891614

Function block finite-state model identification using SAT and CSP solvers

Daniil Chivilikhin, Vladimir Ulyantsev, Anatoly Shalyto, *Member, IEEE* and
Valeriy Vyatkin, *Senior Member, IEEE*

***Abstract*—We propose a two-stage exact approach for identifying finite-state models of function blocks based on given execution traces. First, a base finite-state model is inferred with a method based on translation to the Boolean satisfiability problem (SAT), then the base model is generalized by inferring minimal guard conditions of the state machine with a method based on translation to the constraint satisfaction problem (CSP).**

I. Introduction

Finite-state models are widely used in software engineering and control systems development for testing [1], verification [2], and also in the context of reverse engineering of programs [3], [4]. However, due to considerable effort required, models are seldom developed and, even if available at some point, are rarely maintained to be up-to-date. Therefore, automatic inference of (finite-state) models of software and its components is a topic of great interest and importance: for instance, it allows generating visual models of existing software [5] and inferring models of dependable systems which satisfy given temporal properties [6], [7], [8]. Finite-state model identification methods in their majority rely on discrete optimization procedures: metaheuristic approaches such as genetic algorithms or simulated annealing or complete ones such as methods of Boolean satisfiability problem (SAT) [9] and constraint satisfaction problem (CSP) [10] solving.

Finite-state models are used not only for architecture design, verification and analysis of control systems, but also for their direct implementation. An example is the international standard for distributed automation systems development IEC 61499 [11], where control systems are developed using *function blocks* (FBs) encapsulating control logic and data processing, while finite-state machines are used as basic program elements. However, a large programmable logic controllers (PLC) code base written in IEC 61131-3 [1] languages exists and is widely used. One of the main motivations of this article is automation of code migration from IEC 61131-3 to the modern and actively developed IEC 61499 standard [12], [13], [14]. Existing migration methods rely on legacy source code and thus their application is limited to cases when this source code is available. Automated finite-state model identification based on behaviour examples may enable automatic migration which does not require legacy source code. Behaviour examples can be recorded directly from the PLC during normal operation of the legacy system over a required characteristic time interval. Then, if the finite-state model identification algorithm is scalable enough, the state machine can be reconstructed based on the collected dataset and later used instead of the legacy controller.

This work proposes a method for inferring a finite-state model of an FB from its behaviour examples, so-called execution traces, which can be derived using black-box testing. In preliminary research published in conference proceedings [15] we proposed an exact FB identification method based on translation to CSP. There a very simple representation of a state machine was used where a transition was labeled with a guard condition involving all input variables, producing models that do not generalize to unseen input data. This was countered by a heuristic greedy algorithm for minimizing (simplifying) guard conditions of the inferred state machine – the algorithm tries to delete each input variable from each guard condition; the change is retained only if the modified automaton still satisfies the execution traces.

This paper follows the aforementioned two-stage FB model identification scheme, but substantially extends previous results both qualitatively and quantitatively and with the following unique contributions: (1) we propose a more efficient base FB model identification approach based on translation to SAT instead of CSP, (2) we substitute the heuristic guard condition minimization algorithm with a new exact one based on translation to CSP.

The rest of this paper is structured as follows. In Section II necessary definitions are given and a formal problem statement is set. The main differences of the current research from related work are highlighted in Section III. Then, the proposed SAT-based FB model identification method is described in Section IV. Section V is devoted to the formal statement of the guard conditions minimization problem and the description of the method proposed for solving it. Section VI describes performed experimental evaluation of the suggested approach and comparison with related work. Finally, Section VII concludes the paper and formulates some directions of future work.

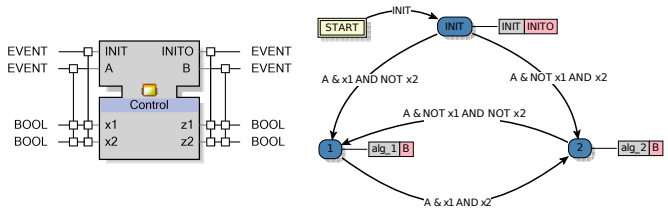
II. FB model identification problem statement

A function block in IEC 61499 is an entity that encapsulates some event and data processing functionality. The *interface* of

D. Chivilikhin, V. Ulyantsev and A. Shalyto are with the Computer Technologies Laboratory, ITMO University, St. Petersburg 197101, Russia (email: chivdan@corp.ifmo.ru, ulyantsev@corp.ifmo.ru, shalyto@mail.ifmo.ru).

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

[1] https://webstore.iec.ch/searchform&q=61131



(a) FB interface example (b) ECC example

Fig. 1. Function block interface (a) and execution control chart example (b)

a function block describes used input/output events, variables and their types, e.g. Boolean, integer, real. For example, consider the FB interface shown in Fig. 1a: it describes input events `A` and `INIT`, output events `B` and `INITO`, Boolean input variables x_1, x_2, and Boolean output variables z_1, z_2.

Two main FB types are composite and basic blocks: a *composite* FB is a network of other FBs (basic or composite), whereas a *basic* FB is defined by a Moore finite-state machine called an *execution control chart* (ECC). A transition of an ECC is labeled by an input event and a *guard condition* – a Boolean formula including input, output, and internal variables, logical operators and constants. In this work we only consider identification of basic FBs and focus on a smaller class with only Boolean input/output variables and no internal variables.

An ECC state can be associated with a list of *output actions*, each described by an output event and an *ECC-algorithm*. Algorithms can be implemented using IEC 61131-3 languages (e.g. *Structured Text*) and are used for changing the values of output variables. An example of an ECC is shown in Fig. 1b, algorithms are denoted as `INIT`, `alg_1`, `alg_2`.

For a complete formal definition of an ECC we refer the reader to [16]. Here a simplified one is sufficient for the case when (1) guard conditions depend on input variables only and (2) each state has exactly one associated output action. With respect to these assumptions, an ECC is described by a set of C states with one state emphasized as *initial*, input events $E^{\text{in}} = \{e_1^{\text{in}}, e_2^{\text{in}}, \ldots\}$, input variables $X = \{x_1, x_2, \ldots\}$, output events $E^{\text{out}} = \{e_1^{\text{out}}, e_2^{\text{out}}, \ldots\}$, output variables $Z = \{z_1, z_2, \ldots\}$ and two functions: (1) a transition function defining for each state transitions to other states, where each transition is marked with an input event and a guard condition (a Boolean formula over input variables) and (2) an output actions function that for each state defines transformation of output variables values and generation of output events.

This paper suggests a new approach for identifying a finite-state model of a (basic) FB with known interface but unknown implementation based on examples of its behaviour – so-called execution traces. The approach does not depend on the procedure employed for behaviour examples collection – it is possible to gather traces using a simulation model of the control system or collect them from actual hardware. The only limitation is that, since the target FB is considered to be a "black box", execution traces may only be collected using *behavioural* or *black-box* testing – a testing methodology which does not use knowledge about the internal structure of the system under test.

