CSP-based inference of function block finite-state models from execution traces

Chivilikhin D., Ulyantsev V., Shalyto A. and Vyatkin V.

Citation:

Chivilikhin D., Ulyantsev V., Shalyto A. and Vyatkin V. CSP-based inference of function block finite-state models from execution traces / In Proceedings of the 15th IEEE International Conference on Industrial Informatics, 2017, pp. 714-719

DOI: not yet available

CSP-based inference of function block finite-state models from execution traces

Daniil Chivilikhin*, Vladimir Ulyantsev* Anatoly Shalyto*, and Valeriy Vyatkin†‡*
*Computer Technologies Laboratory, ITMO University, Saint Petersburg, Russia
Email: chivdan@rain.ifmo.ru
†Department of Electrical Engineering and Automation, Aalto University, Finland
‡Department of Computer Science, Computer and Space Engineering, Lulea Tekniska Universitet, Sweden

***Abstract*—A method for inferring finite-state models of function blocks from given execution traces based on translation to the constraint satisfaction problem (CSP) is proposed. In contrast to the previous method based on a metaheuristic algorithm, the approach suggested in this paper is *exact*: it allows to find a solution if it exists or to prove the opposite. The proposed method is evaluated on the example of constructing a finite-state model of a controller for a Pick-and-Place manipulator and is shown to be significantly faster then the metaheuristic algorithm.**

I. Introduction

Finite-state models are one of the key concepts used in the process of software engineering in general, and, in particular, control systems development. Models are useful on different stages of software development: for testing [1], verification [2], and also in the context of reverse engineering of programs [3], [4]. However, 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 to generate visual models of existing software [5], [6] and infer models of dependable systems which satify given temporal properties [7], [8]. Active research in this area is stimulated by progress in discrete optimization methods such as metaheuristic algorithms [9], methods of Boolean satisfiability problem (SAT) [10] and constraint satifaction problem (CSP) [11] solving.

A practical example of using finite-state models in control systems engineering is the international standard for distributed automation systems development IEC 61499 [12] – control systems are developed in the form of a network of *function blocks* (FB) incapsulating control logic and data processing, finite-state machines are used as base elements of programs. It is often the case that the source code of the FB is unavailable and only an executable can be used. Therefore the problem of FB reverse engineering arises – to reconstruct the source code of the FB using data which can be acquired using the FB executable.

This work proposes a method of inferring finite-state models of FBs from examples of their behavior – so-called execution traces which can be derived using black-box testing of the FB. The proposed approach is based on translating FB model inference to CSP and using modern software for solving CSP.

The problem of FB model inference has been addressed before in [13], [14] with a method based on a metaheuristic algorithm MuACO. This algorithm is based on randomly mutating the current candidate solution to optimize a so-called fitness function that measures conformance of a solution to execution traces. The approach is essentially a randomized search algorithm and therefore there are no guarantees that it will eventually find a correct solution.

The approach proposed in this paper differs from the previous solution [13], [14] both qualitatively and quantitatively. First, on the contrary to the previous algorithm, the suggested approach is exact – if a solution exists, it will be eventually found, otherwise a proof of the absence of the solution will be produced. Second, computational experiments demonstrated that the proposed exact algorithm works significantly faster.

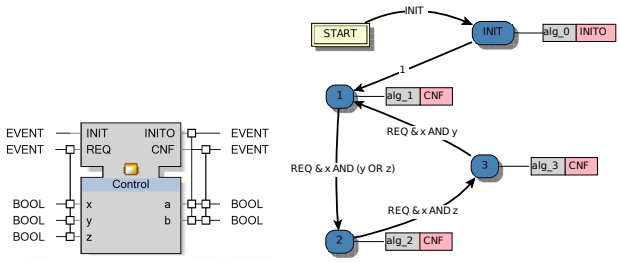
II. FB model inference problem statement

Control systems are often developed in the form of a network of interconnected FBs [12]. An FB is characterized by an *interface* – it defines input/output events and input/output variables. Variables can be, e.g., Boolean, integer, or real-valued. An example of an FB interface is shown in Fig. 1a: it has input events `INIT`, `REQ` and output events `INITO`, `CNF`, Boolean input variables *x*, *y*, *z* and output variables *a*, *b*.

