Counterexample-guided inference of controller logic from execution traces and temporal formulas

Daniil Chivilikhin*, Igor Buzhinsky*†, Vladimir Ulyantsev*,
Andrey Stankevich*, 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*—We developed an algorithm for inferring controller logic for cyber-physical systems (CPS) in the form of a state machine from given execution traces and linear temporal logic formulas. The algorithm implements an iterative counterexample-guided strategy: constraint programming is employed for constructing a minimal state machine from positive and negative traces (counterexamples) while formal verification is used for discovering new counterexamples. The proposed approach extends previous work by (1) considering a more intrinsic model of a state machine making the algorithm applicable to synthesizing CPS controller logic, and (2) using closed-loop verification which allows considering more expressive temporal properties.**

I. Introduction

Cyber-physical systems are widely used in many application domains such as manufacturing, aerospace industry and power plant engineering. Many applications of CPS are safety-critical, thus strong guarantees of CPS correctness are needed. CPS correctness is commonly checked by means of simulation testing – a simulation model of both plant and controller is created and tested. However, for safety-critical applications testing is insufficient – since it only considers finite-length behaviours, absense of errors while processing tests does not ensure that the system is correct. Therefore, formal verification must be used in addition to testing – it allows rigorously proving certain properties for a formal model of the system.

Design of a CPS controller under high dependability constraints is a tedious and time-consuming process. The common approach is to design control logic manually by keeping in mind the dynamics of the plant and knowledge about the environment. Since formal verification must be used in safety-critical applications, the manually designed controller should be verified. If verification discovers that the controller does not satisfy the formal specification, a counterexample is generated – a sequence of system states that leads to the violation of the checked property. Manual analysis of the counterexample and modification of the controller in such a way that this counterexample would be eliminated is a non-trivial process.

Eliminating the human engineer from the controller design process may speed it up and reduce the influence of the human factor. Thus, several works proposed methods for automatic inference of state machine controllers. The classical problem of inferring a minimal deterministic finite automaton that accepts/rejects given input words is known to be NP-hard [1] – no algorithm exists that would solve this problem in polynomial time unless P = NP. For this reason many state machine controller inference approaches are based on constraint programming [2] or propositional encoding [3] – instead of designing an algorithm for the initial problem, the latter is reduced (translated) to other well-known NP-hard problems, for which efficient complete decision procedures are known. Among such problems are Boolean satisfiability (SAT) and constraint satisfaction problem (CSP).

Authors of [4] proposed methods for constructing state machines from a set of linear temporal properties, however behavior examples are not supported. In [5] methods were proposed for inferring extended state machines from behavior examples and temporal properties, however they use a quite simple model of a state machine which is inefficient for modeling industrial CPS controllers. Work [6] also deals with a similar problem, however resulting finite-state machines are merely models and cannot be used as controllers themselves. This paper extends previous work [5], [7] on inferring state machine models. We developed a method that extends the Iterative SAT-based solution from [5] by considering a more expressive formal model of a state machine which allows applying the approach for inferring models of industrial-type controllers. The proposed method has been implemented and tested on the example of inferring a controller for a Pick-and-Place manipulator.

II. Problem statement

A. Function blocks

As a target implementation language in this work we use IEC 61499 function blocks (FBs) [8]. An FB is characterized by its interface which describes input/output events and variables. Only inference of basic FBs with only Boolean input/output variables is addressed in this work. A basic FB is determined by a Moore state machine called an *execution control chart*, where each transition is labeled by a guard condition – Boolean formula over input, output, and internal variables. Here we assume for simplicity that only input variables may be used in a guard condition. A state is associated with a list of output actions – an output event and output algorithm, which can be used for modifying values of output

978-1-5386-7108-5/18/$31.00 ©2018 IEEE

variables. Again, for simplicity it is assumed that each state is associated with only one output action.

B. State machine model and execution traces

The target state machine is described by a set of N states, 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 a transition to the next state, which is marked with an input event and a guard condition (a Boolean function over input variables) and (2) an output function that for each state defines transformation of output variables values and issuing of an output event.

The first type of input data used in this paper for state machine inference is a set of execution traces. A (positive) *execution trace* is a sequence of *trace elements* $s_i = \langle e^{\text{in}}[\bar{x}], e^{\text{out}}[\bar{z}] \rangle$, where each element consists of an input event $e^{\text{in}} \in E^{\text{in}}$, an input variable values tuple $\bar{x} \in \{0,1\}^{|X|}$, an output event $e^{\text{out}} \in E^{\text{out}} \cup \{\varepsilon\}$ (ε denotes "no output event"), and an output variable values tuple $\bar{z} \in \{0,1\}^{|Z|}$. Below is an example of two execution traces where $E^{\text{in}} = \{A\}$, $E^{\text{out}} = \{B\}$, $X = \{x_1, x_2\}$, $Z = \{z_1, z_2\}$:

$$\begin{aligned} t_1 = \Big[&\langle A[x_1 = 1, x_2 = 0], B[z_1 = 1, z_2 = 0] \rangle; \\ &\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 1] \rangle; \\ &\langle A[x_1 = 0, x_2 = 0], B[z_1 = 0, z_2 = 1] \rangle \Big] \\ t_2 = \Big[&\langle A[x_1 = 0, x_2 = 1], B[z_1 = 0, z_2 = 1] \rangle; \\ &\langle A[x_1 = 0, x_2 = 0], B[z_1 = 1, z_2 = 1] \rangle; \\ &\langle A[x_1 = 1, x_2 = 1], B[z_1 = 1, z_2 = 0] \rangle \Big]. \end{aligned} \quad (1)$$

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 $\bar{x}$ it generates output event e^{out} and the values of its output variables become equal to $\bar{z}$. An automaton *satisfies a trace* T if it satisfies all its elements in corresponding states, starting from the initial state.