An *execution trace* is a finite ordered sequence of *trace elements* $s_i = \langle e^{\text{in}}[\bar{x}], e^{\text{out}}[\bar{z}]\rangle$, where $e^{\text{in}} \in E^{\text{in}}$ is an input event, $\bar{x} \in \{0,1\}^{|X|}$ is an input variables values tuple, $e^{\text{out}} \in E^{\text{out}} \cup \{\varepsilon\}$ (ε denotes "no output event") is an output event, and $\bar{z} \in \{0,1\}^{|Z|}$ is an output variables values tuple. Below is an example of a set $\mathbb{T}$ of two traces T_1 and T_2 where $E^{\text{in}} = \{A\}$, $E^{\text{out}} = \{B\}$, $X = \{x_1, x_2\}$, $Z = \{z_1, z_2\}$:

$$
\begin{aligned}
T_1 = \Big[&\langle A[x_1 = 1, x_2 = 0], B[z_1 = 1, z_2 = 0]\rangle; \\
&\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 1]\rangle; \\
&\langle A[x_1 = 0, x_2 = 0], B[z_1 = 0, z_2 = 1]\rangle\Big] \\
T_2 = \Big[&\langle A[x_1 = 0, x_2 = 1], B[z_1 = 0, z_2 = 1]\rangle; \\
&\langle A[x_1 = 0, x_2 = 0], B[z_1 = 1, z_2 = 1]\rangle; \\
&\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 0]\rangle\Big]. \quad (1)
\end{aligned}
$$

An automaton *satisfies a trace element* $\langle e^{\text{in}}[\bar{x}], e^{\text{out}}[\bar{z}]\rangle$ in state s if after receiving input event e^{in} with input variable values $\bar{x}$ the automaton generates output event e^{out} and values of its output variables become equal to $\bar{z}$. Consequently, an automaton *satisfies a trace* T if it satisfies all its elements in corresponding states, starting from the initial one.

The problem addressed in this paper is finding a finite-state model of the target FB which satisfies collected execution traces $\mathbb{T}$ and has the minimal possible number of states. Additionally, in order for the solution to be general, guard conditions of the inferred finite-state model must have the minimum possible structural complexity – in this paper this is formalized by minimizing the total number of nodes in parse trees of Boolean formulas representing the guard condition.

III. Related work

The so-called extended finite-state machine (EFSM) is the finite-state model that is probably most similar to an ECC: its transitions are also labeled with input events and Boolean formulas over input variables, and states are associated with sequences of output actions. However, output actions are not self-contained and merely represent the names of functions or procedures defined in the plant. Several approaches exist for inferring EFSMs from behaviour examples, among which the most efficient is the exact one based on translation to SAT [7]. The current work has two main differences from research on EFSM inference, which, as demonstrated further in this paper, make existing methods inefficient for the addressed FB model identification problem.

First, mentioned EFSM inference methods assume that for each trace element the guard condition (Boolean formula over input variables) is known in advance. Such data cannot be acquired using black-box testing – only values of input variables can be accessed. Though this can be mitigated by assuming that each guard condition includes values of all input variables, this will lead to incomprehensible state machines with large guard conditions. The method proposed in this paper also uses this approach in its first step (base FB model

inference), but later guard conditions are minimized with an exact algorithm.

Second, EFSM output actions do not contain algorithms, they are regarded as merely names of algorithms which are assumed to be defined in the plant: an inference algorithm only needs to select which algorithms to run. This issue can be dealt with by assuming that each unique output variable values tuple encountered in the traces is a separate EFSM-style output action. However, this will lead to an increased number of states since each such output action will be associated with a separate EFSM state.

On the contrary, in the current work we infer a simple algorithm that modifies output variable values in each state – in this way, the proposed method derives computational models. Computational EFSMs are also inferred in [17], although a base finite-state model is assumed to be available beforehand – the method only aims at inferring algorithms for computing output variable values. Moreover, this is done using genetic programming [18], which is an inexact approach and is not guaranteed to find a correct solution. In our work both the automaton and the associated algorithms are inferred simultaneously.

Finally, to the best of our knowledge, inference of state machines with guaranteed minimal guard conditions has not been considered in the literature before. Our work presents the first approach to this by formally stating this problem and providing an exact algorithm.

IV. FB MODEL IDENTIFICATION USING SAT SOLVER

The main idea of the proposed base FB model identification method is to *translate* the problem of inferring a finite-state model of an FB with a given number of states C from execution traces $\mathbb{T}$ to Boolean satisfiability. Finding the solution of the original problem – inferring a state machine with a minimal number of states $C_{\min}$ – is achieved through solving a series of decision problems with $C = 1, 2, \ldots, C_{\min}$.

The base of the proposed method is the *translation function* f, which for the given problem $\langle \mathbb{T}, C \rangle$ constructs a Boolean formula that is satisfiable if and only if the original problem has a solution. First, the input data of the FB model inference problem – set of traces $\mathbb{T}$ and number of states C – is passed to the translation function f, which constructs a Boolean formula that is satisfiable if and only if the FB model inference problem has a solution. The translation works in two steps: (1) trace tree construction, where a prefix tree is built from given traces, and (2) Boolean formula construction. Second, a SAT solver is used to find a satisfying assignment of the Boolean formula. Third, if a solution is found, the sought automaton is constructed from found values of Boolean variables. Otherwise the user is notified that a solution with C states does not exist.

A. Trace tree construction

As a first step of FB model identification, collected traces $\mathbb{T}$ are generalized by representing them in the form of a *trace tree* (E, V) – a prefix tree that contains all prefixes of all traces. An edge $uv \in E$ $(u, v \in V)$ of the tree corresponds to an input event from E^{in} and a tuple of input variable values from $\{0,1\}^{|X|}$. A vertex $v \in V$ of the tree corresponds to an output event from E^{out} and a tuple of output variable values from $\{0,1\}^{|Z|}$. The tree constructed from example traces (Fig. 1) is shown in Fig. 2. For convenience, the trace tree is represented using the following integer arrays:

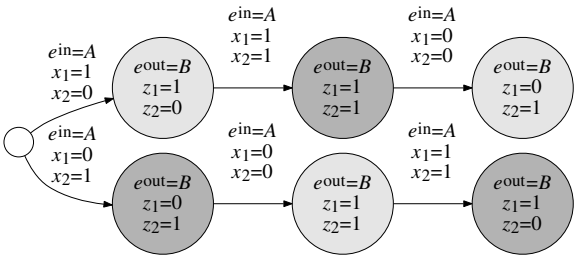


Fig. 2. Trace tree constructed from example traces (Fig. 1), different colors correspond to different automaton states

- $e^{\text{in}}_{uv} \in [1..|E^{\text{in}}|]$ – index of input event on edge $uv \in E$;
- $e^{\text{out}}_{v} \in [1..|E^{\text{out}}|]$ – index of output event in vertex $v \in V$;
- $z_{v,i} \in \{0,1\}$, $i \in [1..|Z|]$ – value of i-th output variable in vertex $v \in V$;
- $g_{uv} \in [1..|G|]$ – index of the input tuple on edge $uv \in E$ in the ordered set of all unique input tuples G present in traces $\mathbb{T}$. For example, for traces (1) we have $G = \{\langle 0,0\rangle; \langle 0,1\rangle; \langle 1,0\rangle; \langle 1,1\rangle\}$, therefore values of g_{uv} are selected from the interval $[1..4]$.

B. Translation to SAT

In this section we describe construction of a Boolean formula that is satisfiable if and only if the constructed automaton satisfies given traces $\mathbb{T}$ and is deterministic. The Boolean formula also contains some auxiliary (redundant) clauses that do not influence the existence of a solution but speed up SAT solving.