IEC 61499 describes two main FB types: composite and basic blocks. A *composite* FB is a network of other FBs, either basic or composite whereas a *basic* FB is defined by a Moore finite-state machine called an *execution control chart* (ECC) which consists of states connected with transitions. Apart from input and output variables, an FB can also have internal variables. A transition of an ECC is labeled by a *guard condition* – a Boolean formula which can use input, output, and internal variables, logical operators and constants. In this work we focus on a smaller class of FBs with only Boolean input/output variables and no internal variables.

An ECC state can be associated with a list of *output actions*. An output action may contain an output event and an *algorithm*. Here and afterwards we will denote algorithms associated with ECC states as *ECC-algorithms*. They can be implemented using IEC 61131 languages (e.g. *Structured Text*) and are used for changing the values of output varibles. An example of an ECC is shown in Fig. 1b, ECC-algorithms are denoted as `alg_0..3`.

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



(a) FB interface example (b) ECC example

Fig. 1. Function block interface and execution control chart examples

A complete formal definition of an ECC can be found in [15]. Below we give a simplified definition for the case when (1) guard conditions depend only on input variables and (2) each state has exactly one associated output action. With respect to these assumptions, an ECC is an eight-tuple $\langle Y, E^{\mathrm{I}}, X, E^{\mathrm{O}}, Z, y_1, \varphi, g, \pi, \lambda \rangle$, where:

- $Y = \{y_1, y_2, \ldots, y_N\}$ is a finite set of *states*;
- $E^{\mathrm{I}} = \{e_1^{\mathrm{I}}, e_2^{\mathrm{I}}, \ldots\}$ is a set of input events;
- $X = \{x_1, x_2, \ldots\}$ is a set of Boolean input variables;
- $E^{\mathrm{O}} = \{e_1^{\mathrm{O}}, e_2^{\mathrm{O}}, \ldots\}$ is a set of output events;
- $Z = \{z_1, z_2, \ldots\}$ is a set of Boolean output variables;
- $y_1 \in Y$ is the initial state (w.l.o.g., $y_1 = 1$);
- $\varphi \subseteq Y \to Y$ is the transitions relation;
- $g \colon \varphi \to \left(E^{\mathrm{I}} \times \{0,1\}^{|X|} \to \{0,1\}\right)$ – defines for each transition a guard condition consisting of an input event and a Boolean formula over input variables;
- $\pi \colon \varphi \to \{1, 2, \ldots\}$ is the transition priority function;
- $\lambda \colon Y \to (\{0,1\}^{|Z|} \to \{0,1\}^{|Z|}) \times \left(E^{\mathrm{O}} \cup \{\varepsilon\}\right)$ is the output actions function which defines for each state a function over Boolean strings and an output event (ε corresponds to absence of an output event).

Hereinafter ECCs are called finite automata or *automata*. It is assumed that the target FB has a known interface but unknown implementation (ECC) and is a part of a closed-loop control system consisting of a controller and a plant. A simulation environment is available that allows to test the target FB and derive examples of its behavior – execution traces. The target FB is considered to be a "black box", therefore for execution traces derivation only *behavioral* or *black-box* testing can be used – a testing methodology which does not use knowledge about the internal structure of the system under test.

An *execution trace* is an ordered sequence of *trace elements* $s_i = \langle e^{\mathrm{I}}[\bar{x}], e^{\mathrm{O}}[\bar{z}] \rangle$, where each element consists of an input event $e^{\mathrm{I}} \in E^{\mathrm{I}}$, a list of input variable values $\bar{x} \in \{0,1\}^{|X|}$, an output event $e^{\mathrm{O}} \in E^{\mathrm{O}} \cup \{\varepsilon\}$, and a list of output variable values $\bar{z} \in \{0,1\}^{|Z|}$. The *length* of a trace is the number of its elements. An example of a set S of two traces S_1 and S_2 where $E^{\mathrm{I}} = \{A\}$, $E^{\mathrm{O}} = \{B\}$, $X = \{x_1, x_2\}$, $Z = \{z_1, z_2\}$ is shown in Fig. 2.