C. Temporal properties and formal plant model

The second type of input data used in this paper is a set of LTL properties that the target state machine should satisfy. An LTL formula may include problem dependent propositional variables (here – input/output variables or predicates assembled from them), Boolean connectors $\vee, \wedge, \neg, \rightarrow$ and a set of temporal operators, for example:

- $\mathbf{G}\, f$ – f holds for all states starting from the current one;
- $\mathbf{X}\, f$ – f holds for the next state;
- $\mathbf{F}\, f$ – f holds for some consequent state.

For example, if x_1 and z_2 are propositional variables, then $\mathbf{G}(x_1 \rightarrow \mathbf{F}(z_2))$ is a LTL formula meaning "Always: if x_1 then eventually z_2".

We assume that the target state machine should act as a controller in a closed-loop system together with the plant. If a formal plant model is available, it can be used to widen the class of LTL properties we can use to synthesize controllers. Such a formal model can be either created manually or inferred automatically [9], [10]. Then we can use closed-loop properties that are only meaningful in the presence of the plant model. If the plant model is not available, we will not be able to use properties such as "Always: if some signal is received, some actuator is triggered" since values of input variables of the controller will not be constrained by the plant model and will thus be arbitrary.

D. Problem statement

With respect to the above definitions, the problem addressed in this paper is formulated as follows: given a set of (positive) execution traces, a set of temporal properties, and optionally a formal model of the plant, find a state machine model for the controller which has a minimal number of states and transitions and satisfies both given execution traces and temporal properties.

III. Related work

One partial solution of the stated problem has been developed in [11] by means of a metaheuristic algorithm. The algorithm used a randomized process to generate candidate solutions (state machines), which were tested against positive traces and verified against given LTL properties using NuSMV [12]. The issues with this approach are that it is (1) not complete, since there are no guarantees that the algorithm will ever terminate and find a correct solution and (2) it requires a very large number of calls to the model checker. Several complete solutions for a similar problem for inferring similar and yet still different and simpler state machine models have been developed in [5].

IV. Proposed approach at a glance

The pseudocode of the proposed approach is shown in Algorithm 1. Two main data structures are maintained: positive traces tree T^+, which is built only once from given positive traces, and negative traces tree T^-, which is initially empty and is updated on each iteration. The algorithm starts from the initially given positive traces and uses constraint programming to construct a model that fits the positive traces and has a minimal number of states (function `findModel`). Then, the current model is verified with NuSMV in closed loop with the provided plant model against given LTL properties (function `verify`). If all properties are satisfied the algorithm terminates and returns the constructed state machine model. Otherwise, the counterexample(s) C returned by the model checker are added to the negative traces tree. Then, the constraint programming method is executed again, ensuring that the generated solution not only conforms to the positive traces, but also does not exhibit behavior described by any negative trace. The main contribution of this paper is the adaptation of the iterative SAT-based algorithm from [5], which has initially been designed for building protocol-like models, to inferring industrial controller logic models. One of the key differences is that models inferred by the proposed approach are less explicit

Algorithm 1: Iterative FB model inference

```
Data: positive traces tree $T^+$, temporal properties $f$
$T^- \leftarrow \varnothing$            // negative traces tree
$N \leftarrow 2$, $R \leftarrow 1$
while True do
    $A \leftarrow$ findModel($T^+, T^-, N, R$)
    if $A = \varnothing$       // solver returned UNSAT
     then
        if $R \leq 2N + 1$ then  $R \leftarrow R + 1$
        else  $N \leftarrow N + 1$
    else
        $C \leftarrow$ verify($A$, $f$)
        if $C = \varnothing$ then  return $A$
        else  $T^-$.add($C$)
```

and more symbolic in comparison with models inferred by methods from [5], both in terms of transitions and states.

First, in [5] a transition is guarded with a so-called input event, which, in terms of the considered problem, can be interpreted as a tuple of values of all input variables. This approach is practically applicable if the number of such input events is adequately small, e.g. 10–20. However, even small examples of controllers such as the controller for the Pick-and-Place manipulator system considered in Section VI can have 10–20 input variables, which leads to a maximum of 2^{10}–2^{20} input events! It is obvious that such explicit handling of guard conditions will not scale to large systems, and a more symbolic approach is needed. Therefore, in this paper we try to overcome this by inferring close-to-minimal guard conditions in which only a few input variables are used.

The second difference from [5] is in a more symbolic way of handling outputs of the state machine. While in [5] an output corresponds to setting explicit values of all output variables, in this work, as well as in [7], we use symbolic output algorithms that transform values of output variables.

The remainder of the paper is devoted to the description of the findModel(T^+, T^-, N, R) function, which tries to find a finite-state model with N states and R transitions that conforms to given trees of positive traces T^+ and negative traces T^-. In this work the problem of model construction is solved by reduction to CSP. In the following sections we formally describe the proposed approach, starting with state machine model representation.

V. Reduction to CSP

The main idea of constraint programming is to reduce the addressed problem to solving a system of constraints on (integer and Boolean) variables. Due to the recent progress in operations research, modern CSP solvers can solve practical problems with millions of such constraints in seconds. In order to solve a problem by means of a CSP solver, we need to (1) model the solution and input data with integer variables and (2) develop a set of constraints such that a valid solution of these constraints would correspond to a valid solution of the initial problem.

