Received 11 November 2022, accepted 5 December 2022, date of publication 14 December 2022, date of current version 21 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3229003

RESEARCH ARTICLE

fbSAT: Automatic Inference of Minimal Finite-State Models of Function Blocks Using SAT Solver

KONSTANTIN CHUKHAREV AND DANIIL CHIVILIKHIN
Computer Technologies Laboratory, ITMO University, 197101 Saint Petersburg, Russia

Corresponding authors: Konstantin Chukharev (kchukharev@itmo.ru) and Daniil Chivilikhin (chivdan@itmo.ru)

This work was supported by the Ministry of Science and Higher Education of Russian Federation under Project 075-03-2020-139/2 (goszadanie no. 2019-1339).

ABSTRACT Finite-state models are widely used in software engineering, especially in the development of control systems. In control applications, such models are often developed manually, which can make it difficult to keep them up to date. To simplify the maintenance process, an automatic approach can be used to infer models from behavior examples and temporal specification. As an example of a specific control systems development application, we focus on inferring finite-state models of function blocks (FBs) defined by the IEC 61499 international standard for distributed automation systems. In this paper, we propose a method for inferring FB models from behavior examples based on reduction to the Boolean satisfiability problem (SAT). Additionally, we take into account linear temporal properties using counterexample-guided synthesis. The developed tool, fbSAT, implementing the proposed method is evaluated in three case studies: inferring a finite-state controller for a Pick-and-Place manipulator, reconstructing randomly generated automata, and minimizing transition systems. In contrast to existing approaches, the suggested method is more efficient and produces finite-state models that are minimal in terms of both the number of states and the complexity of guard conditions.

INDEX TERMS Control system synthesis, inference algorithms, Boolean satisfiability, counterexample-guided inductive synthesis, formal verification, model checking.

I. INTRODUCTION

The non-trivial process of develponig control logic for an industrial control system may be reduced to the creation of a finite-state automaton or a system of interconnected automata. Controller behavior may be represented using the deterministic finite-state model, describing how the system reacts to input events and which output actions it produces. Such models are extensively used in program testing [1], [2] (*e.g.*, for model-based test case generation) and verification [3], [4] (*e.g.*, the behavior of a program is modeled using a finite-state machine, and then model checking is applied to check whether the model has the desired properties), as well as for representing and modelling controllers in control systems. One practical example of finite-state model application is the international standard for distributed automation systems development IEC 61499 [5], which defines control systems as networks of interconnected function blocks (FBs), specified by their *interfaces* and implementations (*control algorithms*).

In practice, most finite-state models for control applications are developed *manually* – this is a tedious and error-prone approach. Furthermore, there is the issue of maintaining these models to be up-to-date and consistent during the changes in system parameters, architecture, and logic. An alternative to the manual process is *automatic* synthesis from given execution scenarios and/or temporal properties [6], [7], [8], [9], [10], [11], [12], [13]. Inferred models can be used for model-based testing, verification, and can even replace the original controller.

The associate editor coordinating the review of this manuscript and approving it for publication was Engang Tian.

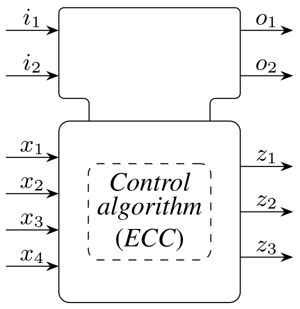


FIGURE 1. IEC 61499 Function block example.

The contribution of this paper is a method for synthesis of minimal finite-state FB models, which, in contrast to existing approaches (see Section III), allows *simultaneously* and *efficiently* accounting for (1) behavior examples, (2) linear temporal logic (LTL) properties, and (3) minimality of synthesized automata both in terms of number of states and guard conditions complexity. We also present a tool FBSAT implementing the proposed method, and evaluate it in several experimental case studies. Our approach is designed for FB model synthesis, but it can be applied to inference of other types of state machines with minimal modifications (see *e.g.*, Section VI).

II. PROBLEM STATEMENT

A function block (FB) (Fig. 1) is characterized by its *interface* and *control algorithm*. The interface defines input/output events (sets $\mathcal{I}$ and $\mathcal{O}$) and input/output variables (sets $\mathcal{X}$ and $\mathcal{Z}$) which can be, for example, Boolean, integer or real-valued. In this paper, we consider Boolean input/output variables only. The control algorithm is represented by a finite-state machine extended with guard conditions, and called *execution control chart* (ECC). In the following we will refer to such a machine simply as *an automaton*. A complete formal definition of an ECC can be found in [14]. Since in this paper we deal only with Boolean inputs and outputs, we use a simplified definition: an automaton $\mathcal{A}$ is a tuple $(Q, q_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{X}, \mathcal{Z}, \tau, \psi, \omega)$, where:

- Q is a finite set of states;
- $q_{\text{init}} \in Q$ is the initial state;
- $\mathcal{I}$ and $\mathcal{O}$ are the sets of input/output events;
- $\mathcal{X}$ and $\mathcal{Z}$ are the sets of input/output variables;
- $\tau: Q \times \mathcal{I} \times \mathbb{B}^{|\mathcal{X}|} \to Q$ is the transition function;
- $\psi: Q \times \mathcal{I} \times \mathbb{B}^{|\mathcal{X}|} \to (\mathcal{O} \cup \{\varepsilon\})$ is the output event function;
- $\omega: Q \times \mathcal{I} \times \mathbb{B}^{|\mathcal{X}|} \times \mathbb{B}^{|\mathcal{Z}|} \to \mathbb{B}^{|\mathcal{Z}|}$ is the output function.

Each state has an associated output event and an *algorithm*, a function that can modify the values of output variables $\mathcal{Z}$. In this paper, we only consider algorithms over Boolean vectors $\mathbb{B}^{|\mathcal{Z}|} \to \mathbb{B}^{|\mathcal{Z}|}$, where each output variable only depends on its previous value (assuming that its initial value is `False`). Each transition has an associated input event and a *guard condition*, which is a Boolean function that indicates the possibility to follow the transition. We consider guard conditions to be Boolean functions over the input variables $\mathcal{X}$: $\mathbb{B}^{|\mathcal{X}|} \to \mathbb{B}$. An automaton $\mathcal{A}$ behaves as a finite-state transducer: it accepts *input actions* and produces *output actions*, while keeping track of output variable values. The transition function $\tau: Q \times \mathcal{I} \times \mathbb{B}^{|\mathcal{X}|} \to Q$ defines the state in which the automaton finishes processing an input action. The output event function $\psi: Q \times \mathcal{I} \times \mathbb{B}^{|\mathcal{X}|} \to (\mathcal{O} \cup \{\varepsilon\})$ defines the output event emission rule. Finally, the output function $\omega: Q \times \mathcal{I} \times \mathbb{B}^{|\mathcal{X}|} \times \mathbb{B}^{|\mathcal{Z}|} \to \mathbb{B}^{|\mathcal{Z}|}$ defines the changes in output variable values. Note that the automaton may not react to some input actions, *i.e.* it may stay in the same state and not produce an output action. In that case:

- $\tau(q, i, \bar{x}) = q$;
- $\psi(q, i, \bar{x}) = \varepsilon$;
- $\omega(q, i, \bar{x}, \bar{z}) = \bar{z}$,

where $q \in Q$, $i \in \mathcal{I}$, $\bar{x} \in \mathbb{B}^{|\mathcal{X}|}$, $\bar{z} \in \mathbb{B}^{|\mathcal{Z}|}$.

An *execution scenario* is a sequence of *elements* $\langle i[\bar{x}], o[\bar{z}]\rangle$, where each element consists of an input action $i[\bar{x}]$ and an output action $o[\bar{z}]$. An input action is a pair of an input event $i \in \mathcal{I}$ and an *input* $\bar{x} \in \mathbb{B}^{|\mathcal{X}|}$, whereas an output action is a pair of an output event $o \in \mathcal{O} \cup \{\varepsilon\}$ and an *output* $\bar{z} \in \mathbb{B}^{|\mathcal{Z}|}$. An *empty* output event ε is necessary to represent the absence of an output action, *e.g.*, in the case when an automaton does not react to an input action. A *positive scenario* is an execution scenario representing a desired behavior of an automaton. Commonly, such scenarios are obtained by simulating an existing model (in a simulation tool, such as Matlab, nxtSTUDIO[1]), or by collecting data from a real control system. An example of a set of three scenarios $\mathcal{S} = \{s_1, s_2, s_3\}$ is shown below:

$$\begin{aligned} s_1 &= [\langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[01]}, \mathtt{B[1]}\rangle; \\ &\qquad \langle \mathtt{R[00]}, \varepsilon\mathtt{[1]}\rangle; \langle \mathtt{R[01]}, \mathtt{B[0]}\rangle], \\ s_2 &= [\langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[10]}, \mathtt{A[0]}\rangle; \\ &\qquad \langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[01]}, \mathtt{B[1]}\rangle], \\ s_3 &= [\langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[10]}, \mathtt{A[0]}\rangle; \langle \mathtt{R[10]}, \mathtt{A[0]}\rangle]. \end{aligned} \tag{1}$$

We say that an automaton *satisfies* a scenario if, while sequentially receiving input actions from scenario elements, it produces the same sequence of output actions as in the scenario.

An *LTL specification* $\mathcal{L}$ is a set of *LTL formulas* that describes the temporal properties of a finite-state model. An LTL formula is an expression that may contain propositional variables (*i.e.* input/output events/variables of the automaton), logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$), and temporal operators (*e.g.*, **X** is "next", **U** is "until", **G** is "always", **F** is "eventually"). An LTL specification can be verified using a model checker, which produces a counterexample for each violated LTL formula. We convert each counterexample into a *negative scenario* representing undesired behavior. We describe this in detail in Section IV-B.