1) Problem representation: The employed method of representing the sought automaton with Boolean variables is described here. The main idea is to find a *coloring* of the trace tree vertices that uses exactly C colors, such that the automaton produced by merging all vertices with the same color into a single state satisfies given execution traces and is deterministic. Therefore, for each vertex $v \in V$ of the tree its *color* is encoded as $c_{v,i}$ $(1 \leq i \leq C)$ which is true if and only if vertex v has color i.

A guard condition on a transition of the automaton corresponds to an input variables values tuple present in traces $\mathbb{T}$, therefore each guard condition depends on all input variables. Note that this does not narrow the field of applicability of the proposed method since all input variables values tuples present in traces will also be present in the constructed automaton.

Transitions of the sought automaton are represented with variables y_{n_1,e,g,n_2} $(1 \leq n_1, n_2 \leq C,\ 1 \leq e \leq |E^{\text{in}}|,\ 1 \leq g \leq |G|)$ – target state for a transition originating from state n_1, marked with input event number e and guard condition (unique input variables values tuple) number g, is state n_2.

The output actions function encodes output events and also ECC-algorithms that calculate values of output variables based on their previous values. ECC-algorithms are described with

variables $d^0_{n,i}$ and $d^1_{n,i}$ which encode the value of the i-th output variable for the algorithm in state n for cases when its previous value is `False` ($d^0_{n,i}$) or `True` ($d^1_{n,i}$). Output events are encoded with variables $o_{n,j}$ ($1 \leq j \leq |E^{\text{out}}|$), which are true if and only if state n has output event j.

2) Main Boolean constraints: In this section the main constraints are described which encode that the sought automaton must (1) satisfy given execution traces and (2) be deterministic. The constraints are verbally described in this section, formal definitions are given in Table I.

First, constraints on the initial state of the automaton are formulated. Without loss of generality, the first state corresponds to the root of the trace tree. Then, since it is not always possible to include the initialization phase of FB execution into the traces, we require the initial state to unconditionally zero out all output variables and not to issue an output event. These constraints are defined in section 1 of Table I.

Next, each state must be colored in at least one and at most one color (section 2 of Table I). For each trace tree edge $uv \in E$, if u is colored with color n_1 (c_{u,n_1}) and v is colored with color n_2 (c_{v,n_2}), then there must exist a transition from state n_1 to state n_2 marked with input event e^{in}_{uv} and guard condition g_{uv}. In order for the automaton to be deterministic, there must be at most one transition from state n_1 labeled with a particular input event e and guard condition g (section 3 of Table I).

If the color of node v is n, the output event in state n must correspond to the output event in the trace tree vertex v. In addition, there must be at least one and at most one output event in each state (section 4 of Table I). For each vertex $v \in V$, the value of the i-th output variable in state n is constrained by the values of this variable in vertex v and its parent u (section 5 of Table I).

In addition, auxiliary symmetry breaking constraints proposed in [19] are used to reduce the search space by demanding that the constructed automaton's states should be enumerated in the order they are visited by the breadth-first search (BFS) algorithm launched from the initial state. These constraints eliminate all but one solution for each equivalence class of isomorphic automata. We also employ a second type of auxiliary constraints that force all transitions not covered by traces to be self-loops. For brevity we omit formally describing the auxiliary constraints and refer the reader to [19].

3) Limiting the number of transitions to unique states: Experimentation showed that inferring FB models only with constraints described above produces state machines that may have an unnecessarily large number of transitions – each state could have a transition to up to all other $C-1$ states.

To counter this and to produce models with more balanced states, after finding a model with the minimum number of states C we can try to minimize the maximal number K of different states that each state is allowed to have transitions to. For that we first introduce variables ζ_{n_1,n_2} to represent whether there exists a transition from state n_1 to state n_2. Then variables $\gamma_{n_1,n_2,l}$ are introduced to represent this maximal number of transitions to unique states: $\gamma_{n_1,n_2,l}$ is true if and only if the number of transitions from state n_1 to different states with numbers smaller than or equal to n_2 is exactly l. Now it remains to demand that each state has at most K transitions to different states. These constraints are formalized in section 6 of Table I.

TABLE I
BOOLEAN CLAUSES FOR FB MODEL INFERENCE

	Constraint	Range
1.1	$c_{1,1}$	
1.2	$\neg d^0_{1,i} \wedge \neg d^1_{1,i}$	$1 \leq i \leq \lvert Z\rvert$
1.3	$\neg o_{1,j}$	$1 \leq j \leq \lvert E^{\text{out}}\rvert$
2.1	$c_{v,1} \vee \ldots \vee c_{v,C}$	$v \in V$
2.2	$c_{v,n_1} \rightarrow \neg c_{v,n_2}$	$1 \leq n_1 < n_2 \leq C$
3.1	$c_{u,n_1} \wedge c_{v,n_2} \rightarrow y_{n_1,e^{\text{in}}_{uv},g_{uv},n_2}$	$\begin{cases} u,v \in V \\ uv \in E \\ 1 \leq n_1, n_2 \leq C \end{cases}$
3.2	$y_{n_1,e,g,n_2} \rightarrow \neg y_{n_1,e,g,n_3}$	$\begin{cases} 1 \leq n_1, n_2, n_3 \leq C \\ n_1 \leq n_2 < n_3 \\ 1 \leq e \leq \lvert E^{\text{in}}\rvert \\ 1 \leq g \leq \lvert G\rvert \end{cases}$
4.1	$c_{v,n} \rightarrow o_{n,e^{\text{out}}_v}$	$v \in V$, $1 \leq n \leq C$
4.2	$o_{n,1} \vee \ldots \vee o_{n,\lvert E^{\text{out}}\rvert}$	$1 \leq n \leq C$
4.3	$o_{n,j_1} \rightarrow \neg o_{n,j_2}$	$\begin{cases} 1 \leq n \leq C \\ 1 \leq j_1 < j_2 \leq \lvert E^{\text{out}}\rvert \end{cases}$
5	$c_{v,n} \rightarrow \begin{cases} \neg d^0_{n,i}, & \text{if } \neg z_{u,i} \wedge \neg z_{v,i}; \\ d^0_{n,i}, & \text{if } \neg z_{u,i} \wedge z_{v,i}; \\ \neg d^1_{n,i}, & \text{if } z_{u,i} \wedge \neg z_{v,i}; \\ d^1_{n,i}, & \text{if } z_{u,i} \wedge z_{v,i}. \end{cases}$	$u,v \in V$ $uv \in E$ $1 \leq i \leq \lvert Z\rvert$ $1 \leq n \leq C$
6.1	$\zeta_{n_1,n_2} \leftrightarrow \bigwedge_{e,g} y_{n_1,e,g,n_2}$	$1 \leq n_1, n_2 \leq C$
6.2	$\bigvee_n \gamma_{n,1,0}$	$1 \leq n \leq C$
6.3	$\gamma_{n_1,n_2-1,k} \wedge \zeta_{n_1,n_2} \rightarrow \gamma_{n_1,n_2,k+1}$	$\begin{cases} 1 \leq n_1, n_2 \leq C \\ n_2 > 1 \\ 0 \leq k < C \end{cases}$
6.4	$\gamma_{n_1,n_2-1,k} \wedge \neg\zeta_{n_1,n_2} \rightarrow \gamma_{n_1,n_2,k}$	$\begin{cases} 1 \leq n_1, n_2 \leq C \\ n_2 > 1 \\ 0 \leq k \leq C \end{cases}$
6.5	$\neg\gamma_{n,C,k}$	$1 \leq n \leq C$ $K+1 \leq k \leq C$

C. Minimum FB model construction

Denote the procedure or searching for a model with C states satisfying execution traces $\mathbb{T}$ as `findModel`$(\mathbb{T}, C)$. If for a particular value of C the corresponding SAT formula is unsatisfiable, the SAT solver issues a corresponding message, and the `findModel` procedure returns an empty model ($\varnothing$). Otherwise, the state machine is built using found values of Boolean variables and the trace tree. Similarly, denote the same procedure but with limiting the maximal number of transitions to different states to K as `findModel`$(\mathbb{T}, C, K)$. The model with the minimum number of states and transitions to unique states is inferred by Algorithm 1.