A. Controller logic model

Our model consists of a set of N states connected with R transitions. To represent transitions we use integer variables: s_r for the source state and d_r for the destination state of the r-th transition ($s_r, d_r \in [1..N]$). Each transition is labeled with a guard condition – Boolean formula restricted to a conjuction of at most M literals, where each literal is either an input variable or its negation, e.g. $x_1 \land x_2 \land \neg x_7$ or $\neg x_4 \land x_6$. To represent guard conditions in our constraint model we use two types of variables: integer variables $g_{r,m}$ for representing the input variable and Boolean variables $\alpha_{r,m}$ for the literal sign. For example, if $x_1 \land x_2 \land \neg x_7$ is the guard condition on the first transition, it is represented by $g_{1,1} = 1$, $g_{1,2} = 2$, $g_{1,3} = 7$ and $\alpha_{1,1} = 1$, $\alpha_{1,2} = 1$, $\alpha_{1,3} = 0$.

Output events are represented with variables o_c. Output algorithms are encoded with variables $a_{c,j} \in \{0, 1, \text{“x”}\}$, where values zero and one indicate that when entering state c, the j-th output variable must be set to 0 or 1 resprectively. If the value of the output algorithm element is “x”, then the corresponding output variable is left unchanged.

All necessary constraints can be classified into three groups: positive tree constraints, negative tree constraints, and auxiliary constraints.

B. Positive traces tree construction

First, given positive traces are used to construct the positive traces tree T^+ – a prefix tree that contains all prefixes of all positive traces. Denote $E(T^+)$ the set of tree edges and $V(T^+)$ the set of tree vertices. Edge $uv \in E(T^+)$ ($u, v \in V(T^+)$) of the tree is marked with an input event from E^{I} and a tuple of input variable values from $\{0,1\}^{|X|}$. A vertex $v \in V(T^+)$ of the tree is marked with an output event from E^{O} and a tuple of output variable values from $\{0,1\}^{|Z|}$. The positive tree is described with the following integers:

- $e^{\text{in}}_{uv} \in [1..|E^{\text{in}}|]$ – input event on edge $uv \in E(T^+)$;
- $e^{\text{out}}_{v} \in [1..|E^{\mathrm{O}}|]$ – output event in vertex $v \in V(T^+)$;
- $l_{v,j} \in \{0, 1\}$ – value of the j-th input variable on edge (u, v), where u is the parent of v;
- $k_v \in [1..|U|]$ (u – parent of v) – index of the input variables values tuple on the edge (u, v) in the ordered set of such tuples U;
- $z_{v,i} \in \{0, 1\}$, $i \in [1..|Z|]$ – value of i-th output variable in vertex $v \in V(T^+)$.

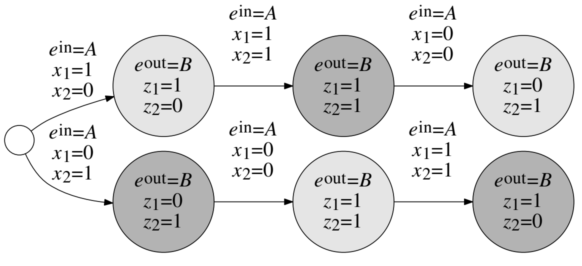
In addition, $p(v)$ denotes the direct parent of vertex v and $p_a(v)$ is the closest node to v in the upwards direction that has a non-ε output event. The tree constructed from example traces (Fig. 1) is shown in Fig. 1.

C. Positive tree constraints

The main idea here is to color the vertices of the positive tree T^+ using $N + 1$ colors: vertices with colors ranging from 1 to N are mapped to a corresponding state of the resulting automaton, and vertices colored with $N + 1$ are not mapped