Ultimately, the problem addressed in this paper is to find the most succint (minimal) automaton that satisfies all

[1]https://www.nxtcontrol.com/en/engineering

positive scenarios $\mathcal{S}^+$ and complies with the given LTL specification $\mathcal{L}$. Commonly, finite-state models are minimized *w.r.t.* their number of states and/or transitions [6], [8], [11]. In this work we additionally explicitly consider complexity of guard conditions: the automaton is minimized *both* in terms of the number of states and the complexity of its guard conditions, measured as the total number of vertices in parse trees of corresponding Boolean formulas.

III. RELATED WORK

There exists a large body of work on SAT-based synthesis of circuits, bit-vector programs, domain-specific programs, *etc*. However, in this work we are interested specifically in synthesis of finite-state machines: first, state-based models are comprehensible, and their formal verification is relatively simple; second, they can be directly used in control applications for controller logic implementation, *e.g.*, in Matlab/Stateflow, nxtSTUDIO [15].

The problem of finding a minimal deterministic finite-state machine from behavior examples is known to be NP-complete [16], and the complexity of the LTL synthesis problem is double exponential in the length of the LTL specification [17]. Despite this, synthesis of various types of finite-state models from behavior examples and/or formal specification has been addressed by many researchers including [3], [6], [7], [8], [10], [11], [12], [18], [19], [20], [21], [22], and [13] with methods based on heuristic state merging, evolutionary algorithms, and SAT solvers. In the context of this paper we are interested in exact methods, so we direct our attention to SAT-based methods.

The extended Finite-State Machine (EFSM) (in terms of [8]) is a model that is similar to the ECC considered in this paper: it combines Mealy and Moore automaton semantics, and uses conditional state transitions. Transitions are labeled with input events and Boolean formulas over the input variables, and automaton states have associated sequences of output actions. Several approaches based on translation to SAT [8], [23] have been proposed for inferring EFSMs from behavior examples and LTL properties. In EFSM-tools [8], LTL properties are accounted for via an iterative counterexample prohibition approach, but minimization of guard conditions is not considered.

BoSy [7] implements bounded synthesis of a *transition system* (a type of automaton similar to EFSM and ECC) only from LTL properties (scenarios are not considered). Apart from a SAT-based approach, a more efficient solution based on a Quantified SAT (QSAT, also called QBF, Quantified Boolean Formula satisfiability problem) encoding is developed. Transition systems inferred using the SAT encoding are *explicit* (in the sense that the guard conditions include all input variables), whereas the QSAT encoding generates *symbolic* models (guard conditions are Boolean formulas over the input variables). BoSy ensures minimality of found solutions *w.r.t.* the number of states, however it does minimize guard conditions, which tend to be large and incomprehensible. An approach to simplify generated solutions is suggested in [24], where the SAT encoding is augmented with constraints for minimizing the number of cycles in the transition system. However, guard conditions complexity is not addressed. Furthermore, BoSy does not support behavior examples. Though they can be modeled with LTL, this approach is inefficient even for behavior examples of moderate size. Other LTL synthesis techniques, *e.g.*, G4LTL-ST [12] and Strix [25], have the same drawbacks in application to the considered problem: no guard conditions minimization and lack of support for behavior examples.

In [26], the FBCSP method is proposed for inferring an FB model from execution scenarios via a translation to the Constraint Satisfaction Problem (CSP). However, FBCSP has the following restrictions. Guard conditions are generated in *complete* form, *i.e.* the corresponding Boolean formulas depend on *all* input variables. Such models do not generalize to unseen data. This is countered by a greedy guard conditions minimization algorithm, but it does not guarantee the minimality of the result. In [27], FBCSP is extended with a counterexample prohibition procedure to account for LTL properties, which is similar to the one used in EFSM-tools. Guard conditions are represented with fixed-size conjunctions of positive/negative literals of the input variables. The drawback of this approach is that it is inefficient when the LTL specification is insufficiently covered with behavior examples.

In [28], a two-stage approach is developed: on the first stage, a base model is inferred using a SAT encoding, and on the second stage its guard conditions are minimized via a CSP encoding, in which guard condition Boolean formulas are represented with parse trees. By introducing a total bound on the number of nodes in these parse trees and solving a series of CSP problems, the method finds a model with minimal guard conditions *w.r.t.* the base model identified on the first stage. Global minimality of guard conditions is not guaranteed due to the two-stage implementation: minimal guards may correspond to another base model, not the one found on the first stage. The same argument applies against any approach based on state machine minimization [29]. In addition, LTL properties are not supported by approaches of this type.

Overall, none of the existing methods allow *simultaneously* and *efficiently* accounting for (1) behavior examples, (2) LTL properties, and (3) minimality of synthesized automata in terms of both the number of states and the complexity of the guard conditions. The approach proposed in this paper extends [28] and contributes to the state-of-the-art in SAT-based state machine synthesis: it supports positive behavior examples, realizes counterexample-guided synthesis to account for LTL properties, and produces models minimal both in terms of the number of states and guard conditions complexity.

IV. PROPOSED APPROACH

In this section we describe the proposed approach for inferring minimal FB models from a given set of positive

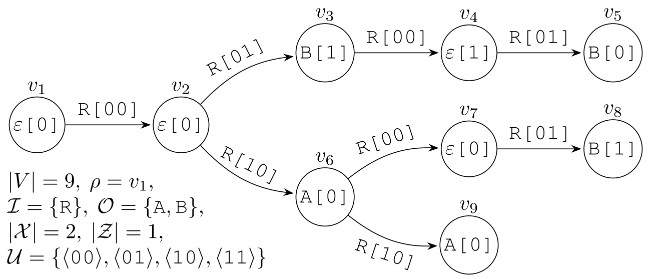


FIGURE 2. Scenario tree constructed from scenarios (1).

scenarios and an LTL specification. In Section IV-A we describe a convenient storage structure for execution scenarios, the *scenario tree*. In Section IV-B we describe the process of verifying an LTL specification using a model checker tool, which produces a counterexample for each violated LTL formula. Obtained counterexamples are converted into *negative scenarios* representing the undesired behavior, which must be prohibited in the desired automaton. In Section IV-C we describe the reduction of the FB model inference problem to SAT. In Section IV-D we describe the process of inferring an FB model that is minimal both in terms of the number of states and guard conditions complexity.

A. SCENARIO TREE

A *scenario tree* $\mathcal{T}$ is a prefix tree built from the given scenarios $\mathcal{S}$. Before the scenario tree construction, we prepend each scenario with an auxiliary element consisting only of an output action $\varepsilon[\langle 0\ldots 0\rangle]$. With this we ensure that all scenarios have a common prefix. Each tree node and its incoming edge correspond to a scenario element: a node is marked with an output action, and an edge is marked with an input action.

Further in this paper, we will refer to the key features of a scenario tree as follows: V is the set of tree nodes; $\rho \in V$ is the root of the tree; $tp(v) \in V$ is the parent of node v, $v \neq \rho$; $tie(v) \in \mathcal{I}$ is an input event on the incoming edge of node v, $v \neq \rho$; $toe(v) \in \mathcal{O} \cup \{\varepsilon\}$ is an output event in node v, where ε is an empty event; $V_{(\mathrm{act})} = \{v \in V \setminus \{\rho\}\, toe(v) \neq \varepsilon\}$ is the set of *active* tree nodes; $V_{(\mathrm{pass})} = \{v \in V \setminus \{\rho\}\, toe(v) = \varepsilon\}$ is the set of *passive* tree nodes; $\mathcal{U} \subseteq \mathbb{B}^{|\mathcal{X}|}$ is the set of *inputs* encountered in scenarios; $tin(v) \in \mathcal{U}$ is an input on the incoming edge of node v, $v \neq \rho$; $tov(v, z) \in \mathbb{B}$ is the value of output variable z in node v. The root ρ has no parent, thus $tp(\rho)$, $tie(\rho)$, and $tin(\rho)$ are undefined. A *positive scenario tree* $\mathcal{T}^+$ is a scenario tree built from positive scenarios $\mathcal{S}^+$. An example of a scenario tree constructed from scenarios (1) is shown in Fig. 2.

B. LTL VERIFICATION, COUNTEREXAMPLES, NEGATIVE SCENARIOS

An LTL specification can be verified using a model checker tool, which produces a *counterexample* for each violated LTL formula. We use a symbolic model checker NuSMV [30]. For safety properties, a counterexample is a finite sequence of

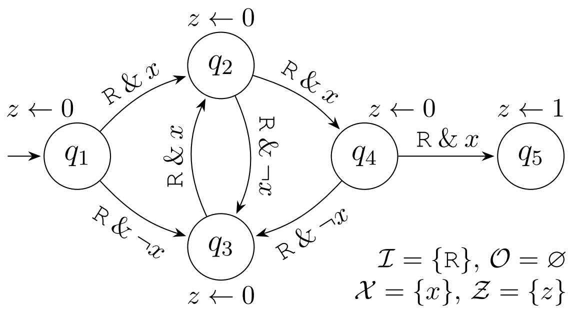
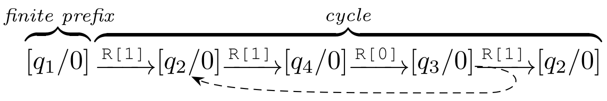


FIGURE 3. Example automaton with a looping behavior.



finite prefix
cycle
$[q_1/0] \xrightarrow{\text{R[1]}} [q_2/0] \xrightarrow{\text{R[1]}} [q_4/0] \xrightarrow{\text{R[0]}} [q_3/0] \xrightarrow{\text{R[1]}} [q_2/0]$

FIGURE 4. Counterexample for the liveness LTL property $\mathcal{L}_2 = \mathbf{F}z$.

execution states. For liveness properties, a counterexample is an infinite but periodic sequence of states, which can be represented as a finite prefix followed by a cycle [31].