An automaton is said to *satisfy a trace element* $s_i = \langle e^{\mathrm{I}}[\bar{x}], e^{\mathrm{O}}[\bar{z}] \rangle$ in state y, if after receiving the input event e^{I} with input variable values $\bar{x}$ the automaton generates the output event e^{O} and the values of its output variables become equal to $\bar{z}$. Consequently, an automaton *satisfies a trace* s if it satisfies all its elements in corresponding states.

$$S_1 = \Big[A[x_1 = 1, x_2 = 0], B[z_1 = 1, z_2 = 0] \rangle;$$
$$\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 1] \rangle;$$
$$\langle A[x_1 = 0, x_2 = 0], B[z_1 = 0, z_2 = 1] \rangle \Big]$$
$$S_2 = \Big[\langle A[x_1 = 0, x_2 = 1], B[z_1 = 0, z_2 = 1] \rangle;$$
$$\langle A[x_1 = 0, x_2 = 0], B[z_1 = 1, z_2 = 1] \rangle;$$
$$\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 0] \rangle \Big].$$

Fig. 2. Example of scenarios

Now we formally state the problem addressed in this paper. The interface of the target FB (sets E^{I}, X, E^{O} and Z) is known and a set of traces $\mathbb{S}$ derived from the target FB is available. The problem is to find a finite-state model of the target FB which satisfies all traces from the given set $\mathbb{S}$ and has exactly N states.

III. Related work

The so-called extended finite-state machine (EFSM) is probably the finite-state model that is most similar to an ECC: its transitions are also labeled with input events and Boolean formulas over input variables, 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 based on translation to SAT [8], [16] as well as genetic [17] and ant colony optimization algorithms [18] have been proposed for inferring EFSMs from behavior examples.

The current work has two main differences from research on EFSM inference, which make existing methods inapplicable for the problem addressed in this paper. 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. Second, output actions in works on EFSM inference do not contain algorithms, they are merely names of algorithms which are assumed to be defined in the plant: an inference algorithm only needs to select which algorithms to run.

On the contrary, in the current work we infer the computatational algorithms themselves – in this way, the proposed method derives computational models. Computational EFSMs are also inferred in [19], however there a base finite-state model is assumed to be available beforehand – the method only aims at inferring algorithms for computing output variable values. Another issue is that this is done using genetic programming [20], which is not an exact approach and is not guaranteed to find a correct solution. In our work both

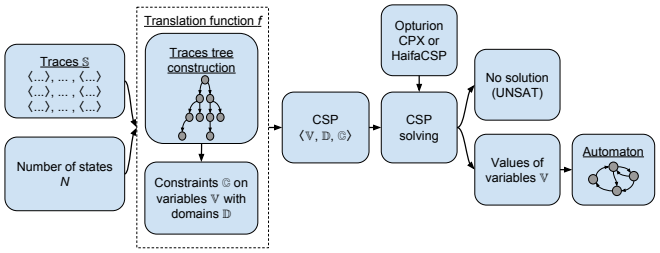


Fig. 3. Scheme of the proposed method

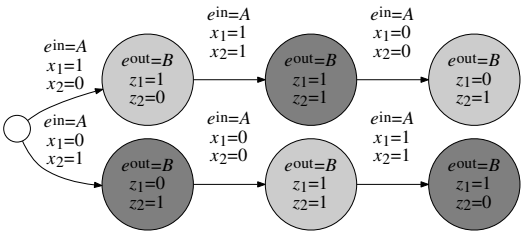


Fig. 4. Traces tree constructed from example traces (Fig. 2)

the automaton and the associated algorithms are inferred simultaneously.

IV. Proposed approach

The main idea of the proposed method is to *translate* the problem of inferring a finite-state model of an FB with a given number of states N from execution traces $\mathbb{S}$ to the problem of satisfying some *constraints* on integer variables (CSP). Denote the FB model inference problem $\langle \mathbb{S}, N \rangle$. The CSP is a triple $\langle \mathbb{V}, \mathbb{D}, \mathbb{C} \rangle$, where $\mathbb{V} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is the set of variables, $\mathbb{D} = \{D_1, D_2, \ldots, D_n\}$ is the set of variable *domains*, and $\mathbb{C} = \{C_1, C_2, \ldots, C_m\}$ is the set of constraints on the variables. A *solution* of a CSP is an assignment of variables which satisfies stated constraints.