The base FB model constructed from the trace tree from Fig. 2 is shown in Fig. 3. The initial state is marked with an incoming transition with no source. State colors correspond to the colors of trace tree vertices. In each state s an algorithm for updating each variable is written. An example of how an

Algorithm 1: Minimum base FB model inference

```
Data: execution traces 𝕋
C ← 1
while True do
    A ← findModel(𝕋, C)
    if A = ∅ then C ← C + 1
    else
        foreach K ← 1 to C do
            A ← findModel(𝕋, C, K)
            if A ≠ ∅ then return A
```

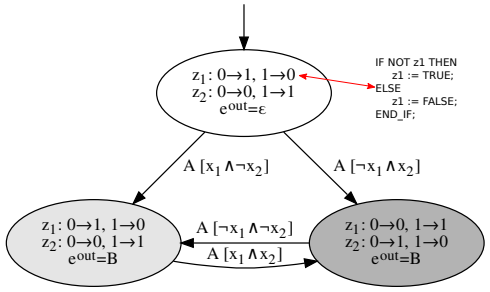


Fig. 3. FB model constructed from the trace tree from Fig. 2

algorithm described by variables d_s^0 and d_s^1 is translated to *Structured Text* code is shown in Fig. 3.

V. Exact guard conditions minimization using CSP solver

In previous work [15], after an automaton satisfying given traces was found, guard conditions on its transitions were minimized (generalized) using a greedy algorithm:

- try to remove each input variable from each guard condition;
- retain the change if the modified automaton still satisfies all traces, otherwise revert;
- repeat until no more input variables can be removed.

Below we formally state the guard conditions minimization problem and propose an exact approach for solving it based on translation to CSP.

A. Guard conditions minimization problem statement

Informally, the problem is, for a given state machine A and set of execution traces $\mathbb{T}$, to find a state machine A' such that it satisfies all traces $\mathbb{T}$ and has the minimum possible complexity of guard conditions. This complexity can be measured in different ways, but in this work we use the total number of nodes in the parse trees of Boolean formulas representing the guard conditions.

More formally, consider automaton A in state s processing a trace element with input event e^{in} and input variable values $\bar{x} = \langle x_1, \ldots, x_{|X|} \rangle$. Then, the following constraints are imposed on automaton A':

- if for inputs $e^{\text{in}}[\bar{x}]$ automaton A makes a transition to state s', then A' should also make a transition to the same state;
- if for $e^{\text{in}}[\bar{x}]$ automaton A does not make any transitions, then A' should behave the same way.

Following from this formulation, the guard conditions minimization problem can be solved for each state separately.

Consider state s of automaton A with K' transitions to exactly K different states $s'_1, \ldots, s'_K$ ($K' \geq K$). We run the traces $\mathbb{T}$ through automaton A and consider D trace elements that are processed in the selected state s. For each d-th trace element the following information is recorded:

- $\delta_d \in [0..K]$ ($1 \leq d \leq D$) – index $k \in [1..K]$ if the fired transition leads to state s_k, or zero if no transition was fired;
- $\omega_{d,i}$ ($1 \leq d \leq D$, $1 \leq i \leq |X|$) – value of input variable x_i before the d-th trace element is processed; ω_d denotes the tuple $\omega_{d,1} \ldots \omega_{d,|X|}$.

In the most general way, the sought solution is described with K Boolean functions $f_1 \ldots f_K$ ($f_k \colon \{0,1\}^{|X|} \to \{0,1\}$, $1 \leq k \leq K$) representing guard conditions on transitions to K destination states. These functions are subject to the following constraints. First, for all data elements for which a transition has fired ($\delta_d \neq 0$), function f_{δ_d} must evaluate to `True`, while all functions f_k with $k < \delta_d$ should evaluate to `False`:

$$\bigwedge_{1 \leq d \leq D \colon \delta_d \neq 0} \left(f_{\delta_d}(\omega_d) \wedge \bigwedge_{1 \leq k < \delta_d} \neg f_k(\omega_d) \right). \tag{2}$$

Second, for all data elements for which no transition has been fired ($\delta_d = 0$), all functions $f_1 \ldots f_K$ must evaluate to `False`:

$$\bigwedge_{1 \leq d \leq D \colon \delta_d = 0} \left(\bigwedge_{1 \leq k \leq K} \neg f_k(\omega_d) \right). \tag{3}$$

Mathematically, the considered problem is equivalent to finding the minimum representation of a Boolean function given its incomplete truth table, where missing values correspond to situations when the value of the function is not important.

B. Guard conditions minimization using CSP solver

Our approach is to explicitly encode the structure of Boolean formulas for each guard condition, define rules of their values calculation, and incrementally search of a set for K Boolean formulas of minimum size.

1) Boolean function structure encoding: The structure of each Boolean function representing a single guard condition is described as a (possibly, incomplete) binary parse tree with at most P nodes. Since there are exactly K functions to synthesize for each state, types of all nodes can be described with Boolean variables $\tau_{k,p,i}$ ($1 \leq k \leq K$, $1 \leq p \leq P$, $i \in \{\alpha, \wedge, \vee, \neg, \$\}$), where α denotes a terminal node, $\wedge$, $\vee$, $\neg$ are logical connectors and "$" denotes a none-typed node. The latter are needed to model "extra" nodes that are not included in the tree.

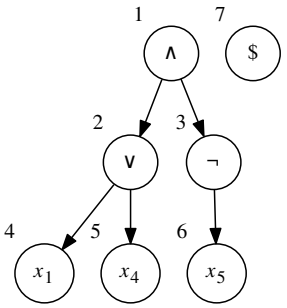


Fig. 4. Tree representation of Boolean formula $(x_1 \vee x_4) \wedge \neg x_5$ ($P = 7$)

Terminal nodes ($\tau_{k,p,\alpha}$) represent FB input variables and have an associated terminal number – index of the input variable from X. Terminal numbers are represented with variables $t_{k,p} \in [0..|X|]$ where $t_{k,p} = 0$ corresponds to a non-terminal node. Node children are represented with variables $l_{k,p} \in [1..(P+1)]$ (left child) and $r_{k,p} \in [1..(P+1)]$ (right child). If $l_{k,p} = P+1$ or $r_{k,p} = P+1$ then the corresponding node does not have a left (right) child. An example of a guard condition tree representation for formula $(x_1 \vee x_4) \wedge \neg x_5$ with $P = 7$ is shown in Fig. 4.