A *negative scenario* is an execution scenario representing an undesired behavior. We convert each counterexample into a negative scenario as follows. Consider the example automaton in Fig. 3, where $\mathcal{I} = \{$R$\}$, $\mathcal{O} = \varnothing$, $\mathcal{X} = \{x\}$, $\mathcal{Z} = \{z\}$. Also consider two LTL properties: $\mathcal{L} = \{\mathbf{G}\,\neg z,\ \mathbf{F}\,z\}$. A counterexample for the safety property $\mathcal{L}_1 = \mathbf{G}\,\neg z$ is a finite sequence $[q_1/0] \xrightarrow{\text{R[1]}} [q_2/0] \xrightarrow{\text{R[1]}} [q_4/0] \xrightarrow{\text{R[1]}} [q_5/1]$. Here, the notation $[q_1/0]$ indicates the execution state (in NuSMV's sense), in which the automaton is in the state q_1, and the current value of the output variable z is 0. The corresponding "loopless" negative scenario looks as follows: $\widehat{s}_1 = [\langle$R[1], A$\rangle$; $\langle$R[1], A$\rangle$; $\langle$R[1], A$\rangle]$. Counterexample for the liveness property $\mathcal{L}_2 = \mathbf{F}\,z$ is a finite prefix followed by a cycle (Fig. 4).

The corresponding *looping* negative scenario looks as follows: $\widehat{s}_2 = [\underline{\langle \text{R[1]}, \text{A}\rangle}$; $\langle$R[1], A$\rangle$; $\langle$R[0], A$\rangle$; $\langle$R[1], A$\rangle]$, where the first element (underlined) is the beginning of a loop.

A *negative scenario tree* $\mathcal{T}^-$ is a scenario tree built from negative scenarios $\mathcal{S}^-$. We denote the set of all nodes that correspond to the last elements of loopless scenarios as $\widehat{V}^{(\mathrm{ends})}$. We augment the tree with special *back edges* from the end to the beginning of each loop from looping scenarios. We denote the set of all nodes which are connected with node $\widehat{v}$ via a back edge as $\widehat{tbe}(\widehat{v}) \subseteq \widehat{V}$. For example, for the negative scenario tree built from the scenario $\widehat{s}_2$ only: $\widehat{tbe}(\widehat{v}_5) = \{\widehat{v}_2\}$ (indices are shifted by 1, since we prepend the scenario tree with an auxiliary root node $\widehat{\rho} = \widehat{v}_1$). All other tree features are the same as defined in Section IV-A, but marked with the hat symbol, *e.g.*, $\widehat{v} \in \widehat{V}$, $\widehat{tp}(\widehat{v})$, $\widehat{tie}(\widehat{v})$.

C. FB MODEL INFERENCE USING SAT SOLVER

We propose a method for inferring an FB model based on a reduction to SAT. The reduction consists in formally describing an automaton $\mathcal{A}$ of size C by constructing a Boolean formula that is satisfiable if and only if there exists an automaton which satisfies given positive scenarios $\mathcal{S}^+$ and does not satisfy given negative scenarios $\mathcal{S}^-$. For encoding non-Boolean variables with bounded domains we use standard pairwise encoding (also known as "sparse" or "direct" [32]) and Onehot+Binary [33] encoding.

The reduction includes four parts. First, we encode the *automaton structure*. Second, we encode the *positive* scenario tree $\mathcal{T}^+$ *mapping* and enforce its satisfaction. Third, we encode the *guard conditions structure*, *i.e.* the structure of parse trees of corresponding Boolean formulas, and declare cardinality constraints bounding the guard conditions size. Lastly, we encode the *negative* scenario tree $\mathcal{T}^-$ *mapping* and prohibit its satisfaction.

The goal is to infer an automaton with $|Q| = C$ states. We assume that each state has at most K outgoing transitions. Further in this paper we assume that $K = C \cdot |\mathcal{I}|$, since it is the safest minimum value that does not prohibit the inference of an automaton, which may happen for smaller values of K. However, lowering this value greatly reduces the size of the reduction (*i.e.* number of declared Boolean variables and clauses), which is likely to significantly increase the solving efficiency. Further in this section we assume that $b \in \mathbb{B} = \{\bot, \top\} = \{0, 1\}$, $q, q' \in Q$, $k \in [1\,..\,K]$, $i \in \mathcal{I}$, $u \in \mathcal{U}$, $v \in V$, unless stated otherwise.

1) ENCODING THE AUTOMATON STRUCTURE

Each state has an associated output event and an algorithm. Variable $\phi_q \in \mathcal{O} \cup \{\varepsilon\}$ denotes the output event in state q. Variable $\gamma_{q,z,b} \in \mathbb{B}$ represents the algorithm for the output variable z.

Each transition has an associated input event and a *guard condition*, which is a Boolean function over the input variables $\mathcal{X}$. Variable $\tau_{q,k} \in Q_0$ ($Q_0 = Q \cup \{q_0\}$) denotes the destination state of the k-th transition from the state q. "Transitions" to the auxiliary state $q_0 \notin Q$ are called *null-transitions* and represent the absence of a transition. *W.l.o.g.* we ensure that null-transitions have the largest indices:

$$(\tau_{q,k} = q_0) \rightarrow (\tau_{q,k+1} = q_0).$$

Variable $\xi_{q,k} \in \mathcal{I} \cup \{\varepsilon\}$ denotes the input event on the k-th transition from the state q. Only null-transitions are marked with the special ε input event:

$$(\tau_{q,k} = q_0) \leftrightarrow (\xi_{q,k} = \varepsilon).$$

Variable $\theta_{q,k,u} \in \mathbb{B}$ represents a guard condition represented by a truth table: it denotes the value of the guard condition of the k-th transition from the state q for the input u. Variable $\delta_{q,k,i,u} \in \mathbb{B}$ represents a guard firing function by denoting whether the k-th guard from the state q fires on the input action $i[u]$. These two variables are related as follows:

$$\delta_{q,k,i,u} \leftrightarrow (\xi_{q,k} = i) \wedge \theta_{q,k,u}.$$

According to the IEC 61499 standard, each state has a transition priority: the automaton follows the *first fired* transition or stays in the same state if no transition fired. Variable $f\!f_{q,i,u} \in [0\,..\,K]$ denotes the index of a transition which *fires first* on input action $i[u]$. $f\!f_{q,i,u} = 0$ means that no transition fires at all. A transition *fires first* iff all previous transitions do not fire:

$$(f\!f_{q,i,u} = k) \leftrightarrow \delta_{q,k,i,u} \wedge \bigwedge\nolimits_{1 \le k' < k} (\neg \delta_{q,k',i,u}).$$

When the automaton in state q processes an input action $i[u]$, it either (1) switches to another state q' or (2) *ignores* it (or rather, *reacts by ignoring*) by staying in the state q. Such behavior is represented by variable $\lambda_{q,i,u} \in Q_0$, where $\lambda_{q,i,u} = q_0$ denotes the second (2) case. Note that in the first (1) case the automaton may switch (through a *loop*-transition) into the same state $q' = q$. Formally, the definition for λ is the following:

$$(\lambda_{q,i,u} = q') \leftrightarrow \bigvee\nolimits_{k \in [1..K]} \Big[(\tau_{q,k} = q') \wedge (f\!f_{q,i,u} = k) \Big].$$

2) BOUNDING THE NUMBER OF TRANSITIONS

In order to declare an upper bound for the total number of not-null transitions T in the automaton, we impose the *cardinality constraint*:

$$\sum\nolimits_{q \in Q,\ k \in [1..K]} \mathrm{BOOL2INT}(\tau_{q,k} \neq q_0) \le T.$$

In order to encode this in CNF, we use a technique described by [34], which consists in declaring a *totalizer*, which encodes the sum in unary form, and a *comparator*, which bounds this sum. We omit the formal definition of the resulting constraints which can be found in [34].

3) BFS-BASED SYMMETRY BREAKING FOR AUTOMATON STATES

Additionally, we declare auxiliary symmetry-breaking constraints [9], which force the automaton states to be enumerated in the order they are visited by the breadth-first search (BFS) algorithm launched from the initial state. Variable $\tau^{\text{bfs-}\mathcal{A}}_{q_a,q_b} \in \mathbb{B}$ ($q_a, q_b \in Q$) indicates the existence of a transition from q_a to q_b:

$$\tau^{\text{bfs-}\mathcal{A}}_{q_a,q_b} \leftrightarrow \bigvee\nolimits_{k \in [1..K]} (\tau_{q_a,k} = q_b).$$

Variable $\pi^{\text{bfs-}\mathcal{A}}_{q_b} \in \{q_1, \ldots, q_{b-1}\}$ ($b \in [2\,..\,C]$) denotes the parent of the state q_b in the BFS traversal tree:

$$\left(\pi^{\text{bfs-}\mathcal{A}}_{q_b} = q_a\right) \leftrightarrow \tau^{\text{bfs-}\mathcal{A}}_{q_a,q_b} \wedge \bigwedge\nolimits_{c<a} \left(\neg \tau^{\text{bfs-}\mathcal{A}}_{q_c,q_b}\right).$$

The actual BFS constraint is defined as follows:

$$\left(\pi^{\text{bfs-}\mathcal{A}}_{q_b} = q_a\right) \rightarrow \bigwedge\nolimits_{c<a} \left(\pi^{\text{bfs-}\mathcal{A}}_{q_{b+1}} \neq q_c\right).$$

4) ENCODING THE MAPPING OF POSITIVE SCENARIO TREE

The goal is to organize a mapping $\mu \colon V \rightarrow Q$ between the nodes of the positive scenario tree $\mathcal{T}^+$ and the states of the automaton $\mathcal{A}$ (see Fig. 5). Variable $\mu_v \in Q$ denotes the *satisfying state* in which the automaton finishes processing the sequence of scenario elements formed by the path from the

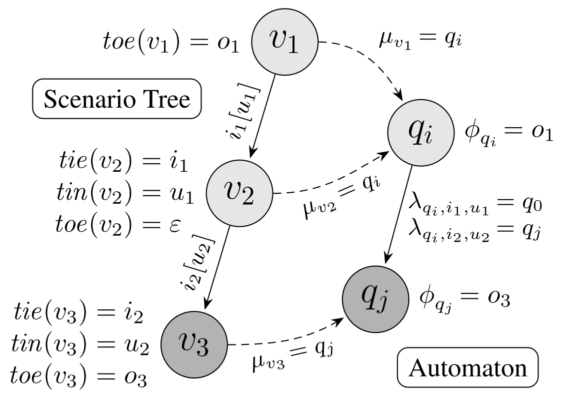


FIGURE 5. Tree-to-automaton mapping example.

root ρ to the node v. The root ρ itself maps to the initial state: $\mu_\rho = q_{\text{init}}$.