TABLE I
POSITIVE TREE CONSTRAINTS

	Constraint	Domain
1.1	$s_1 = 1$	–
1.2	$s_r \leq s_{r+1}$	$1 \leq r < R$
1.3	$(s_r = s_{r+1}) \rightarrow (d_r \leq d_{r+1})$	$1 \leq r < R$
1.4	$\#_n^{\text{tran}} = \sum\limits_{1 \leq r \leq R} I(s_r = n)$	$1 \leq n \leq N$
1.5	$\#_n^{\text{tran}} > 0$	$1 \leq n \leq N$
1.6	$\#_r^{\text{var}} = \sum\limits_{1 \leq m \leq M} I(g_{r,m} \leq \lvert X \rvert)$	$1 \leq r \leq R$
1.7	$(m > \#_r^{\text{var}}) \rightarrow \neg \alpha_{r,m} \wedge (g_{r,m} = \lvert X \rvert + 1)$	$1 \leq r \leq R,\ 1 \leq m \leq M$
1.8	$(g_{r,m} \leq \lvert X \rvert) \rightarrow (g_{r,m+1} > g_{r,m})$	$1 \leq r \leq R,\ 1 \leq m < M$
1.9	$(g_{r,m} = \lvert X \rvert + 1) \rightarrow (g_{r,m+1} = \lvert X \rvert + 1)$	$1 \leq r \leq R,\ 1 \leq m < M$
2.1	$\varphi_{v,r} \leftrightarrow \bigwedge\limits_{m} \left(m \leq \#_r^{\text{tran}} \rightarrow l_{v,g_{r,m}} = \alpha_{r,m}\right)$	$1 \leq v \leq \lvert U \rvert,\ 1 \leq r \leq R,\ 1 \leq m \leq M$
2.2	$\varphi'_{v,1} \leftrightarrow \varphi_{v,1}$	$1 \leq v \leq \lvert U \rvert$
3.1	$(s_r = s_{r-1}) \rightarrow (\varphi'_{v,r} \leftrightarrow \varphi_{v,r} \wedge \overline{\varphi}_{v,r-1})$	$2 \leq r \leq R,\ 1 \leq v \leq \lvert U \rvert$
3.2	$(s_r > s_{r-1}) \rightarrow (\varphi'_{v,r} \leftrightarrow \varphi_{v,r})$	$2 \leq r \leq R,\ 1 \leq v \leq \lvert U \rvert$
4.1	$\overline{\varphi}_{v,1} \leftrightarrow \neg \varphi_{v,1}$	$1 \leq v \leq \lvert U \rvert$
4.2	$(s_r = s_{r-1}) \rightarrow (\overline{\varphi}_{v,r} \leftrightarrow \neg \varphi_{v,r} \wedge \overline{\varphi}_{v,r-1})$	$2 \leq r \leq R,\ 1 \leq v \leq \lvert U \rvert$
4.3	$s_r > s_{r-1} \rightarrow (\overline{\varphi}_{v,r} \leftrightarrow \neg \varphi_{v,r})$	$2 \leq r \leq R,\ 1 \leq v \leq \lvert U \rvert$
5.1	$\bigvee\limits_{1 \leq r \leq R} (c_{p_a(v)} = s_r) \wedge (c_v = d_r) \wedge \varphi'_{k_v,r}$	$v \in \left\{V(T^+) \mid e_v^{\text{out}} \neq \varepsilon\right\}$
5.2	$(s_r = c_{p_a(v)}) \rightarrow \neg \varphi_{k_v,r}$	$1 \leq r \leq R,\ v \in \left\{V(T^+) \mid e_v^{\text{out}} = \varepsilon\right\}$
6.1	$(c_1 = 1) \wedge (o_1 = \texttt{INITO})$	–
6.2	$c_v = N + 1$	$v \in \{V(T^+) \mid e_v^{\text{out}} = \varepsilon\}$
6.3	$(a_{1,j} = 0) \wedge (a_{N+1,j} = 0)$	$1 \leq j \leq L$
6.4	$c_v \neq 1 \rightarrow (o_{c_v} = e_v^{\text{out}})$	$v \in \left\{V(T^+) \mid e_v^{\text{out}} \neq \varepsilon\right\}$
6.5	$(a_{c_v,j} = z_{v,j}) \vee (a_{c_v,j} = \text{“x”})$	$v \in \left\{V(T^+) \mid e_v^{\text{out}} \neq \varepsilon \wedge z_{p_a(v),j} = z_{v,j}\right\},\ 1 \leq j \leq \lvert Z \rvert$
6.6	$a_{c_v,j} = z_{v,j}$	$v \in \left\{V(T^+) \mid e_v^{\text{out}} \neq \varepsilon \wedge z_{p_a(v),j} \neq z_{v,j}\right\},\ 1 \leq j \leq \lvert Z \rvert$



e^{in}=A x_1=1 x_2=0
e^{in}=A x_1=0 x_2=1
e^{out}=B z_1=1 z_2=0
e^{out}=B z_1=0 z_2=1
e^{in}=A x_1=1 x_2=1
e^{in}=A x_1=0 x_2=0
e^{out}=B z_1=1 z_2=1
e^{out}=B z_1=1 z_2=1
e^{in}=A x_1=0 x_2=0
e^{in}=A x_1=1 x_2=1
e^{out}=B z_1=0 z_2=1
e^{out}=B z_1=1 z_2=0

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

to any state. To represent colors of vertices we use integer variables $c_v \in [1..(N+1)]$, $v \in V(T^+)$. Without loss of generality, the root node is assigned color 1. In this section we describe the constraints verbally, formal definitions are given in Table I. In the table I is a function such that $I(\texttt{False}) = 0$, $I(\texttt{True}) = 1$.

Recall that transitions are encoded using variables s_r and d_r. This representation was chosen because it allows to efficiently limit the number of transitions, thus proceeding from small automata with less transitions to larger automata with more transitions. In order to break possible symmetries, we add monotonicity constraints to ensure that (1) source states form a non-decreasing sequence, and (2) for equal values of sources s_r corresponding destinations d_r should also form a non-decreasing sequence. These constraints are formalized in section 1 of Table I.

Guard conditions on transitions are represented with variables $g_{r,m}$ and $\alpha_{r,m}$. We also need some means to encode whether a guard condition of a transition is satisfied for specific values of input variables given by an edge of T^+ – thus Boolean variables $\varphi_{v,r}$ are introduced with corresponding constraints. Furthermore, transtion execution semantics of a basic function block are accounted for: satisfiability of guard conditions on transitions is checked in the specified order of transition priority. In other words, the i-th transition is triggered if and only if its guard condition is satisfied while all guard conditions of previous transitions from this state are falsified. To encode this property two new types of Boolean variables are introduced: $\overline{\varphi}_{v,r}$ for a falsified guard condition and $\varphi'_{v,r}$ for indicating that only the r-th guard condition is satisfied. Corresponding constraints are formalized in sections 2–4 of Table I. Instead of adding guard conditions constraints for all input tuples given by positive tree edges, we consider all unique combinations of input variable values denoted by set U.