Below we verbally describe constraints on mentioned variables, while formal definitions are presented in Table II. Here I is a function such that $I(\texttt{True}) = 1$, $I(\texttt{False}) = 0$.

Node types (section 1 of Table II). Each node of each tree has to have at least one type and at most one type. Only terminal-typed nodes can have a non-zero terminal number $t_{k,p}$. In addition, we would like all none-typed nodes to have larger numbers than all other nodes in a tree. Therefore, if the p-th node is none-typed, then the next node also should have the same type.

Number of children (section 2 of Table II). Terminal-typed and none-typed nodes do not have children. Nodes "$\wedge$" and "$\vee$" must have two children, while a "$\neg$" node only has the left child.

Children enumeration and ordering (section 3 of Table II). If a node has a defined child then this child is not none-typed. Children of a defined node should have greater numbers than their parents and must be ordered from left to right. If the left child is undefined then the right child is also undefined.

Tree structure constraints (section 4 of Table II). Each guard condition should indeed be a tree – for each k-th tree the number of edges E'_k must be equal to the number of nodes V'_k minus one.

2) Boolean function values encoding: Next, Boolean variables $\text{val}_{k,p,d}$ represent the value of node p in function k for data element ω_d. Auxiliary variables $\text{val}^l_{k,p,d}$ and $\text{val}^r_{k,p,d}$ represent values of the left and right child of a node, respectively.

Children values (section 5 of Table II). Left child value $\text{val}^l_{k,p,d}$ is expressed using $l_{k,p}$ and $\text{val}_{k,p,d}$ variables, right child value – similarly using $r_{k,p}$ and $\text{val}_{k,p,d}$.

Node values (section 6 of Table II). Now, having left and right child value variables $\text{val}^l_{k,p,d}$ and $\text{val}^r_{k,p,d}$, values of parent nodes are expressed according to their types and children values.

3) Transition firing constraints: "Transition fired" and "transition not fired" variables definition (section 7 of Table II). Note that the value of the k-th guard for d-th data element is the value of the root node $\text{val}_{k,1,d}$. We alias this with variables $\psi_{k,d}$ which bear the meaning "k-th transition fired for d-th data element" with trivial constraints. To make the encoding more compact we also introduce variables $\overline{\psi}_{k,d}$ bearing the meaning "k-th transition does not fire for d-th data element".

"First transition fired" variables definition (section 8 of Table II). To efficiently encode constraints "if the k-th transition fired then all previous transitions do not fire" we introduce Boolean variables $\psi'_{k,d}$ such that $\psi'_{k,d} = \texttt{True}$ if and only if $\forall k' < k$: $\neg\psi_{k',d}$. To encode this we first note that $\psi'_{1,d} \leftrightarrow \psi_{1,d}$. Then the required condition can be encoded linearly using "transition not fired" variables $\overline{\psi}$.

Main constraints (section 9 of Table II). At last, the main functional conditions synonymous to (2) and (3) may be encoded. First, the constraint "all transitions that fired in the original automaton must fire in the simplified one" is encoded using ψ' variables. Second, the constraint "transitions that did not fire in the original automaton should not fire" is expressed using $\overline{\psi}$ variables.

4) Iterative minimum guard conditions inference: Let us denote as $\texttt{findGuards}(K, P, \delta, \omega)$ the procedure of finding guard conditions of K transitions for fixed P given data on fired transition numbers δ and input variables values ω. This procedure constructs a set of constraints described in Table II and uses a CSP solver to find a satisfying assignment of variables or demonstrate that it does not exist. The value of P is given as an essential parameter of the problem. If the current value of P is smaller than the minimal possible one, then the CSP solver will return an unsatisfiability message and the `findGuards` function will return an empty solution ($\varnothing$).

Minimum possible guard conditions should correspond to the minimum value of the total number of typed nodes N in all trees and depend on P. While the value of P is given as an essential parameter of the problem and should be iterated by an external algorithm, minimizing the total number of nodes in all trees N is done by using the CSP solver in optimization mode to minimize the expression:

$$\sum_{1 \le k \le K} \sum_{1 \le p \le P} I(\neg\tau_{k,p,\$}) \to \min .$$

The described iterative procedure is formalized in Algorithm 2.

Algorithm 2: Iterative minimum guard conditions inference

```
Data: number of transitions K, indices of fired
      transitions δ, input variable values ω
foreach P ← 1 to K(2|X| − 1) do
    F ← findGuards (P, δ, ω)
    if F ≠ ∅ then
        return F
```

TABLE II
GUARD CONDITIONS SIMPLIFICATION CONSTRAINTS

	Constraint	Domain
1.1	$\tau_{k,p,\alpha} \vee \tau_{k,p,\wedge} \vee \tau_{k,p,\vee} \vee \tau_{k,p,\neg} \vee \tau_{k,p,\$}$	$1 \le k \le K$ $1 \le p \le P$
1.2	$\tau_{k,p,i} \rightarrow \neg\tau_{k,p,j},\ i < j$	
1.3	$\tau_{k,p,\alpha} \leftrightarrow t_{k,p} \neq 0$	
1.4	$\tau_{k,p,\$} \rightarrow \tau_{k,p+1,\$},\ p < P$	
2.1	$\tau_{k,p,\alpha} \rightarrow \big(l_{k,p} = r_{k,p} = P+1\big)$	
2.2	$\tau_{k,p,\$} \rightarrow \big(l_{k,p} = r_{k,p} = P+1\big)$	
2.3	$\tau_{k,p,\wedge} \rightarrow \big(l_{k,p} \le P \wedge r_{k,p} \le P\big)$	
2.4	$\tau_{k,p,\vee} \rightarrow \big(l_{k,p} \le P \wedge r_{k,p} \le P\big)$	
2.5	$\tau_{k,p,\neg} \rightarrow \big(l_{k,p} \le P \wedge r_{k,p} = P+1\big)$	
3.1	$l_{k,p} \le P \rightarrow \neg\tau_{k,l_{k,p},\$}$	
3.2	$r_{k,p} \le P \rightarrow \neg\tau_{k,r_{k,p},\$}$	
3.3	$\neg\tau_{k,p,\$} \rightarrow \big(l_{k,p} > p \wedge r_{k,p} > p\big)$	
3.4	$\big(l_{k,p} < P+1\big) \rightarrow \big(l_{k,p} < r_{k,p}\big)$	
3.5	$l_{k,p} = P+1 \rightarrow r_{k,p} = P+1$	
4.1	$V'_k = \sum_{1 \le p \le P} \big(I(\neg\tau_{k,p,\$})\big)$	$1 \le k \le K$
4.2	$E'_k = \sum_{1 \le p \le P} \big(I(l_{k,p} \le P) + I(r_{k,p} \le P)\big)$	
4.3	$E'_k = V'_k - 1$	
5	$l_{k,p_1} = p_2 \rightarrow \mathrm{val}^l_{k,p_1,d} = \mathrm{val}_{k,p_2,d}$ $r_{k,p_1} = p_2 \rightarrow \mathrm{val}^r_{k,p_1,d} = \mathrm{val}_{k,p_2,d}$	$1 \le k \le K$ $1 \le p_1, p_2 \le P$ $p_1 < p_2$ $1 \le d \le D$
6.1	$\tau_{k,p,\alpha} \rightarrow \Big(\mathrm{val}_{k,p,d} \leftrightarrow \omega_{d,t_{k,p}}\Big)$	$1 \le k \le K$ $1 \le p < P$ $1 \le d \le D$
6.2	$\tau_{k,p,\wedge} \rightarrow \Big(\mathrm{val}_{k,p,d} \leftrightarrow \mathrm{val}^l_{k,p,d} \wedge \mathrm{val}^r_{k,p,d}\Big)$	
6.3	$\tau_{k,p,\vee} \rightarrow \Big(\mathrm{val}_{k,p,d} \leftrightarrow \mathrm{val}^l_{k,p,d} \vee \mathrm{val}^r_{k,p,d}\Big)$	
6.4	$\tau_{k,p,\neg} \rightarrow \Big(\mathrm{val}_{k,p,d} \leftrightarrow \neg\mathrm{val}^l_{k,p,d}\Big)$	
7.1	$\psi_{k,d} \leftrightarrow \mathrm{val}_{k,0,d}$	$1 \le k \le K$ $1 \le d \le D$
7.2	$\overline{\psi}_{1,d} \leftrightarrow \neg\psi_{1,1,d}$	
7.3	$\overline{\psi}_{k,d} \leftrightarrow \neg\psi_{k,d} \wedge \overline{\psi}_{k-1,d}$, if $k > 1$	
8.1	$\psi'_{1,d} \leftrightarrow \psi_{1,d}$	
8.2	$\psi'_{k,d} \leftrightarrow \psi_{k,d} \wedge \overline{\psi}_{k-1,d}$	
9	$\begin{cases}\psi'_{\delta_d,d}, & \text{if } \delta_d \neq 0\\ \overline{\psi}_{K,d}, & \text{if } \delta_d = 0\end{cases}$	$1 \le d \le D$