Passive nodes (with $toe(v) = \varepsilon$) map to the same states as their parents and correspond to the situation when the automaton ignores the input action, thus:

$$(\mu_p = q) \to (\mu_v = q) \wedge (\lambda_{q,i,u} = q_0),$$

where $v \in V_{(\text{pass})}$, $p = tp(v)$, $q \in Q$, $i = tie(v)$, $u = tin(v)$.

Active nodes correspond to the situation when the automaton reacts to an input action by switching the state and producing an output action, which we constrain according to the tree node:

$$(\mu_p = q) \to \Big((\mu_v = q') \to (\lambda_{q,i,u} = q') \wedge (\phi_{q'} = o) \wedge \bigwedge\nolimits_{z \in \mathcal{Z}} (\gamma_{q',z,b} = b')\Big),$$

where $v \in V_{(\text{act})}$, $p = tp(v)$, $q, q' \in Q$, $i = tie(v)$, $u = tin(v)$, $o = toe(v)$, $z \in \mathcal{Z}$, $b = tov(p, z)$, $b' = tov(v, z)$. See Fig. 5 for a mapping example.

5) BASIC ALGORITHM

Constraints declared so far allow inferring a *computable* automaton capable of processing input actions and reacting to them by emitting output actions. Denote by BASIC*$(\mathcal{S}^+, C, T)$ the procedure of inferring an automaton satisfying positive scenarios $\mathcal{S}^+$, which has C states and at most T transitions. This procedure consists of (1) building a positive scenario tree, (2) encoding the automaton structure, the scenario tree mapping, and the cardinality constraint "total number of *not-null* transitions is at most T", and (3) delegating the constructed CNF formula to the SAT solver. The constructed CNF formula consists of $\mathcal{O}(C^2 \cdot |\mathcal{I}| \cdot |\mathcal{U}| + C \cdot |V|)$ variables and $\mathcal{O}(C^3 \cdot |\mathcal{I}| \cdot |\mathcal{U}| + C^2 \cdot |V| \cdot |\mathcal{Z}| + C \cdot |V| \cdot |\mathcal{U}|)$ clauses. Additionally, denote by BASIC$(\mathcal{S}^+, C)$ = BASIC*$(\mathcal{S}^+, C, \infty)$ an alias for the call without a bound on T.

6) ENCODING THE STRUCTURE OF GUARD CONDITIONS

In the above encoding, guard conditions are represented as truth tables (by variable θ), which are not human/interpretable or usable in control systems development software such as Matlab or nxtSTUDIO [15], where guard conditions must be explicitly represented with Boolean formulas. Therefore, we supplement the reduction with an encoding of arbitrary Boolean formulas over the input variables $\mathcal{X}$; each Boolean formula is represented by its parse tree. Each tree is built of P nodes, where P is a parameter. Each node may be either a Boolean operator node or a terminal node representing an input variable. However, not all formulas may need as much as P nodes, and some nodes may remain unused, *i.e.* not included in the tree. We call such nodes *none-typed*. We define the *size* of a parse tree as the number of *typed* (*i.e.* not none-typed) nodes in it. Further in this section we assume that $p \in [1 \,..\, P]$, $q \in [1 \,..\, C]$, $x \in \mathcal{X}$, unless stated otherwise.

Variable $\eta_{q,k,p} \in \{\square, \wedge, \vee, \neg, \bullet\}$ denotes the type of the p-th parse tree node of the guard condition on the k-th transition from the state q, where $\square$ denotes a terminal node, $\bullet$ denotes a none-typed node, and the rest are logical operators. Variable $\chi_{q,k,p} \in \mathcal{X} \cup \{0\}$ denotes the input variable associated with the node p (or its absence). Only terminal nodes have associated input variables:

$$(\eta_{q,k,p} = \square) \leftrightarrow (\chi_{q,k,p} \neq 0).$$

Variables $\pi_{q,k,p} \in [0 \,..\, (p{-}1)]$ and $\sigma_{q,k,p} \in \{0\} \cup [(p{+}1) \,..\, P]$ denote, respectively, the parent and the (left) child of the p-th node (or their absence, *e.g.*, $\pi_{q,k,p} = 0$). These variables are related as follows:

$$(\sigma_{q,k,p} = ch) \to (\pi_{q,k,ch} = p).$$

Only typed nodes, except the root ($p = 1$), have parents:

$$(\eta_{q,k,p} \neq \bullet) \leftrightarrow (\pi_{q,k,p} \neq 0).$$

We do not encode the right child explicitly, but for binary operators we assume that it is next to the left one:

$$\eta_{q,k,p} \in \{\wedge, \vee\} \wedge (\sigma_{q,k,p} = c) \to (\pi_{q,k,c+1} = p).$$

Since each binary operator node must have two children, the P-th and $(P-1)$-th nodes cannot be of type "$\wedge$" or "$\vee$". Similarly, the P-th node cannot be of type "$\neg$".

Variable $\vartheta_{q,k,p,u} \in \mathbb{B}$ denotes the Boolean value of the subformula (represented by the subtree rooted in node p) on input u. Variable θ defined earlier is a shortcut for the root node value: $\theta_{q,k,u} \leftrightarrow \vartheta_{q,k,1,u}$. Terminals have values from the associated input variables; values of non-terminal nodes are calculated according to their types and children values; and none-typed nodes have `False` values:

$$\begin{aligned}
&(\eta_{q,k,p} = \square) \wedge (\chi_{q,k,p} = x) \\
&\quad \to \bigwedge\nolimits_{u \in \mathcal{U}} \left[\vartheta_{q,k,p,u} \leftrightarrow u_x\right]; \\
&(\eta_{q,k,p} = \wedge) \wedge (\sigma_{q,k,p} = c) \\
&\quad \to \bigwedge\nolimits_{u \in \mathcal{U}} \left[\vartheta_{q,k,p,u} \leftrightarrow \vartheta_{q,k,c,u} \wedge \vartheta_{q,k,c+1,u}\right]; \\
&(\eta_{q,k,p} = \vee) \wedge (\sigma_{q,k,p} = c) \\
&\quad \to \bigwedge\nolimits_{u \in \mathcal{U}} \left[\vartheta_{q,k,p,u} \leftrightarrow \vartheta_{q,k,c,u} \vee \vartheta_{q,k,c+1,u}\right];
\end{aligned}$$

$$(\eta_{q,k,p} = \neg) \wedge (\sigma_{q,k,p} = c) \to \bigwedge\nolimits_{u \in \mathcal{U}} \left[\vartheta_{q,k,p,u} \leftrightarrow \neg\vartheta_{q,k,c,u} \right];$$
$$(\eta_{q,k,p} = \bullet) \to \bigwedge\nolimits_{u \in \mathcal{U}} \left[\neg\vartheta_{q,k,p,u} \right].$$

7) BOUNDING THE TOTAL GUARD CONDITIONS SIZE

In order to declare an upper bound for the total size of all guard conditions, *i.e.* the total number of typed (*i.e.* not none-typed) parse tree nodes N, we impose the *cardinality constraint*:

$$\sum\nolimits_{q \in Q,\ k \in [1..K],\ p \in [1..P]} \text{BOOL2INT}(\eta_{q,k,p} \neq \bullet) \leq N.$$

For encoding it in CNF, we again use a technique described by [34], as explained in Section IV-C2.

8) BFS-BASED SYMMETRY BREAKING FOR GUARD CONDITIONS

Additionally, we declare auxiliary symmetry-breaking constraints, which enforce the nodes of parse trees representing the guard conditions to be enumerated in BFS order. Essentially, they are almost identical to the BFS constraints for automaton states, but declared for each parse tree separately (for each $q \in Q$, $k \in [1..K]$). Variable $\tau^{\text{bfs-}\mathcal{G}}_{a,b} \in \mathbb{B}$ $(1 \leq a < b \leq P)$ indicates the existence of an edge from the a-th to the b-th node:

$$\tau^{\text{bfs-}\mathcal{G}}_{a,b} \leftrightarrow (\pi_{q,k,b} = a).$$

Variable $\pi^{\text{bfs-}\mathcal{G}}_{b} \in [1..(b-1)]$ $(b \in [2..P])$ denotes the parent of the b-th node in the BFS traverse tree:

$$\left(\pi^{\text{bfs-}\mathcal{G}}_{b} = a\right) \leftrightarrow \tau^{\text{bfs-}\mathcal{G}}_{a,b} \wedge \bigwedge\nolimits_{c<a} \left(\neg\tau^{\text{bfs-}\mathcal{G}}_{c,b}\right).$$

The actual BFS constraint is defined as follows:

$$\left(\pi^{\text{bfs-}\mathcal{G}}_{b} = a\right) \to \bigwedge\nolimits_{c<a} \left(\pi^{\text{bfs-}\mathcal{G}}_{b+1} \neq c\right).$$

9) EXTENDED ALGORITHM

Denote by $\textsc{Extended}^*(\mathcal{S}^+, C, P, T, N)$ the procedure for inferring an automaton which satisfies positive scenarios $\mathcal{S}^+$, has C states and at most T transitions, while each guard condition (represented by a parse tree) consists of at most P nodes, and the total guards size is at most N. This procedure consists of (1) building a positive scenario tree, (2) encoding the automaton structure, the scenario tree mapping, the guard conditions structure, and the cardinality constraints "total number of *not-null* transitions is at most T" and "total size of guard conditions is at most N", and (3) delegating to the SAT solver. The constructed CNF formula consists of

$$\mathcal{O}(C^2 \cdot |\mathcal{I}| \cdot |\mathcal{U}| + C^2 \cdot P \cdot |\mathcal{U}| + C \cdot |V|)$$

variables and

$$\mathcal{O}(C^3 \cdot |\mathcal{I}| \cdot |\mathcal{U}| + C^2 \cdot |V| \cdot |\mathcal{Z}| + C^2 P^2 \cdot |\mathcal{U}| + C \cdot |V| \cdot |\mathcal{U}|)$$

clauses. Additionally, denote by $\textsc{Extended}(\mathcal{S}^+, C, P) = \textsc{Extended}^*(\mathcal{S}^+, C, P, \infty, \infty)$ an alias for the call without bounds on T and N.