The base of the proposed method is the *translation function* f, which transforms problem $\langle \mathbb{S}, N \rangle$ to problem $\langle \mathbb{V}, \mathbb{D}, \mathbb{C} \rangle$. The scheme of the suggested method is shown in Fig. 3. The method is divided into three main steps.

First, the input data of the FB model inference problem – set of traces $\mathbb{S}$ and the number of states N – are passed to the translation function f, which transforms the initial problem into a CSP. The translation works in two steps: traces tree construction, where a prefix (trie) is built from given traces, and construction of a set of constraints $\mathbb{C}$ on variables $\mathbb{V}$ with domains $\mathbb{D}$. Here, the traces tree and the sought automaton are encoded using integer variables $\mathbb{V}$ with domains $\mathbb{D}$, constraints $\mathbb{C}$ are placed on variables $\mathbb{V}$ that state that the sought automaton must satisfy given execution traces. Second, a third-party CSP solver is used to solve the constructed CSP. Third, if a solution of the CSP is found, the sought automaton is constructed from found values of variables $\mathbb{V}$. Otherwise the user is notified that a solution with N states does not exist. Below the steps of the method are described in detail.

A. Traces tree construction

A *traces tree* (E, V) is a prefix tree – a tree that contains all prefices of all traces. An edge $uv \in E$ $(u, v \in V)$ of the tree is marked with an input event from E^{I} and a list of input variable values from $\{0,1\}^{|X|}$. A vertex $v \in V$ of the tree is marked with an output event from E^{O} and a list of output variable values from $\{0,1\}^{|Z|}$. The tree constructed from example traces (Fig. 2) is shown in Fig. 4.

B. Problem representation and constraints

In this section we describe construction of the set of constraints $\mathbb{C}$ on integer variables: it contains main constraints $\mathbb{C}_{\mathbb{S}}$ which state that the sought automaton must satisfy given traces $\mathbb{S}$, and can also include auxiliary constraints.

1) Problem representation: The following integer variables are used to represent the traces tree:

- $e^{\mathrm{in}}_{uv} \in [1..|E^{\mathrm{I}}|]$ – index of input event on edge $uv \in E$;
- $e^{\mathrm{out}}_{v} \in [1..|E^{\mathrm{O}}|]$ – 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 $v \in V$.

Now we define the employed method of representing the sought automaton with integer variables. The main idea of the translation is to color the vertices of the traces tree in N colors in such a way, that the automaton produced by merging together all vertices with the same color satisfies given execution traces. Therefore, for each vertex on the traces tree v its *color* is introduced as $c_v \in [1..N]$.

Guard conditions on the transitions of the automaton are represented in the following way. Without loss of generality we can assume that guard conditions depend on all input variables: this allows to consider the lists of input variables values on the edges of the traces tree as symbols of some alphabet. We emphasize that this does not narrow the field of applicability of the proposed method since all lists of input variables that are present in traces will also be present in the constructed automaton. To represent guard conditions we calculate $\hat{X}$ – an ordered set of unique guard conditions present in the traces. Then guard conditions on the edges of the traces tree can be represented using variables $x_{uv} \in [1..|\hat{X}|]$ – for each edge uv the index of the corresponding guard condition in the ordered set $\hat{X}$ is used. For example, for scenarios (Fig. 2) we have $\hat{X} = \{\langle 0,0\rangle; \langle 0,1\rangle; \langle 1,0\rangle; \langle 1,1\rangle\}$, therefore values of x_{uv} are selected from the interval $[1..4]$. Also note that since guard conditions depend on all input variables the situation when two transitions originating from the same state simultaneously evaluate to `True` is impossible. Therefore we can avoid encoding the transition priority function π.

Transitions of the sought automaton are represented with variables $t_{n,e,x} \in [1..N]$, which encode the target state for a transition originating from state number n, marked with input event number e and guard condition number x.