VI. CASE STUDY: RECONSTRUCTING A CONTROLLER FB FOR THE PICK-AND-PLACE MANIPULATOR

The proposed approach was evaluated on the example of inferring a controller for the Pick-and-Place (PnP) manipulator [20] shown in Fig. 5. The manipulator is used for moving work pieces (*) that appear on input sliders (1, 2, 3) to the output slider (V). This is done with two horizontal cylinders (I, II), one vertical cylinder (III) and a suction unit (IV) for picking up work pieces (WPs). When a WP appears on an input slider, cylinders are used for positioning the vacuum unit on top of the WP where it is grabbed, moving the WP to the output slider and releasing it. The control system is implemented with IEC 61499 function blocks in *nxtSTUDIO* (www.nxtcontrol.com), the controller is represented by a single basic FB with 10 input and 7 output Boolean variables. The purpose of the experiments was to infer a model of this controller FB using different approaches and to compare their efficiency and quality of the identified models.

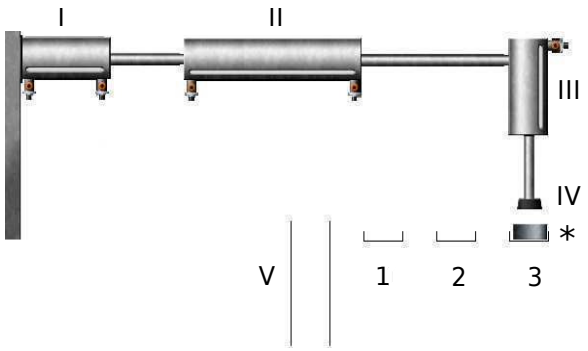


Fig. 5. Pick-and-Place manipulator

A. Traces generation for the PnP system

The simplest approach to generation of black-box traces for a controller is pure random trace generation: on each step a random input event is issued and random values of controller input variables are chosen. However, this approach is not well-suited for systems with a large number of variables since random traces do not ensure sufficient coverage of possible FB behaviours. Furthermore, random traces often do not correspond to any possible behaviour of the closed-loop system: in reality, inputs of the controller are not arbitrary but constrained by the plant.

Therefore, in this paper traces are generated by testing the entire closed-loop system in simulation based on *tests* formulated in terms of the interface of the control system (WP processing) rather than the interface of the controller or plant (events and variables). Two types of training sets were used – simple tests and random tests.

To compare the proposed method with the previous one based on CSP [15], experiments were conducted on traces considered in that paper. A *simple test* of length L_i is a sequence of input slider numbers $\langle w_1, w_2, \dots w_{L_i}\rangle$, $w_j \in \{1,2,3\}$ to which WPs arrive. The test is called *simple* since it is assumed that the manipulator returns to its initial position before the next WP arrives – this restricts the behaviour of the system in the sense that only one WP is present at any moment of time. The used set contains all tests of length up to three: $\langle 1\rangle$, $\langle 2\rangle$, $\langle 3\rangle$, $\langle 1,1\rangle$, …, $\langle 3,3\rangle$, $\langle 1,1,1\rangle$, …, $\langle 3,3,3\rangle$ – a total of 39 tests, which are sorted in lexicographical order.

Random tests for the PnP system were generated in the following way. Each trace was generated as a result of simulation of the system with a total duration of M seconds. During one simulation, each five seconds zero to three WPs were placed on the input trays: WPs were selected randomly without replacement. Before running the next simulation the PnP system was reinitialized. We argue that a sufficient number of such tests, with each test run for a sufficiently large time M, should be sufficient to fully cover possible behaviours of the controller. We generated 20 tests with $M = 60$ and 20 tests with $M = 120$.

All experiments were run on a laptop computer with an Intel(R) Core$^{\text{TM}}$i7-8550U @ 1.80 GHz processor and 16 Gb

TABLE III
EXPERIMENTAL COMPARISON WITH EFSM-TOOLS AND CSP-BASED METHOD: INFERRING BASE FB MODELS

	Training set	$\|V\|$	$\|G\|$	EFSM-tools [22]			CSP-based method [15]		Prosed SAT-based method	
				# states	time without loop constraints, sec.	time with loop constraints, sec.	# states	time, sec.	# states	time, sec.
simple tests	1	10	16	9	16	17	6	2	6	0.2
	2	18	26	9	132	30	7	3	7	0.25
	3	26	39	17	TL	60	8	6	8	0.4
	10	81	39	17	TL	77	8	8	8	1.4
	39	313	40	17	TL	86	8	26	8	5.5
random tests	10×60 sec.	1237	161	17	TL	805	8	91	8	10
	20×60 sec. 20×120 sec.	4778	161	17	TL	1107	8	416	8	19

of RAM. For SAT solving we used the *glucose* [2] tool and for solving CSP *Chuffed* [21] was used.

B. Comparison with EFSM-tools [22]

The first set of experiments is devoted to performance comparison between (1) the proposed base FB model identification method described in Section IV, (2) the EFSM-tools software [22] described in [7], and also (3) with our previous method from [15]. To compare with EFSM-tools we created a wrapper tool that transforms our execution traces to the input format accepted by EFSM-tools, runs EFSM-tools, and validates the generated state machine. EFSM-tools was used in two different variations, in the first one the original code was used. In the second one we added constraints forcing transitions not covered by traces to be self-loops [19] – this proved to be crucial for solving unsatisfiable instances.