10) ENCODING THE MAPPING OF NEGATIVE SCENARIO TREE

The mapping $\widehat{\mu}\colon \widehat{V} \to Q_0$ for the negative scenario tree is similar to the positive one. The key difference is that the negative tree may represent behaviors which the automaton does not have. Moreover, it contains *looping* behaviors which the automaton is explicitly prohibited to have.

Variable $\widehat{\mu}_{\widehat{v}} \in Q_0$ denotes the satisfying state (or its absence) of the negative tree node $\widehat{v} \in \widehat{V}$, where $\widehat{\mu}_{\widehat{v}} = q_0$ denotes the absence of a satisfying state and corresponds to the situation when the automaton does not have the behavior represented by the negative tree. The root $\widehat{\rho}$ of the negative tree maps to the initial state of the automaton: $\widehat{\mu}_{\widehat{\rho}} = q_{\text{init}}$.

Passive nodes either (1) map to the same states as their parents, or (2) map to q_0:

$$\left(\widehat{\mu}_{\widehat{p}} = q\right) \to \left(\widehat{\mu}_{\widehat{v}} = q\right) \vee \left(\widehat{\mu}_{\widehat{v}} = q_0\right),$$

where $\widehat{v} \in \widehat{V}^{(\text{pass})}$, $\widehat{p} = \widehat{tp}(\widehat{v})$, $q \in Q$. The first (1) case corresponds to the situation when the automaton ignores the input action:

$$\left(\widehat{\mu}_{\widehat{v}} = q\right) \to \left(\widehat{\lambda}_{q,i,u} = q_0\right),$$

where $\widehat{v} \in \widehat{V}^{(\text{pass})}$, $q \in Q$, $i = \widehat{tie}(\widehat{v})$, $u = \widehat{tin}(\widehat{v})$. And the second (2) case corresponds to the situation when the automaton *actively* reacts to the input action, thus unsatisfying the passive behavior captured in the negative scenario tree.

Similarly, active nodes either map to the state in which the automaton switches after processing an input action, or stay unmapped (*i.e.* mapped to q_0):

$$\left(\widehat{\mu}_{\widehat{p}} = q\right) \to \left(\left(\widehat{\mu}_{\widehat{v}} = q'\right) \leftrightarrow \left(\widehat{\lambda}_{q,i,u} = q'\right) \wedge \left(\widehat{\phi}_{q'} = o\right) \wedge \bigwedge\nolimits_{z \in \mathcal{Z}} \left(\widehat{\gamma}_{q',z,b} = b'\right)\right),$$

where $\widehat{v} \in \widehat{V}^{(\text{act})}$, $\widehat{p} = \widehat{tp}(\widehat{v})$, $q, q' \in Q$, $i = \widehat{tie}(\widehat{v})$, $u = \widehat{tin}(\widehat{v})$, $o = \widehat{toe}(\widehat{v})$, $z \in \mathcal{Z}$, $b = \widehat{tov}(\widehat{p}, z)$, $b' = \widehat{tov}(\widehat{v}, z)$. Note that this constraint requires the equivalence operator $(\leftrightarrow)$ in contrast to the positive one (where implication $(\to)$ in the same place is sufficient): the codomain of $\widehat{\mu}$ is Q_0, but the constraint is only declared for $q' \in Q$. By using the equivalence operator, we can avoid declaring constraints for the special case when $\widehat{\mu}_{\widehat{v}} = q_0$.

Additionally, if some node is unmapped, then this propagates down the tree:

$$\left(\widehat{\mu}_{\widehat{tp}(\widehat{v})} = q_0\right) \to \left(\widehat{\mu}_{\widehat{v}} = q_0\right).$$

Lastly, in order to prohibit the undesired looping behavior represented by back edges, we ensure that the start and the end of each loop either map to different states, or both are unmapped (*i.e.* mapped to q_0):

$$\bigwedge_{\widehat{v}' \in \widehat{tbe}(\widehat{v})} \left[\left(\widehat{\mu}_{\widehat{v}} \neq \widehat{\mu}_{\widehat{v}'}\right) \vee \left(\widehat{\mu}_{\widehat{v}} = \widehat{\mu}_{\widehat{v}'} = q_0\right) \right].$$

11) COMPLETE ALGORITHM

Denote by $\textsc{Complete}^*(\mathcal{S}^+, \mathcal{S}^-, C, P, N, T)$ the procedure for inferring an automaton which satisfies positive scenarios $\mathcal{S}^+$, does not satisfy negative scenarios $\mathcal{S}^-$, has C states and at most T transitions, while each guard condition parse tree consists of at most P nodes, and the total guards size is at most N. This procedure consists of (1) building positive and negative scenario trees, (2) encoding the automaton structure, the mapping of both positive and negative scenario trees, the guard conditions structure, and the cardinality constraints "total number of *not-null* transitions is at most T" and "total size of guard conditions is at most N", and (3) delegating to the SAT solver. The constructed CNF formula consists of

$$\mathcal{O}(C^2 \cdot |\mathcal{I}| \cdot |\mathcal{U}| + C^2 \cdot P \cdot |\mathcal{U}| + C^2 \cdot P \cdot |\widehat{\mathcal{U}}| + C \cdot |V| + C \cdot |\widehat{V}|)$$

variables and

$$\mathcal{O}(C^3 \cdot |\mathcal{I}| \cdot |\mathcal{U}| + C^3 \cdot |\mathcal{I}| \cdot |\widehat{\mathcal{U}}| + C^2 \cdot |V| \cdot |\mathcal{Z}| + C^2 \cdot |\widehat{V}| \cdot |\mathcal{Z}| + C^2 \cdot P^2 \cdot |\mathcal{U}| + C^2 \cdot P^2 \cdot |\widehat{\mathcal{U}}| + C \cdot |V| \cdot |\mathcal{U}| + C \cdot |\widehat{V}| \cdot |\widehat{\mathcal{U}}|)$$

clauses. Additionally, denote by $\textsc{Complete}(\mathcal{S}^+, \mathcal{S}^-, C, P) = \textsc{Complete}^*(\mathcal{S}^+, \mathcal{S}^-, C, P, \infty, \infty)$ an alias for the call without bounds on T and N.

D. MINIMAL MODEL INFERENCE

Proposed methods require the automaton parameters C and P to be known *in advance*. In order to *automate* the inference of minimal models so that the knowledge of C and P is not needed, we use an iterative approach which we describe below.

1) BASIC-MIN ALGORITHM

In order to *quickly* estimate the minimal number of states, we use the $\textsc{Basic}(\mathcal{S}^+, C)$ algorithm by iterating C bottom-up: starting from 1 until we find a solution, an automaton $\mathcal{A}$ with $C_{\min}$ states satisfying $\mathcal{S}^+$. Next, in order to obtain a model with a minimal number of transtions, we use the $\textsc{Basic}^*(\mathcal{S}^+, C_{\min}, T)$ algorithm by iterating T top-down: decreasing it by 1 until an even smaller model could not be found: the solution is the last found automaton with $T_{\min}$ transitions. Let us denote this algorithm, which is formall described in Algorithm 1, as $\textsc{Basic-min}(\mathcal{S}^+)$.

Algorithm 1 $\textsc{Basic-min}(\mathcal{S}^+)$

```
Input: positive scenarios $\mathcal{S}^+$
Output: automaton $\mathcal{A}$ with minimal number of
        states $C_{\min}$ and transitions $T_{\min}$
   // minimize number of states C
1: for $C_{\min} = 1$ to $\infty$ do
2:     $\mathcal{A} \leftarrow \textsc{Basic}(\mathcal{S}^+, C_{\min})$
3:     if $\mathcal{A} \neq$ null then break
   // minimize number of transitions T
4: while $\mathcal{A} \neq$ null do
5:     $\mathcal{A}_{\text{best}} \leftarrow \mathcal{A}$
6:     $T' \leftarrow$ getT$(\mathcal{A}_{\text{best}}) - 1$
7:     $\mathcal{A} \leftarrow \textsc{Basic}^*(\mathcal{S}^+, C_{\min}, T')$
8: return $\mathcal{A}_{\text{best}}$
```

2) EXTENDED-MIN ALGORITHM

Assuming that the parameter P is known and the number of states C is either given or estimated using the $\textsc{Basic-min}$ algorithm, we minimize the total size of guard conditions N using the same top-down iterative approach as during the T minimization described above, but using the $\textsc{Extended}$ algorithm internally. Let us denote the described algorithm as $\textsc{Extended-min}(\mathcal{S}^+, P)$.

Note that we could additionally minimize the total number of not-null transitions T, but in our current implementation we do not do this. The reason for this is the observation that the minimal value of N already implies a small number of transitions in the automaton. Moreover, each naive top-down minimization ends with a proof that the smaller solution does not exist, leaving the SAT solver in the UNSAT state. Hence, in order to perform two consecutive top-down minimizations, we have to reset the solver, redeclare all the constraints and re-solve the corresponding SAT problem from scratch. A well-known technique allowing to overcome this is known as *solving under assumptions*: instead of *declaring* a cardinality constraint, we can *assume* it, and even if the resulting problem is unsatisfiable, the SAT solver will still be operational and able to continue the solving process with (possibly) other assumptions.