Now we define the way of encoding the output actions function λ. It is assumed that the value of the i-th variable in state number n depends only from the previous value of this variable. Thus to represent the part of the output actions function which defines output variables modification it suffices to use two types of variables $d^0_{n,i}$ and $d^1_{n,i}$ which encode the value of the i-th output variable in cases when its previous value equals zero ($d^0_{n,i}$) or one ($d^1_{n,i}$). Output events are encoded with variables $o_n \in [1..(|E^{\mathrm{O}}|+1)]$, which represent the index of the output event ($|E^{\mathrm{O}}|+1$ corresponds to ε).

2) Constraints: In this section the main constraints $\mathbb{C}_{\mathbb{S}}$ are described which encode the requirement that the sought automaton must satisfy given execution traces. The following constraints place requirements on the initial state of the automaton:

- $c_1 = 1$ – the first vertex of the traces tree must correspond to the initial state of the automaton;
- $\bigwedge\limits_{1 \leqslant i \leqslant |Z|} \left(d^0_{1,i} = 0 \wedge d^1_{1,i} = 0\right)$ – the ECC-algorithm in the initial state unconditionally zeroes out all output variables;
- $o_1 = |E^{\mathrm{O}}| + 1$ – the initial state must not be associated with any output event.

The last two constraints are formulated because it is not always possible to include the initialization phase of the FB execution into the execution traces. Therefore we create a "dummy state" that simply zeroes out all output variables.

The following constraints require that the transition of the automaton leading from state c_u to state c_v is marked with input event e^{in}_{uv} and guard condition x_{uv}:

$$\bigwedge_{u,v \in V} \bigwedge_{uv \in E} \left(t_{c_u, e^{\mathrm{in}}_{uv}, x_{uv}} = c_v\right). \tag{1}$$

The output event in state c_v must correspond to the output event in the traces tree vertex v: $\bigwedge\limits_{v \in V} \left(o_{c_v} = e^{\mathrm{out}}_v\right)$.

If the value of the i-th output variable in vertex u equals zero, then the i-th element of the ECC-algorithm $d^0_{c_v,i}$ in state c_v must be equal to the value of the i-th output variable in vertex v (similar constraints are use when $z_{u,i} = 1$ for $d^1_{c_v,i}$):

$$\bigwedge_{u,v \in V} \bigwedge_{1 \leqslant i \leqslant |Z|} \left(z_{u,i} = 0 \implies d^0_{c_v,i} = z_{v,i}\right). \tag{2}$$

3) Auxiliary constraints: Described constraints $\mathbb{C}_{\mathbb{S}}$ are sufficient for constructing an automaton satisfying given execution traces. However note that not all possible automaton transitions and ECC-algorithm elements may be *covered* by the traces. This leads to some variables being *free* – no constraints are placed on them. Existence of free variables often negatively influences the efficiency of CSP solvers.

Therefore additional auxiliary constraints $\mathbb{C}_{\mathrm{loop}}$ are used for fixing the values of free variables. The following constraints require all transitions from state n labeled with event e and guard condition x such that the pair (e, x) is never encountered in traces to be self-loops (i.e. lead to state n):

$$\bigwedge_{1 \leqslant n \leqslant N} \bigwedge_{1 \leqslant e \leqslant |E^{\mathrm{I}}|} \bigwedge_{1 \leqslant x \leqslant |X|} ((\nexists uv \in E : e_{uv} = e \wedge x_{uv} = x \wedge x_u = n) \implies t_{n,e,x} = n).$$

If for some output variable j and state n there is no such trace tree edge uv that $c_v = n$ and $z_{u,j} = 0$, then $d^0_{n,j}$ must equal zero (similar constraints are used for $d^1_{n,j}$):

$$\bigwedge_{1 \leqslant j \leqslant |Z|} \bigwedge_{1 \leqslant n \leqslant N} \left(\nexists uv \in E : z_{u,j} = 0 \wedge c_v = n \implies d^0_{n,j} = 0\right).$$

In addition note that automata inference problems exhibit symmetry – for each automaton with N states there are $N!-1$ isomorphic automata, which differ only by the ordering of states. So-called *symmetry breaking* constraints can be used to limit the number of considered isomorphic solutions [21], [22]. In this work we use symmetry breaking predicates $\mathbb{C}_{\mathrm{bfs}}$ proposed in [22] for deterministic finite automata identification based on SAT solvers: they imply that the states must be numbered in the order they are visited by the breadth-first search (BFS) algorithm launched from the initial state. Overall, the final set of constraints $\mathbb{C}$ has the form: $\mathbb{C} = \mathbb{C}_{\mathbb{S}} \wedge \mathbb{C}_{\mathrm{loop}} \wedge \mathbb{C}_{\mathrm{bfs}}$.