Results of experimental runs are summarized in Table III. Here $|V|$ is the number of vertices in the traces trees and $|G|$ is the number of unique input variables tuples encountered in traces. Each next training set contains all previous ones as a subset. Run times of algorithms include all iterations starting from $C = 1$ up to the first satisfiable run of the solver in which the minimal solution is constructed.

As seen from Table III, EFSM-tools always constructs a solution with a larger number of states compared to the proposed SAT-based method – this is due to the used state machine model, in which one state corresponds to one unique output variables values tuple. Furthermore, the original EFSM-tools code without self-loops fixing constraints was not able to find solutions for training sets larger than the second one in under a time limit (TL) of one hour. However, adding these constraints allowed finding solutions for all training sets, though considerably slower than with the proposed method. Our previous method from [15] was able to find minimal solutions for all training sets, but also much slower than the method proposed in this paper.

[2]http://www.labri.fr/perso/lsimon/glucose/

C. Inferring models with minimum guard conditions

The purpose of the second set of experiments was to evaluate the ability of the proposed approach to infer minimum guard conditions of previously generated FB models and its efficiency. For each training set and corresponding FB model generated in previous experiments, we ran the heuristic guard conditions minimization algorithm used in [15] and the proposed exact CSP-based minimization algorithm. For each resulting model with minimized guards we measured the number of transitions, total size of generated guard conditions as the total number of nodes in parse trees of corresponding Boolean formulas, and the time used by the minimization algorithms. Experimental results are summarized in Table IV.

The conclusion is that the proposed exact guard conditions minimization method always produces smaller models compared to the heuristic algorithm. This was to be expected, since the guaranteed ability to find the minimal solution is the main qualitative advantage of the exact algorithm over the greedy heuristic one. Furthermore, the heuristic algorithm's execution time substantially increases with the size of the training set. This is natural, since its execution time is proportional to the number of performed iterations, which strongly depends on the size of the state machine and the number of transitions. Also, for the largest training set the exact algorithm is even faster than the heuristic one. Note that the execution time of the heuristic minimization algorithm increases monotonically with the size of the training set, while the time used by our CSP-based algorithm for the largest training set is slightly smaller than for the previous training set. This is not unusual, since both algorithms are heavily based on randomization and heuristics, therefore their execution time does not necessarily have to follow a strict monotonic pattern.

Similar logic is in place for the total guards sizes produced by both heuristic and CSP-based methods – in general, a larger training set corresponds to a larger total guards size. However, there is an exception: for the largest training set the resulting total guards sizes are smaller than for the previous 10×60 training set. Here, an additional factor is the two-staged nature of the suggested identification approach: the second stage (guard conditions inference) finds the minimal guard conditions, but with respect to the state machine found in

TABLE IV
EXPERIMENTAL RESULTS ON INFERRING FB MODELS WITH MINIMAL GUARD CONDITIONS

	Training set	Base model			Heuristic minimization			Proposed exact CSP-based minimization		
		# states	# transitions	guards size	# transitions	guards size	time, sec.	# transitions	guards size	time, sec.
simple tests	1	6	8	80	8	16	0.3	8	15	0.4
	10	8	21	168	17	41	1	17	36	2
	39	8	21	168	16	51	3	15	32	7
random tests	10×60 sec.	8	82	656	20	69	21	18	60	26
	20×60 sec. 20×120 sec.	8	79	632	17	65	98	15	47	23

the first stage (base model inference). Since the state machine constructed in the first stage is, in general, not unique, another state machine also consistent with the training set might correspond to a smaller total guards size. In order to overcome this limitation both the state machine and the minimal guard conditions need to be constructed in one step. However, this is a harder problem which lies outside the scope of this paper and will be subject of future work.

Quality of inferred models was assessed for state machines generated from the largest training set. First, we checked that the state machine conforms to a test set generated independently based on the legacy system. Next, we simulated the generated controller in closed loop with the plant within nxtSTUDIO. Therefore, we can conclude that synthesized models are behaviourally equivalent to the original system with respect to considered training and test sets. Smaller training sets were only used to study how well different algorithms scale with the amount of training data. Since experiments showed that our approach scales well, the amount of training data may be increased if needed. Overall, the proposed approach generates small state machines which can be used instead of the legacy controller.

An example of a model synthesized by the combination of SAT-based state machine inference and CSP-based guard conditions minimization methods is shown in Fig. 6, ECC-algorithms are described separately in Table V.

Note that this state machine is neither identical nor isomorphic to the original controller state machine of the legacy system shown in Fig. 7. This is not unusual, since synthesized models are almost always different from human-crafted solutions. On the one hand, the approach proposed in this paper infers a minimal solution based on a strict formalization of the problem. On the other hand, humans, though following the Occam's razor principle in general, tend to come up with more comprehensible solutions that might not be minimal.

As an example, consider state 1 of the generated state machine in Fig. 6 and state *Wait* of the legacy solution in Fig. 7. Both states have three outgoing transitions, dedicated to processing WPs arriving in input sliders 1, 2, and 3. Each transition from state *Wait* of the legacy automaton ensures that there are no WPs on input sliders with smaller numbers – for example, the transition to state *Piece2* checks not only `pp2 = True`, but also that `pp1 = False`. The transition priority function in state *Wait* (not shown in the figure) gives

TABLE V
ECC ALGORITHMS USED IN THE SYNTHESIZED MODEL IN FIG. 6

#	Output variable						
	c1Extend	c1Retract	c2Extend	c2Retract	vcExtend	vac_on	vac_off
1	$0 \to 0$, $1 \to 0$						
2		$1 \to 0$	$0 \to 1$	$1 \to 0$	$1 \to 0$		$1 \to 0$
3	$0 \to 1$ $1 \to 0$	$1 \to 0$	$0 \to 1$ $1 \to 0$	$1 \to 0$		$1 \to 0$	$1 \to 0$
4	$0 \to 1$	$1 \to 0$		$1 \to 0$	$1 \to 0$	$1 \to 0$	
5					$0 \to 1$ $1 \to 0$		
6					$0 \to 1$	$0 \to 1$	$1 \to 0$
7	$1 \to 0$	$0 \to 1$	$1 \to 0$	$0 \to 1$	$1 \to 0$		
8	$1 \to 0$	$0 \to 1$	$1 \to 0$	$0 \to 1$		$1 \to 0$	$0 \to 1$

the highest priority to the transition to state *Piece1*, the second priority is for the transition to *Piece2*, the smallest priority is for *Piece3*. Therefore, e.g., for the transition to *Piece2* it is redundant to check that `pp1 = False`, since this condition is guaranteed by the priority function. We can see that the automatically generated solution makes maximal use of the transition priority function and excludes unnecessary conditions from the guards.

VII. CONCLUSION

The paper suggests a black-box method for FB model identification from given behaviour examples in the form of execution traces. The proposed approach first infers a simple FB model using translation to SAT and then generalizes this model by inferring minimal-sized guard conditions using a translation to CSP. Experimental results indicate that the proposed approach scales well with the increase of the amount of training data.

Future work includes inferring a model minimal with respect to the number of states and total size of guard conditions with one step, studying other models of output algorithms, and other metrics of guard conditions complexity.

ACKNOWLEDGEMENT

The project is supported by the Russian Ministry of Science and Higher Education by the State Task No. 2.8866.2017/8.9, and also partially by RFBR according to the research project No. 16-37-00205 mol_a.