3) EXTENDED-MIN-UB ALGORITHM

One can note that P, the maximum allowed size of a single guard condition, is a *required* parameter and has to be provided by the user or an external algorithm. Ultimately, an *automatic* way of determining an appropriate value of parameter P is desirable. Note that the solution, which is an automaton satisfying the given scenarios, exists only when P is large enough to capture the necessary complexity of the guard conditions. The simplest strategy is to iterate P starting from 1 and use $\textsc{Extended-min}(\mathcal{S}^+, P)$ until we find a solution – automaton with $N = N^*_{\min}$ – for some P^*. However, there may exist some value $P' > P^*$ for which the corresponding $N'_{\min}$ is even smaller than $N^*_{\min}$. Therefore, in order to obtain the globally minimal automaton in terms of N, we shall continue the search process for $P > P^*$ up to a theoretical upper bound as described below.

Consider $P = P'$. Ideally, we expect that all guard conditions will be of size 1, and only one of them will be of size P'. Also, ideally, there are exactly $T_{\min}$ guards, therefore, the ideal minimal total size of guard conditions is $N'_{\min} = T_{\min} - 1 + P''$. Let us denote by $N^{\text{best}}_{\min}$ the best, *i.e.* the most minimal value found so far. Ultimately, we are looking for

$N'_{\min} < N^{\text{best}}_{\min}$, thus $T_{\min} - 1 + P' < N^{\text{best}}_{\min}$, from where the upper bound for P is $P' \leq N^{\text{best}}_{\min} - T_{\min}$.

Algorithm 2 EXTENDED-MIN-UB($\mathcal{S}^+$, w)

```
Input: positive scenarios $\mathcal{S}^+$, maximum plateau width $w$
Output: automaton $\mathcal{A}$ with minimal number of states $C_{\min}$ and guard conditions size $N_{\min}$
1: $\mathcal{A}_{\text{basic}} \leftarrow$ BASIC-MIN($\mathcal{S}^+$)
2: $T_{\min} \leftarrow$ getT($\mathcal{A}_{\text{basic}}$)
3: $C_{\min} \leftarrow$ getC($\mathcal{A}_{\text{basic}}$)
4: $N^{\text{best}}_{\min} \leftarrow N^{\text{prev}}_{\min} \leftarrow P_{\text{low}} \leftarrow \infty$
5: for $P = 1$ to $\infty$ do
6:     if $P > (N^{\text{best}}_{\min} - T_{\min})$ then break    /* upper bound reached */
7:     if $(P - P_{\text{low}}) > w$ then break    /* max width reached */
8:     $\mathcal{A} \leftarrow$ EXTENDED-MIN($\mathcal{S}^+$, $C_{\min}$, $P$)
9:     if $\mathcal{A} \neq$ null then
10:        $N_{\min} \leftarrow$ getN($\mathcal{A}$)
11:        if $N_{\min} < N^{\text{best}}_{\min}$ then $N^{\text{best}}_{\min} \leftarrow N_{\min}$    /* update best found $N$ */
12:        if $N_{\min} \neq N^{\text{prev}}_{\min}$ then $P_{\text{low}} \leftarrow P$    /* update local minimum */
13:        $N^{\text{prev}}_{\min} \leftarrow N_{\min}$
14: return $\mathcal{A}$
```

The process of searching P up to the theoretical upper bound can take an extensive amount of time. Hence, we propose the following heuristic. Consider the two successive values P' and $P'' = P' + 1$, and the corresponding values $N'_{\min}$ and $N''_{\min}$. The equality $N'_{\min} = N''_{\min}$ indicates the local minimum (plateau). As we go further by incrementing the value of P'', the remaining equality extends the plateau width. By choosing the critical plateau width w, on which to stop incrementing P, we provide a trade-off between the execution time and global minimality of the solution. When $w = 0$, the algorithm is equivalent to the simplest strategy of searching P until the first SAT. When $w = \infty$, the algorithm continues to iterate P until an upper bound, resulting in globally minimal $N_{\min}$. An arbitrary choice of $w = 2$ has shown a good performance in our initial study. It is worth noting that with this heuristic applied, our proposed method remains *exact* in the sense that the inferred automata still satisfy the given positive scenarios $\mathcal{S}^+$.

Let us denote by EXTENDED-MIN-UB($\mathcal{S}^+$, w) the minimization process described above. It is depicted by Algorithm 2 and consists of two stages. First, we estimate the automaton parameters $C_{\min}$ and $T_{\min}$ using the BASIC-MIN algorithm. Second, we iterate P starting from 1 and use the EXTENDED-MIN algorithm to infer minimal models. We stop the search in two cases: if the current P is greater than the upper bound ($N^{\text{best}}_{\min} - T_{\min}$), or if the current local minumum width is greater than the provided threshold w.

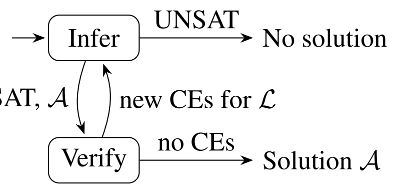


FIGURE 6. CEGIS loop.

E. COUNTEREXAMPLE-GUIDED INDUCTIVE SYNTHESIS

In order to make the inferred automaton not only satisfy given positive scenarios, but also comply with a given LTL specification, we use a counterexample-guided inductive synthesis (CEGIS) [35] iterative approach. On each CEGIS iteration, we infer an automaton $\mathcal{A}$ using the COMPLETE algorithm, verify the LTL specification $\mathcal{L}$ using the NuSMV [30] model checker, and supplement the negative scenario tree with obtained counterexamples, if any. The process shown in Fig. 6 repeats until there are no more counterexamples, thus, the automaton complies with $\mathcal{L}$. Denote by CEGIS*($\mathcal{S}^+$, $\mathcal{L}$, C, P, T, N) the procedure implementing the described inductive synthesis, where arguments are similar to ones in the COMPLETE* algorithm, and $\mathcal{L}$ is an LTL specification. Additionally, denote by CEGIS($\mathcal{S}^+$, $\mathcal{L}$, C, P) $=$ CEGIS*($\mathcal{S}^+$, $\mathcal{L}$, C, P, ∞, ∞) an alias for the call without bounds on T and N.

1) CEGIS-MIN ALGORITHM

Consider an automaton $\mathcal{A}$ produced by the CEGIS algorithm. If we start minimizing the total size of guard conditions N, the automaton will most likely stop complying with the LTL specification, though the already obtained negative scenarios will still not be satisfied. Therefore, we propose to maintain a minimal model on each CEGIS iteration. We begin with a model produced by EXTENDED-MIN-UB($\mathcal{S}^+$, w) (or by EXTENDED($\mathcal{S}^+$, P), if P is provided) and continue by running CEGIS*($\mathcal{S}^+$, $\mathcal{L}$, C^*, P^*, ∞, N^*) with estimated C^*, P^* and N^*, but without a bound on T. The UNSAT result indicates that N^* is too small for an automaton to comply with the given LTL specification $\mathcal{L}$, hence we increase it and continue the CEGIS. Note that this is the only moment we stop solving incrementally, because we weaken the constraints (upper bound for N). Let us denote the described process as CEGIS-MIN($\mathcal{S}^+$, $\mathcal{L}$, w).

F. THE FBSAT TOOL

We implemented the proposed methods in an open-source tool[2] FBSAT, which is written in Kotlin. FBSAT uses a SAT solver to synthesize minimal finite-state models of function blocks from given specifications, and it supports both execution scenarios and LTL properties. To make FBSAT

[2]https://www.github.com/ctlab/fbSAT

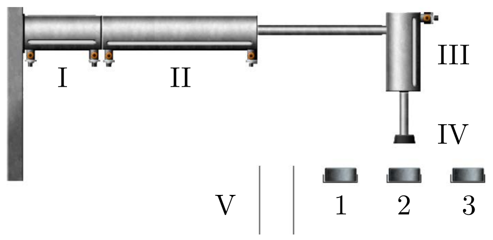


FIGURE 7. Pick-and-Place manipulator.

more flexible and efficient, we have also developed an open-source library[3] called `kotlin-satlib`, which provides a common SAT solver interface for Java/Kotlin and includes native wrappers for several popular SAT solvers, such as MiniSat 2.2, Glucose 4, Cadical 1.4.1, and CryptoMiniSat 5.8.0.

V. CASE STUDY: PICK-AND-PLACE MANIPULATOR

The experimental evaluation of proposed methods was done on a case study devoted to the inference of a finite-state model of the controller for a Pick-and-Place (PnP) manipulator [36] shown in Fig. 7. We also performed an evaluation on random automata (Section VII) and transition system minimization (Section VI). Experiments were conducted on a computer with an Intel(R) Core^{TM} i5-7200U CPU @ 2.50 GHz and 8 GB of RAM.

The PnP manipulator consists of two horizontal pneumatic cylinders (I, II), one vertical cylinder (III), and a suction unit (IV) for picking up work pieces (WPs). When a WP appears on one of the input sliders (1, 2, 3), the horizontal cylinders position the suction unit on top of the WP, the vertical cylinder lowers the suction unit where it picks up the WP and then moves in to the output slider (V). The control system is implemented using IEC 61499 function blocks in nxtSTUDIO. The controller is a basic FB with 10 input and 7 output Boolean variables. Input variables include: `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), `pp1`/`pp2`/`pp3` (is a WP present on input slider 1/2/3), `vac` (is the vacuum unit IV on). Output variables include: `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). The purpose of this case study was to infer a finite-state model of this controller FB. The process of capturing scenarios for the PnP manipulator controller is described by [26]. We used sets of scenarios of various sizes: 1, 10, 39 and 49 scenarios in each.

A. INFERENCE OF MINIMAL AUTOMATA FROM POSITIVE SCENARIOS

In the first set of experiments, we compare methods that infer minimal models from positive scenarios with explicit regard of the guard conditions size. Our method was compared to the TWO-STAGE approach from [28], where on the first stage a basic automaton model is inferred with a SAT solver, and then this model's guard conditions are minimized with a CSP solver *w.r.t.* given scenarios. Note that the TWO-STAGE method has already been shown to be superior to EFSM-tools.