C. CSP solving and automata construction

Constraints $\mathbb{C}$ described in the previous section were implemented in *MiniZinc* (www.minizinc.org) – a high-level language that allows to formulate CSPs regardless of the concrete CSP software used for solving them. The translation function f realizing construction of a CSP and reconstruction of the automaton from CSP solution was implemented as an open source *Java* application [23]. For solving CSP we employ two software tools *Opturion CPX* [24] and *HaifaCSP* [25].

If for the given number of states N a solution of the CSP does not exist, the CSP solver issues a corresponding message which we denote "UNSAT" (unsatisfiable). Otherwise, the sought automaton is constructed using found values of variables $\{c_v, t_{n,e,x}\}$ and the traces tree. The FB model constructed from the traces tree from Fig. 4 is shown in Fig. 5.

The initial state is marked with an incoming transition with no source. ECC state colors correspond to the colors of traces tree vertices. In each state an algorithm for updating each variable is written, an example of how algorithms are translated to *Structured Text* code is shown in Fig. 5.

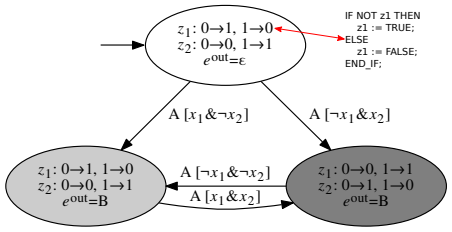
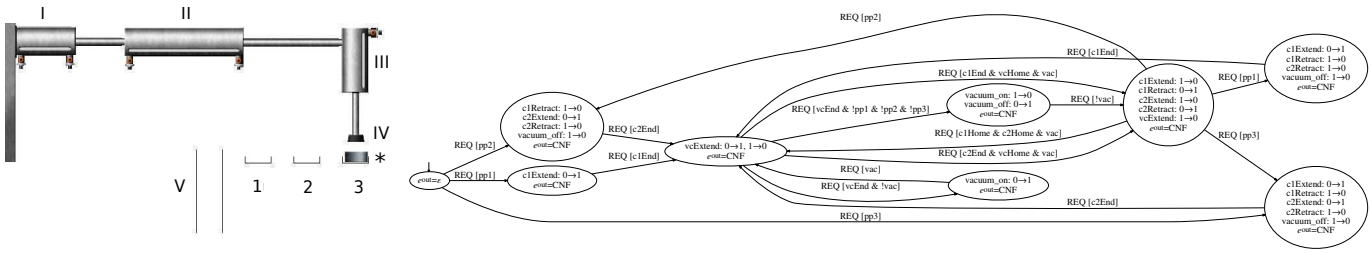


Fig. 5. FB model constructed from the traces tree from Fig. 4



(a) PnP manipulator

(b) Inferred PnP controller model example

Fig. 6. PnP system (left) and generated FB model (right)

V. Case study: reconstructing a controller FB for the Pick-and-Place manipulator

The proposed method was evaluated on the example of a control system for a Pick-and-Place (PnP) manipulator [26] shown in Fig. 6a. The studied PnP manipulator consists of two horizontal pneumatic cylinders (I, II), one vertical cylinder (III), and a suction unit (IV) for picking up work pieces (*). When a work piece appears on one of the input sliders (1, 2, 3) the horizontal cylinders position the suction unit on top of the work piece, the horizontal cylinder lowers the suction unit where it picks up the work piece and then moves it to the output slider. The control system is implemented using IEC 61499 in *NxtStudio* (www.nxtcontrol.com). The controller is a basic FB with 10 input and 7 output variables. The puprose of the experiments was to infer a model of this controller FB.