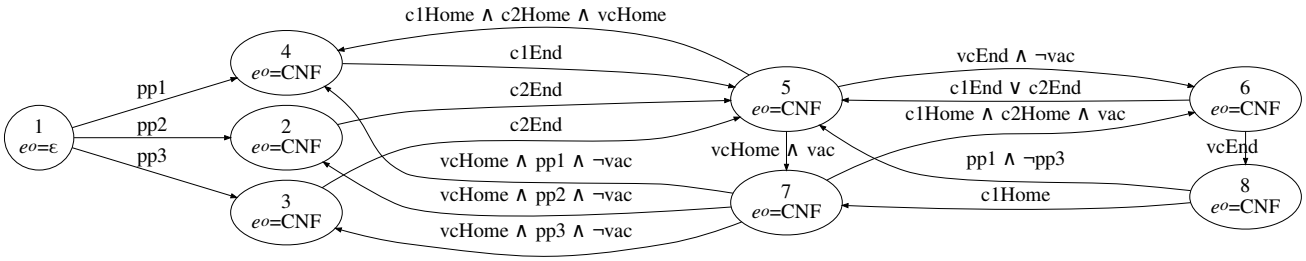


Fig. 6. Synthesized PnP controller model

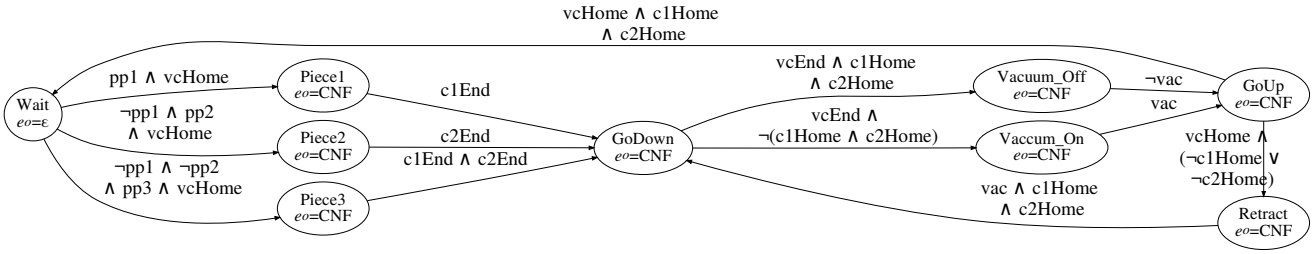


Fig. 7. Original state machine of the legacy PnP controller

REFERENCES

[1] L. Apfelbaum and J. Doyle, "Model based testing," in *Softw. Qual. Week Conf.*, 1997, pp. 296–300.

[2] 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.

[3] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, 2009.

[4] M. Shahbaz and R. Groz, "Analysis and testing of black-box component-based systems by inferring partial models," *Softw. Test. Verif. Reliab.*, vol. 24, no. 4, pp. 253–288, 2014.

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

[6] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 248–257.

[7] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 1, pp. 35–55, 2018.

[8] P. Faymonville, B. Finkbeiner, M. N. Rabe, and L. Tentrup, "Encodings of bounded synthesis," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2017, pp. 354–370.

[9] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. IOS Press, 2009.

[10] L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional satisfiability and constraint programming: A comparative survey," *ACM Comput. Surv.*, vol. 38, no. 4, 2006.

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

[12] C. Gerber, H.-M. Hanisch, and S. Ebbinghaus, "From IEC 61131 to IEC 61499 for distributed systems: a case study," *EURASIP J. Embedded Syst.*, vol. 2008, pp. 4:1–4:8, 2008.

[13] W. Dai and V. Vyatkin, "Redesign distributed PLC control systems using IEC 61499 function blocks," *IEEE Transactions on Automation Science and Engineering*, vol. 9, no. 2, pp. 390–401, 2012.

[14] M. Wenger, A. Zoitl, C. Sunder, and H. Steininger, "Transformation of IEC 61131-3 to IEC 61499 based on a model driven development approach," in *7th IEEE Conference on Industrial Informatics*. IEEE, 2009, pp. 715–720.

[15] D. Chivilikhin, V. Ulyantsev, A. Shalyto, and V. Vyatkin, "Csp-based inference of function block finite-state models from execution traces," in *IEEE 15th International Conference on Industrial Informatics*, 2017, pp. 714–719.

[16] V. Dubinin and V. Vyatkin, "Towards a formal semantic model of IEC 61499 function blocks," in *IEEE Int. Conf. Ind. Informat.*, 2006, pp. 6–11.

[17] N. Walkinshaw and M. Hall, "Inferring computational state machine models from program executions," in *IEEE Int. Conf. Softw. Maint. Evol.*, 2016, pp. 122–132.

[18] J. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992.

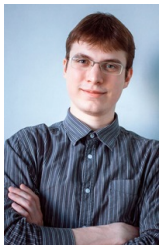
[19] I. Zakirzyanov, A. Shalyto, and V. Ulyantsev, "Finding all minimum-size DFA consistent with given examples: SAT-based approach," in *Software Engineering and Formal Methods*, A. Cerone and M. Roveri, Eds. Cham: Springer International Publishing, 2018, pp. 117–131.

[20] S. Patil, V. Vyatkin, and M. Sorouri, "Formal verification of intelligent mechatronic systems with decentralized control logic," in *IEEE Conf. Emerg. Technol. Factory Autom.*, 2012, pp. 1–7.

[21] G. Chu, "Improving combinatorial optimization," 2011, PhD thesis, University of Melbourne.

[22] Tools for extended finite-state machine synthesis and testing. [Online]. Available: https://github.com/ulyantsev/EFSM-tools

Daniil Chivilikhin received his B.Sc. and M.Sc. degrees in applied mathematics and informatics from ITMO University, St. Petersburg, Russia, in 2011 and 2013. In 2015 he defended his Ph.D. under the supervision of prof. Anatoly Shalyto in ITMO University, St. Petersburg, Russia. He is currently a researcher at ITMO University and works in the Computer Technologies Laboratory since 2013. His research interests include program synthesis and verification, industrial informatics and SAT solver applications.



Vladimir Ulyantsev Vladimir Ulyantsev received his B.Sc. and M.Sc. degrees in applied mathematics and informatics from ITMO University, St. Petersburg, Russia, in 2011 and 2013. In 2015 he defended his Ph.D. under the supervision of prof. Anatoly Shalyto in ITMO University, St. Petersburg, Russia. He is currently an Associate Professor (since 2015) and the head of the Computer Technologies Laboratory (since 2018) at ITMO University. His research interests include bioinformatics, evolutionary algorithms, combinatorial optimization and machine learning.

Anatoly Shalyto is a professor and a leading researcher at the department of computer technologies, faculty of IT and programming, ITMO University, St. Petersburg, Russia. His research mostly concerns automata-based programming. In particular, he introduced a methodology for automata-based programming called Switch-technology. In later years, his research is mostly dedicated to connection of automata-based programming to machine learning techniques for solving such problems as automata synthesis, testing and verification. The results of this research are currently used in a variety of Russian industrial companies. He is the author of a series of articles devoted to the problems of Computer Science and education in Russia. For his achievements in education, in 2008 he received a Russian State Government award.

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

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