We apply the proposed EXTENDED-MIN-UB method to infer an automaton with the minimal number of states $C_{\min}$ and total size of guard conditions $N_{\min}$. Three values of the parameter w were used: $w = 0$ for the case when first solution found is considered final, $w = 2$ for the case with the proposed heuristic applied, and $w = \infty$ for the "without heuristic" case. Results are summarized in Table 1, where for TWO-STAGE: $C_{\min}$ is the minimal number of states, $T_{\min}$ is the minimal number of transitions, $N_{\min}$ is the minimal total size of guard conditions; and for EXTENDED-MIN-UB: w is the maximum width of local minima plateau, P is the maximum guard condition size, T is the number of transitions, $N_{\min}$ is the minimal total size of guard conditions. Results indicate that EXTENDED-MIN-UB produces compact automata: in studied cases, $w = 2$ already gives optimal results in terms of $N_{\min}$.

B. COMPARISON WITH LTL SYNTHESIS TOOLS

We considered tools BoSy [7] and G4LTL-ST [12], which accept LTL specifications as input. Comparison was only done for synthesis from scenarios, which were converted to LTL formulas. For BoSy, we considered a simplified version of scenario $\mathcal{S}^{(1)}$, for which passive elements were removed, leaving only 8 scenario elements. The input-symbolic version of BoSy was the only one that worked for this example, generating a solution with 9 states and 17 transitions in 273 s. For G4LTL-ST, we selected the number of unroll steps (a required parameter of this tool) equal to the length of the largest scenario. For $\mathcal{S}^{(1)}$, a solution with 10 states (though with verbose guard conditions) was found in 10 s. Larger sets of scenarios required 16 unroll steps, and runs failed with a memory limit of 8 GB. As expected, experiments showed that LTL synthesis tools are not well-suited for inference of models from finite-length scenarios. Experiments with LTL properties were not considered due to (1) poor performance on scenarios, and (2) lack of support for general-form NuSMV plant model, which is crucial for synthesis from liveness properties.

C. INFERENCE OF AUTOMATA FROM POSITIVE SCENARIOS AND LTL PROPERTIES

The third set of experiments is devoted to CEGIS. In order to use the liveness LTL properties, the verification of candidate models with NuSMV was performed in a closed loop [37] with a manually prepared formal model of the plant – PnP manipulator. This model defines the state of the plant and its actions as implied by the controller commands. The set of considered LTL properties (see Table 3) includes safety properties $\varphi_1, \ldots, \varphi_6$ ("controller does not lead the system to an unsafe state") and liveness properties $\varphi_7, \ldots, \varphi_{10}$ ("something useful eventually happens"). Properties $\varphi_1, \ldots, \varphi_7$ are

[3]https://www.github.com/Lipen/kotlin-satlib

TABLE 1. Inference of automata with minimal guard conditions from positive scenarios.

$\mathcal{S}^+$	$\lvert\mathcal{T}^+\rvert$	TWO-STAGE [28]				FBSAT: EXTENDED-MIN-UB ($w=0$)				FBSAT: EXTENDED-MIN-UB ($w=2$)				FBSAT: EXTENDED-MIN-UB ($w=\infty$)			
		time, s.	$C_{\min}$	$T_{\min}$	$N_{\min}$	time, s.	P	T	$N_{\min}$	time, s.	P	T	$N_{\min}$	time, s.	P	T	$N_{\min}$
$\mathcal{S}^{(1)}$	24	8	6	8	15	2	3	8	14	3	3	8	14	3	3	8	14
$\mathcal{S}^{(10)}$	234	3	8	17	36	16	3	18	38	51	5	16	25	88	5	16	25
$\mathcal{S}^{(39)}$	960	13	8	15	32	37	3	18	38	118	5	16	25	170	5	16	25
$\mathcal{S}^{(49)}$	2939	36	8	18	60	602	5	18	44	4305	6	16	39	51666	6	16	39

TABLE 2. Temporal properties for the Pick-and-Place system.

Property	Description
Fixed part	
$\varphi_1 = \mathbf{G}(\neg(\texttt{c1Extend} \wedge \texttt{c1Retract}))$	Cylinder I must not be issued commands to extend and retract simultaneously.
$\varphi_2 = \mathbf{G}(\neg(\texttt{c2Extend} \wedge \texttt{c2Retract}))$	Similar property for cylinder II.
$\varphi_3 = \mathbf{G}(\neg(\texttt{vacuum\_on} \wedge \texttt{vacuum\_off}))$	Similar property for the vacuum unit.
$\varphi_4 = \mathbf{G}(\neg\texttt{vcHome} \wedge \neg\texttt{vcEnd} \rightarrow \texttt{c1Home} \vee \texttt{c1End})$	If the vertical cylinder is in the intermediate position, cylinder I must be either in home or end position.
$\varphi_5 = \mathbf{G}(\neg\texttt{c1Home} \wedge \neg\texttt{c1End} \rightarrow \texttt{vcHome} \vee \texttt{vcEnd})$	If cylinder I is in the intermediate position, the vertical cylinder must be either in home or end position.
$\varphi_6 = \mathbf{G}(\texttt{all\_home} \wedge \neg\texttt{pp1} \wedge \neg\texttt{pp2} \wedge \neg\texttt{pp3} \wedge \neg\texttt{lifted} \rightarrow \mathbf{X}(\neg\texttt{c1Extend} \wedge \neg\texttt{c2Extend} \wedge \neg\texttt{vcExtend}))$	If all cylinders are in home position and no WP should be processed, no commands to move any cylinders should be issued.
$\varphi_7 = \mathbf{G}(\texttt{lifted} \rightarrow \mathbf{F}(\texttt{dropped}))$	If a WP is lifted from the input slider it must eventually be dropped to the output slider.
Variable part (one at a time)	
$\varphi_8 = \mathbf{G}(\texttt{pp1} \rightarrow \mathbf{F}(\texttt{vp1}))$	If a WP appears on slider 1, it must be eventually lifted.
$\varphi_9 = \mathbf{G}(\texttt{pp2} \rightarrow \mathbf{F}(\texttt{vp2}))$	If a WP appears on slider 2, it must be eventually lifted.
$\varphi_{10} = \mathbf{G}(\texttt{pp3} \rightarrow \mathbf{F}(\texttt{vp3}))$	If a WP appears on slider 3, it must be eventually lifted.

fixed and used in all experiments, while the use of $\varphi_8, \ldots, \varphi_{10}$ varies. We focus on these last properties, which define that whenever a WP is placed on some input slider, it will eventually be removed. Note that for the original PnP system [36] only φ_8 is satisfied, while φ_9 and φ_{10} are false (the controller is not wait-free for sliders 2 and 3: if a WP is always present on slider 1, WPs from sliders 2 and 3 will never be picked up). Therefore, we consider the property for each input slider separately, assuming that WPs never appear on other input sliders. For the experiment with φ_9, we use a special set of scenarios $\mathcal{S}^{(1)''}$, which consists of a single scenario describing the processing of the WP from slider 2. Likewise, for φ_{10}, set $\mathcal{S}^{(1)'''}$ is used, which describes a one-time WP processing from slider 3.

Three algorithms are compared: the proposed CEGIS-MIN and CEGIS, and the CEGIS-extension of FBCSP [27], which we refer to as FBCSP+LTL. Note that EFSM-tools, BoSy and G4LTL-ST are not considered here, mainly due to the poor performance on scenarios only. For our algorithms we use $w = 2$, as this value has shown a good performance in our initial study. For a SAT solver we use MiniSat. Apart from running time and N, we measure N_{init} (for the automaton initially built only from positive scenarios using EXTENDED-MIN-UB) and the number of CEGIS iterations (#iter). Experimental results are summarized in Table 3. The automaton generated using the CEGIS-MIN algorithm from scenarios $\mathcal{S}^{(1)}$ and LTL specification $\varphi_1, \ldots, \varphi_7, \varphi_8$ is shown in Fig. 8.

Solutions found with CEGIS methods are always larger than ones constructed from scenarios only (in terms of N). This indicates that the used sets of scenarios are incomplete and do not cover considered specifications completely. Then, CEGIS-MIN always finds the smallest solutions and is always faster than FBCSP+LTL. Most interestingly, CEGIS-MIN allows for efficient construction of models for scenarios $\mathcal{S}^{(1)}$, $\mathcal{S}^{(1)''}$, and $\mathcal{S}^{(1)'''}$ – these scenarios do not "cover" corresponding liveness properties of interest (*e.g.*, $\varphi_8 = \mathbf{G}(\texttt{pp1} \rightarrow \mathbf{F}(\texttt{vp1}))$) in the sense that each scenario describes only a single processing of a WP. The existing method FBCSP+LTL

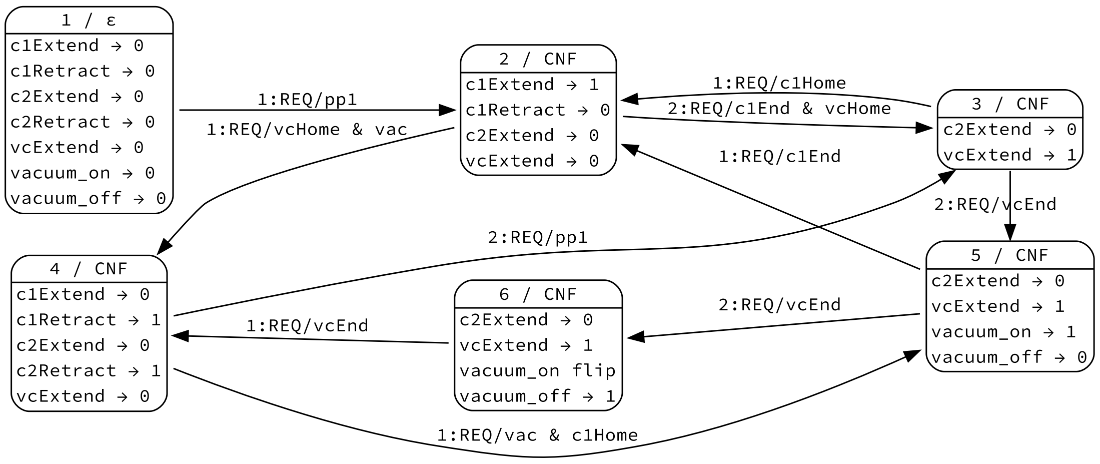


FIGURE 8. Automaton generated by CEGIS-MIN algorithm from $\mathcal{S}^{(1)}$ and $\varphi_1, \ldots, \varphi_7, \varphi_8$.