A. Traces generation for the PnP system

One of the simple ways to generate traces for a controller is to use random trace generation: on each step a random input event is issued and random values of controller input variables are chosen. However this method is not well-suited for systems with a large number of variables since random traces do not ensure sufficient coverage of possible FB behaviors. Furthermore, random traces in most cases do not correspond to any possible behaviors of the closed-loop system.

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 (work pieces processing) rather than the interface of the controller (events and variables). A *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 work pieces arrive. We assume that the manipulator returns to its initial position before the next work piece arrives.

The used set of tests 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. The set of all tests corresponds to a set of traces of length 150170. By saying that an automaton is constructed from tests we denote that it is constructed from traces derived from these tests.

B. Inferring automata with a fixed number of states

The purpose of the first set of experiments is to compare the proposed method and the previous approach based on a metaheuristic algorithm. The following algorithms are compared:

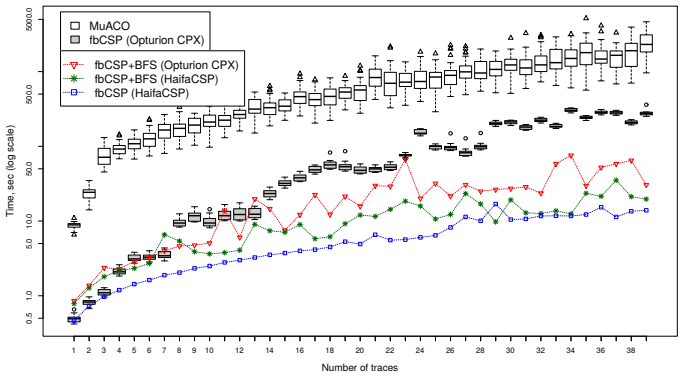


Fig. 7. Results of experimental runs of methods MuACO, fbCSP and fbCSP+BFS in dependence from the number of tests (traces) when $N = 10$

- MuACO: metaheuristic algorithm [13], [14];
- fbCSP: proposed method ($\mathbb{C} = \mathbb{C}_{\mathbb{S}} \wedge \mathbb{C}_{\text{loop}}$);
- fbCSP+BFS: proposed method ($\mathbb{C} = \mathbb{C}_{\mathbb{S}} \wedge \mathbb{C}_{\text{loop}} \wedge \mathbb{C}_{\text{bfs}}$).

Methods based on CSP have been tested together with both *Opturion CPX* and *HaifaCSP* solvers. A sequence of trace sets has been generated in a way that the i-th set of traces is built from the first i tests. Thus we measure the scalability of the methods with the increase of input date size.

The experiment for each algorithm was conducted in the following way. For each set of tests each algorithm was independently executed 30 times. For each constructed automaton its conformance to traces was checked. Also each automaton was checked in closed-loop simulation with the plant using *NxtStudio*. FB model inferred from 39 tests is shown in Fig. 6b.

In this experiment we fixed $N = 10$. A computer was used with an AMD Opteron™ 6378 @ 2.4 GHz processor limited to one core and 4 Gb of RAM. Fig. 7 depicts execution time of the algorithms: for some methods boxplots are presented, for others only mean execution times are plotted. This is due to some methods' execution times having such small deviation from the mean that the boxplots are not informative. Plotted times include the time for guard conditions generalization [13], [14], which less than two seconds per one automaton.

Variations of the proposed CSP-based method find a solution significantly faster than MuACO. Furthermore, they yield smaller deviation from the mean execution time. The

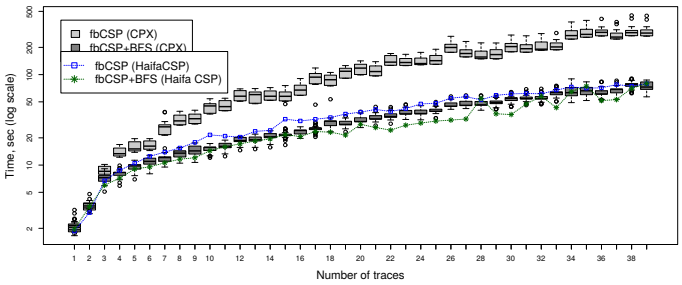


Fig. 8. Experimental results for minimal automata inference

use of BFS constraints significantly decreases the execution time of the CSP method using *Opturion CPX* for large sets of traces. However with *HaifaCSP* the BFS constraints most often increase the execution time. Overall, (1) the proposed CSP-based methods are significantly faster than MuACO (2) BFS constraints do not always decrease the execution time of the CSP-based method, (3) *HaifaCSP* tool is faster than *Opturion CPX* for the considered problems.