Now we can describe the main positive tree constraints. First, it is required that for all positive tree vertices with a non-ε output event there exists a corresponding triggered transition. Second, for each tree vertex with an ε output event there should be no triggered transitions. These constraints are formalized in section 5 of Table I.

Finally, constraints on output algorithms are formulated for each tree edge on the basis of values of output variables (section 6 of Table I).

D. Negative tree constraints

Negative tree constraints used in this paper are inspired by the Iterative SAT-based approach from [5], but are adapted to the more complex industrial controller logic state machines. The purpose of the constraints is to ensure that the resulting automaton will not exhibit behaviors described by the previously discovered counterexamples stored in the negative tree. The approach is based on the fact that any counterexample to an LTL property can be represented in lasso-shaped form as a finite prefix followed by a cycle [13].

The negative tree is analogous to the positive tree in all aspects except one – it may contain non-deterministic transitions: a node can have several outgoing edges with the same input label. This is because the negative tree represents how the system should not behave rather than how it should.

The approach to counterexample prohibition suggested in [5] is based on the idea of *propagating* colors of negative tree vertices along the tree on the basis of the model of the state machine.

1) The root of the tree corresponds to the initial state of the automaton.
2) For each edge, if a transition corresponding to this edge exists, then the child vertex of the edge is colored.
3) Finally, to prohibit the counterexample it suffices to state that the start and end nodes of the cycle have different colors.

Our approach mainly differs from [5] in the implementation of step 2, colors propagation. This is due to a more complex model of the state machine considered in this paper.

To account for satisfaction of guard conditions on negative tree edges we introduce variables $\psi_{v,r}$, $\overline{\psi}_{v,r}$, and $\psi'_{v,r}$ ($v \in V(T^-)$, $r \in [1..R]$) which are analogous in every way to corresponding positive tree variables $\varphi_{v,r}$, $\overline{\varphi}_{v,r}$, and $\varphi'_{v,r}$. Integer variables $n_v \in [1..(N+1)]$ ($v \in V(T^-)$) represent colors of negative tree vertices. In addition, Boolean variables π_v ($v \in V(T^-)$) are introduced indicating whether a vertex corresponds to a possible behavior of the state machine or not.

The negative tree is described with integers in the same way as the positive tree using analogous variables $\overline{e^{\text{in}}}(v)$, $\overline{e^{\text{out}}}(v)$, $\overline{z}_{v,i}$, $\overline{l}_{v,j} \in \{0,1\}$ and $\overline{k}_v \in [1..|U|]$, where $\overline{U}$ is the ordered set of all unique input variable values combinations present on edges of T^-. In addition, additional parameters $\text{loop}(v) \in [0..|T^-|]$ are introduced for storing lasso-shaped counterexample back-edges: $\text{loop}(v) \neq 0$ indicates the vertex with which v is connected via a back-edge. Analogous to the positive tree, $p(v)$ is the direct parent of vertex $v \in T^-$ and $p_a(v)$ is the closest node to v in the upwards direction that has a non-ε output event.

First of all, without loss of generality, we state that the negative tree root corresponds to the initial state of the automaton and that the root corresponds to a possible behavior of the automaton ($\pi_1 = \top$). Different constraints are placed on nodes with non-ε output events (these correspond to a fired transition) and with ε output events (these correspond to a situation when no transition should be fired).

1) Constraints on nodes with an output event: In the case of a non-ε output event, consider a negative tree edge (u, v) and the corresponding node $p_a(v)$. The following constraint encodes the situation when a color should be propagated to vertex v with $\overline{e^{\text{out}}}(v) \neq \varepsilon$. If:

- node $p_a(v)$ has color c_1 and corresponds to a possible behavior ($\pi_{p_a(v)} = \top$);
- there exists a transition from c_1 to c_2 such that its guard condition is satisfied for $\overline{l}_{uv}$;
- output events in state c_2 and node v are equal;
- the algorithm in state c_2 allows to produce output variable values $\overline{z}_{v,i}$ stored in the tree,

then node v also corresponds to a possible behavior of the model ($\pi(v) = \top$) and its color is c_2 ($n_v = c_2$).

The following constraint encodes the situation when a color should not be propagated to vertex v with $\overline{e^{\text{out}}}(v) \neq \varepsilon$. If:

- node $p_a(v)$ has color c_1 and corresponds to a possible behavior ($\pi_{p_a(v)} = \top$);
- either there is no transition such that its guard condition is satisfied for $\overline{l}_{uv}$;
- or a transition exists, but leads to a state with an algorithm that does not produce correct values of output variables,

then node v does not correspond to a possible behavior ($\pi(v) = \bot$).

2) Constraints on nodes without an output event: Nodes without an output event correspond to a situation when no transition should be triggered in the state machine. The following constraint encodes the situation when color c_1 should be propagated to v. If:

- node $p_a(v)$ has color c_1 and corresponds to a possible behavior ($\pi_{p_a(v)} = \top$);
- there is no transition such that its guard condition is satisfied for $\overline{l}_{uv}$,

then node v also corresponds to a possible behavior of the model ($\pi(v) = \top$) and its color is c_1.

For the case of not propagating a color to v the following constraint is formulated. If:

- node $p_a(v)$ has color c_1 and corresponds to a possible behavior ($\pi_{p_a(v)} = \top$);
- there exists a transition from c_1 such that its guard condition is satisfied for l_{uv},

then node v does not correspond to a possible behavior ($\pi(v) = \bot$).

3) Final negative tree constraints: In addition, if a parent of a node $p(v)$ or its closest non-ε node $p_a(v)$ do not correspond to a possible behavior, then v also does not. A node with possible behavior must also have a valid color: $n(v) \neq N+1$. Finally, to prohibit a counterexample, if a node has a defined back-edge ($loop(v)$) and both nodes v and $loop(v)$ correspond to a possible behavior of the system, their colors should be different.

E. Auxiliary constraints

In addition to the described constraints we use additional auxiliary symmetry breaking constraints proposed in [14] aimed at reducing the search space size. To enforce the states of the automaton to be enumerated in the order they would be visited by a breadth-first search algorithm started from the initial state. This allows excluding isomorphic state machine from the search space and greatly reduces the time necessary to prove unsatisfiability.

Moreover, to further reduce symmetry we use optimized Minizinc built-in constraints to enforce numbers of input variables in guard conditions on transitions from the same state to be sorted lexicographically: if $s_r = s_{r+1}$ then g_r is lexicographically smaller or equal to g_{r+1}, where g_r denotes $g_{r,1}, \dots, g_{r,M}$.

F. Optimization criterion

Since CSP-solvers allow to solve not only satisfiability but also minimization problems, we use this feature to reduce the number of explicit numerical parameters of the problem. Each launch of the solver aims to minimize the sum of (1) total number of input variables used in guard conditions and (2) total number of non-“x” output algorithm elements:

$$\left(\sum_{1 \le r \le R} \#_r^{\text{var}} + \sum_{1 \le c \le N} \sum_{1 \le j \le L} I(a_{c,j} \neq \text{“x”}) \right) \to \min .$$

G. Estimating initial values of N and R

The main integer parameters of the of the described constraint system are the numbers of states N and transitions R, their minimal values are $N = 2$, $R = 1$. However, starting the algorithm from these minimal values often leads to large execution times. Instead, we use the approach from [7] that quickly constructs a minimal automaton consistent with positive traces T^+ only. Then we employ the guard condition generalization algorithm used in [7] that heuristically tries to decrease the number of input variables used in guard conditions and, in some cases, also the number of transitions. Parameters of the resulting automaton are used as initial estimates for N and R in Algorithm 1. However, before proceeding with the algorithm we try to decrease parameter R until the first “UNSAT” is generated.

H. Search space reduction using assumptions

In order to speed up the algorithm an additional heuristic is used that allows reducing search space size on each iteration. The heuristic is applied at iteration i if the previous iteration $i-1$ resulted in new counterexamples. The idea is to fix values of output algorithm variables $a_{c,j}$ found at iteration $i-1$ while solving CSP at iteration i – *assume* that they are preserved. If the resulting problem is satisfiable, this leads to considerably smaller solver run times. If, on the other hand, the solver returns “UNSAT”, then the assumption is removed and the solver is run again.

I. Implementation

Described constraints were encoded using the Minizinc [15] modeling language. The developed method was implemented in the form of a stand-alone Java application that (1) reads input traces and LTL properties, (2) builds the initial positive traces tree and encodes it using integer variables, (3) interacts with the CSP solver by supplying it with input data and parsing the output, (4) interacts with the NuSMV model checker and incrementally builds the negative traces tree from counterexamples generated on each iteration.

We experimented with several award-winning CSP-solvers and found that the Chuffed [16] solver works best for our data. Our implementation generates the state machine model in three different output formats: (1) textual representation visualizable with GraphViz [17], (2) NuSMV module containing the model of the generated controller and the provided plant model, and (3) an XML file of the generated basic FB readable by NxtStudio [18] which can be directly plugged in and tested in closed-loop simulation with the plant.

VI. Case study system: Pick-and-Place manipulator

The proposed method was tested on the example of generating a controller for the Pick-and-Place manipulator system [19] shown in Fig. 2. The system is designed for moving the work pieces (*) that arrive to input sliders (1, 2, 3) to the output slider (V). This is done using three pneumatic cylinders (I, II, III) that are used to move the vacuum unit (IV) to the designated position above a specific input slider. The work piece (WP) is then captured by the vacuum unit and moved to the output slider. The control logic of the system is implemented using one basic FB that is described by the state machine shown in Fig. 3. The controller uses the following signals from the plant represented by Boolean input variables:

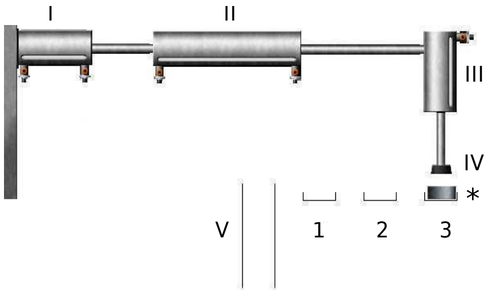


Fig. 2. Pick-and-Place manipulator

- `c1Home`/`c1End` – is horizontal cylinder I in fully retracted/extended position;
- `c2Home`/`c2End` – is horizontal cylinder II in fully retracted/extended position;
- `vcHome`/`vcEnd` – is vertical cylinder III in fully retracted/extended position;
- `pp1pp2`/`pp3` – is a WP present on input slider 1/2/3;
- `vac` – is the vacuum unit IV on.