C. Inferring automata with a minimal number of states

As was mentioned before, the proposed method is an exact one – it allows to find a solution if it exists or to prove the opposite. This qualitative advantage allows to use the proposed method for finding an automaton with a *minimal number of states*. For that we start with $N = 1$ and in case of UNSAT increase N until the CSP solver produces a solution. Since the MuACO algorithm is not exact and cannot issue UNSATs, here only suggested methods are compared.

Experimental results are plotted in Fig. 8. The minimal number of states is: six for test set 1, seven for tests set 2, and eight for all other test sets. It is evident from the plots that usage of BFS constraints substantially decreases the time needed to find the minimal automaton. As in the first experiment, *HaifaCSP* works faster than *Opturion CPX*.

VI. Conclusion

In this paper we proposed a new CSP-based method for inferring finite-state models of function blocks from execution traces derived from black-box testing. The proposed method is superior to the previous metaheuristic approach both quantitatively (it works faster) and qualitatively – it is exact, which allows to use it for finding minimal automata.

Future work in this direction may include generalization of the method for supporting integer FB variables, timers, development of more general models of ECC-algorithms representation, and composite FB model inference.

Acknowledgements

This work was financially supported by the Government of Russian Federation, Grant 074-U01.

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 Trans. Ind. Informat.*, 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] M. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Softw. Eng.*, vol. 18, no. 4, pp. 825–856, 2013.
[6] 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.
[7] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 248–257.
[8] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *Int. Journ. Softw. Tools Techn. Transf.*, pp. 1–21, 2016.
[9] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 2003.
[10] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. IOS Press, 2009.
[11] L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional satisfiability and constraint programming: A comparative survey," *ACM Comput. Surv.*, vol. 38, no. 4, 2006.
[12] 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.
[13] D. Chivilkhin, A. Shalyto, S. Patil, and V. Vyatkin, "Reconstruction of function block logic using metaheuristic algorithm: Initial explorations," in *IEEE Int. Conf. Ind. Informat.*, 2015, pp. 1239–1242.
[14] D. Chivilikhin, A. Shalyto, S. Patil, and V. Vyatkin, "Reconstruction of function block logic using metaheuristic algorithm," *IEEE Trans. Ind. Informat.*, 2017.
[15] 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.
[16] V. Ulyantsev and F. Tsarev, "Extended Finite-State Machine Induction Using SAT-Solver," *IEEE Int. Conf. Machine Learning and Applications*, vol. 2, pp. 346–349, 2011.
[17] F. Tsarev and K. Egorov, "Finite state machine induction using genetic algorithm based on testing and model checking," in *Conf. Comp. Genetic Evol. Comput.* ACM, 2011, pp. 759–762.
[18] D. Chivilikhin and V. Ulyantsev, "MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines," in *Conf. Genetic Evol. Comput.*, 2013, pp. 511–518.
[19] N. Walkinshaw and M. Hall, "Inferring computational state machine models from program executions," in *IEEE Int. Conf. Softw. Maint. Evol.*, 2016, pp. 122–132.
[20] J. Koza, *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT Press, 1992.
[21] M. Heule and S. Verwer, "Exact DFA Identification Using SAT Solvers," in *Int. Colloquium Conf. on Grammatical Inference*, 2010, pp. 66–79.
[22] V. Ulyantsev, I. Zakirzyanov, and A. Shalyto, "BFS-Based Symmetry Breaking Predicates for DFA Identification," ser. Lecture Notes in Computer Science, 2015, vol. 8977, pp. 611–622.
[23] fbCSP. [Online]. Available: https://github.com/chivdan/cspgen
[24] Opturion CPX. [Online]. Available: http://www.opturion.com/cpx
[25] M. Veksler and O. Strichman, *Learning General Constraints in CSP*. Springer International Publishing, 2015, pp. 410–426.
[26] 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.