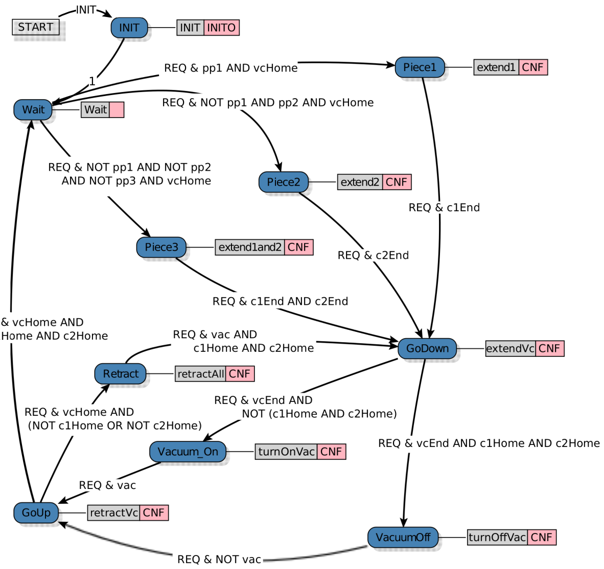


Fig. 3. The original state machine implementing control logic for the Pick-and-Place manipulator system

The following commands can be issued by the controller to the plant:

- `c1Extend/c1Retract` – extend/retract cylinder I;
- `c2Extend/c2Retract` – extend/retract cylinder II;
- `vcExtend` – extend cylinder III;
- `vacuum_on/vacuum_off` – turn the vacuum unit on/off.

Considered temporal properties include safety properties expressing that the system never reaches unsafe states and liveness properties expressing some functional requirements. The set of atomic propositions includes all input and output variables described above and additional plant variables `lifted` – whether a WP has been lifted from some input slider. In addition, considered temporal properties are formulated using the following predicates:

- $\mathtt{vp1} = \mathtt{c1Home} \land \mathtt{c2Home} \land \mathtt{vcEnd} \land \mathtt{vac} \land \mathtt{pp1}$ – a WP has been picked up from input slider 1;
- $\mathtt{dropped} = \mathtt{lifted} \land \mathtt{c1Home} \land \mathtt{c2Home} \land \mathtt{vcEnd} \land \neg\mathtt{vacuum\_on} \land \mathtt{vacuum\_off}$ – indicates that a previously lifted WP has been dropped to the output slider;
- $\mathtt{all\_home} = \mathtt{c1Home} \land \mathtt{c2Home} \land \mathtt{vcHome}$ – all cylinders are in home position.

The set of considered temporal properties is described in Table II. In experiments we focused on processing WPs from the first input slider. The first three properties φ_1–φ_3 are safety properties expressing that the controller must not issue contradictory commands to the plant. Properties φ_4 and φ_5 prohibit situations when both cylinders I and III move simultaneously. Property φ_6 is a liveness property ensuring that if a WP is detected on input slider 1, it is eventually lifted from this slider. The second liveness property φ_7 ensures that if a WP has been captured by the vacuum unit, it is eventually dropped to the output slider. Finally, property φ_8 prohibits unnecessary spontaneous movement of cylinders in the case when no WP should be processed. Note that the original controller does not comply with properties $\mathbf{G}(\mathtt{pp2} \to \mathbf{F}(\mathtt{vp2}))$ and $\mathbf{G}(\mathtt{pp3} \to \mathbf{F}(\mathtt{vp3}))$: if a new WP always appears on input slider 1 immediately after the previous one is lifted, WPs from other sliders will never be lifted. Therefore these properties are not used in our experiments.

As in [11], the formal model of the plant was prepared manually. Before running experiments, we built a NuSMV model of the original state machine of the controller shown in Fig. 3 and checked that the temporal properties described in Table II are satisfied for the closed-loop system model.

VII. Experimental evaluation: inferring controllers for the Pick-and-Place system

The purpose of the experiment was to assess the ability of the developed method to infer a correct controller for the Pick-and-Place system from incomplete data – temporal specification described in Table II and a set of execution scenarios. Execution traces were derived from the original controller (Fig. 3) by means of closed-loop simulation in NxtStudio. Each recorded trace corresponded to a simulation in which several WPs were consequently processed by the manipulator. The first trace $\langle 1 \rangle$ correpsonds to a single simulation in which a single WP is lifted from slider 1, the second trace $\langle 2 \rangle$ – a WP is lifted from slider 2, the fourth trace $\langle 1, 1 \rangle$ – a WP is lifted from slider 1 two times in a row, etc. We recorded 39 traces which correspond to 313 positive scenario tree nodes.

The main considered functional LTL property is φ_6 : $\mathbf{G}(\mathtt{pp1} \to \mathbf{F}(\mathtt{vp1}))$, which requires that whenever a WP appears on the first input slider (`pp1`), it will be eventually lifted (`vp1`). With respect to this property, all used sets of execution traces can be divided into two classes.

1) Sets $\{\langle 1 \rangle\}$, $\{\langle 1 \rangle, \langle 2 \rangle\}$, $\{\langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle\}$ describing behaviours in which a WP is only once lifted from slider 1;
2) All other sets starting from $\{\langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 1, 1 \rangle\}$ in which a WP in slider 1 is processed at least twice.

Informally, sets of traces from the second class *cover* property φ_6 better than sets from the first class: the desired behaviour of the controller is already demonstrated by the trace $\langle 1, 1 \rangle$, so all the inference algorithm needs to do is to forbid undesired behaviours. On the other hand, for sets from the first class the inference procedure needs to come up with the desired behavior and forbid undesired ones on the basis of counterexamples only, which should be much harder.

Indeed, for traces of the second class the proposed algorithm generates a correct solution satisfying all LTL properties within an average of 376 seconds – from 92 to 858 depending on the size of the positive traces. This requires an average of 11 runs of the CSP solver, and the negative trees on the last iteration have an average of 54 vertices – from 42 to 84.

On the contrary, for traces of the first class the algorithm took from 8 to 18 hours to finish, used from 461 to 649 runs of the solver, and resulted in negative trees with 4067 to 5079 vertices.

One of generated state machines is shown in Fig. 4. All inferred solutions comply with given positive traces and LTL

TABLE II

TEMPORAL PROPERTIES VERIFIED FOR THE PICK-AND-PLACE SYSTEM

	Property	Description
φ_1	$\mathbf{G}(\neg(\texttt{c1Extend} \wedge \texttt{c1Retract}))$	cylinder I must not be issued commands to extend and retract simultaneously
φ_2	$\mathbf{G}(\neg(\texttt{c2Extend} \wedge \texttt{c2Retract}))$	similar property for cylinder II
φ_3	$\mathbf{G}(\neg(\texttt{vacuum\_on} \wedge \texttt{vacuum\_off}))$	similar property for the vacuum unit
φ_4	$\mathbf{G}(\neg\texttt{vcHome} \wedge \neg\texttt{vcEnd} \rightarrow \texttt{c1Home} \vee \texttt{c1End})$	if the vertical cylinder is in the intermediate position, cylinder I must be either in home or end position
φ_5	$\mathbf{G}(\neg\texttt{c1Home} \wedge \neg\texttt{c1End} \rightarrow \texttt{vcHome} \vee \texttt{vcEnd})$	if cylinder I is in the intermediate position, the vertical cylinder must be either in home or end position
φ_6	$\mathbf{G}(\texttt{pp1} \rightarrow \mathbf{F}(\texttt{vp1}))$	if a WP appears on input slider 1 it must be eventually lifted
φ_7	$\mathbf{G}(\texttt{lifted} \rightarrow \mathbf{F}(\texttt{dropped}))$	if a WP is lifted from the input slider it must eventually be dropped to the output slider
φ_8	$\mathbf{G}(\texttt{all\_home} \wedge \neg\texttt{pp1} \wedge \neg\texttt{lifted} \rightarrow \mathbf{X}(\neg\texttt{c1Extend} \wedge \neg\texttt{c2Extend} \wedge \neg\texttt{vcExtend}))$	if all cylinders are in home position and no WP should be processed, no commands to move any cylinders should be issued

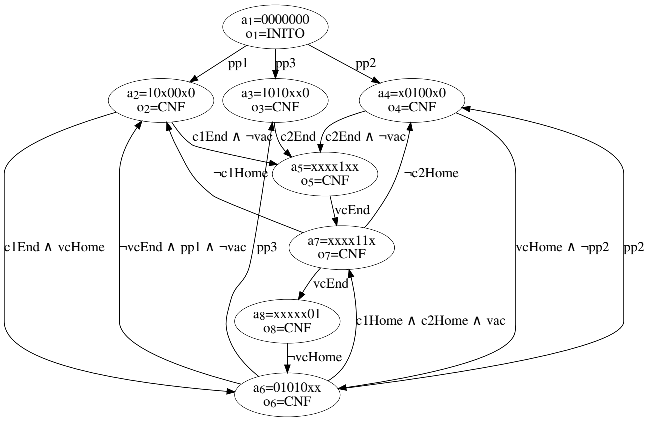


Fig. 4. Example of a state machine constructed using the proposed approach

formulas by construction, and were additionally validated with closed-loop simulation testing in nxtStudio.

VIII. CONCLUSION

We have proposed an algorithm for inferring controller logic in the form of a state machine from given behavior examples (execution traces) and formal specification (set of LTL properties). The proposed approach extends previous work and is based on constraint programming implementing iterative counterexample elimination. Experimental evaluation has shown that the approach is practically applicable to inferring moderate-size controllers and is particularly efficient in the case when the formal specification is in some sense covered by supplied execution traces. In this particular implementation we reduced the problem to CSP, but it is possible to use a SAT translation instead, which will probably result in increased performance. In addition, the use of incremental SAT solvers may increase efficiency even further. This will be part of future work in this direction.

ACKNOWLEDGEMENTS

This work was financially supported by the Government of Russian Federation, Grant 08-08, and also partially funded by RFBR according to the research project No. 16-37-00205 mol_a.

REFERENCES

[1] M. Gold, “Complexity of Automaton Identification from Given Data,” *Inf. Control*, vol. 37, no. 3, pp. 302–320, 1978.

[2] K. R. Apt, *Principles of Constraint Programming*. New York: Cambridge University Press, 2003.

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

[4] 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.

[5] 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.

[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] 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.

[8] 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.

[9] I. Buzhinsky and V. Vyatkin, “Plant model inference for closed-loop verification of control systems: Initial explorations,” in *IEEE Int. Conf. Ind. Informat.*, 2016, pp. 736–739.

[10] ——, “Automatic inference of finite-state plant models from traces and temporal properties,” *IEEE Trans. Ind. Informat.*, 2017.

[11] D. Chivilikhin, I. Ivanov, A. Shalyto, and V. Vyatkin, “Reconstruction of function block controllers based on test scenarios and verification,” in *14th IEEE International Conference on Industrial Informatics*, 2016, pp. 646–651.

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

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

[14] 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.

[15] MiniZinc. [Online]. Available: http://www.minizinc.org

[16] The Chuffed CP solver. [Online]. Available: https://github.com/chuffed/chuffed

[17] Graphviz. [Online]. Available: https://www.graphviz.org/

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

[19] 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.