TABLE 3. Results of CEGIS experiments for PnP controller inference.

LTL properties	$\mathcal{S}$	N_{init}	CEGIS-MIN				CEGIS			FBCSP+LTL		
			time, s	#iter	P	$N_{\min}$	time, s	#iter	N	time, s	#iter	$N_{\min}$
$\varphi_1, \ldots, \varphi_7, \varphi_8$	$\mathcal{S}^{(1)}$	14	56	273	3	16	45	151	34	>12h	>500	–
	$\mathcal{S}^{(10)}$	25	250	20	5	28	87	55	152	613	10	40
	$\mathcal{S}^{(39)}$	25	350	70	5	28	155	56	150	1019	2	41
$\varphi_1, \ldots, \varphi_7, \varphi_9$	$\mathcal{S}^{(1)\prime\prime}$	14	361	193	3	16	192	430	32	>12h	>500	–
$\varphi_1, \ldots, \varphi_7, \varphi_{10}$	$\mathcal{S}^{(1)\prime\prime\prime}$	14	476	353	3	16	30	57	34	>12h	>500	–

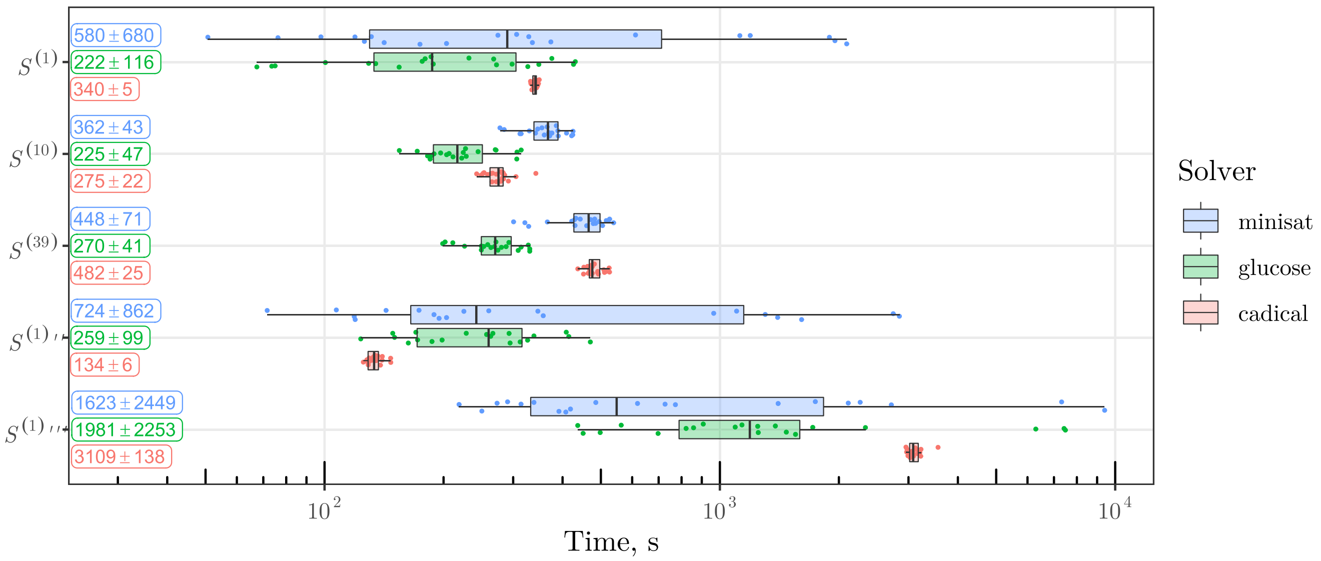


FIGURE 9. Inference time distribution for the CEGIS-MIN method with different SAT solvers (Minisat 2.2, Glucose 4, Cadical 1.4.1) using 20 different random seeds. Numbers on the left denote the mean and standard deviation ($\mu \pm \sigma$).

failed on these cases, while the proposed approach succeeds with ease. Lastly, the CEGIS algorithm allows constructing models fast, but losing the guard conditions minimality.

Additionally, we repeated the CEGIS-MIN evaluation with different SAT solvers using 20 different random seeds. For MiniSat and Glucose, we specified the following options: `random_var_freq=0.1`, `rnd_pol=true`,

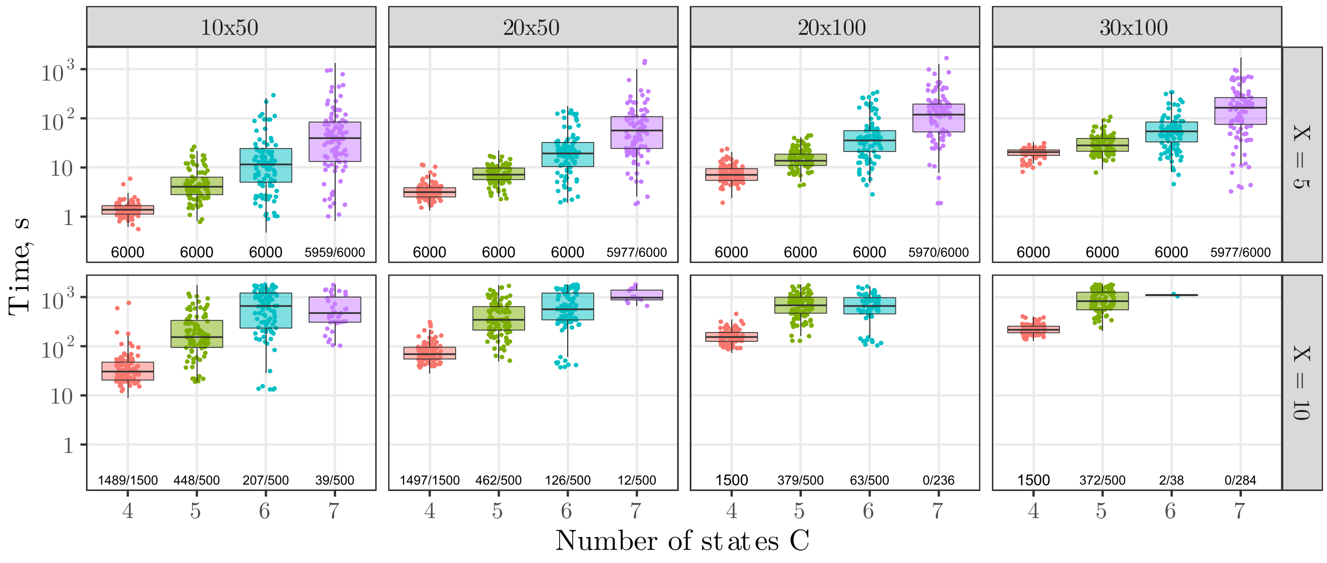


FIGURE 10. Evaluation results for random automata synthesis. Each boxplot represents the inference time distribution of SAT instances finished within a 30 min timeout. The numbers below each boxplot denote the number of SAT runs out of the total.

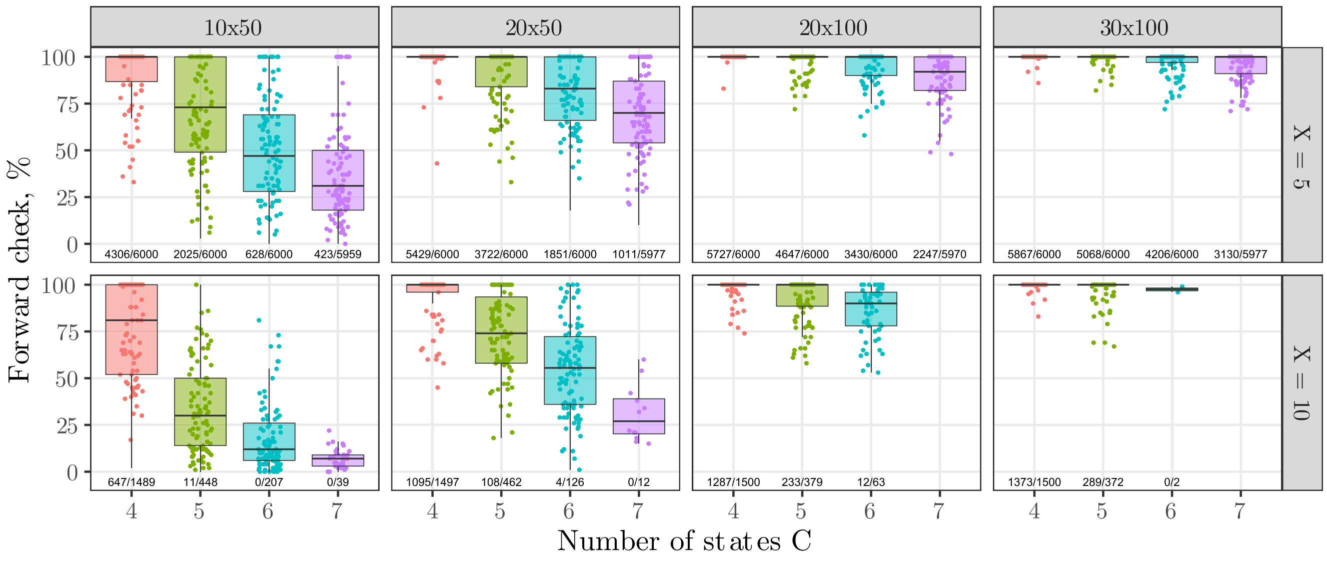


FIGURE 11. Validation results for random automata synthesis. Each boxplot represents the distribution of "forward check" validation (percentage of scenarios that the automaton satisfies). The numbers below each boxplot denote the number of 100%-validated runs out of the total SAT instance.

rnd_init_act=true. The resulting inference time distributions are shown in Fig. 9. The high variance of inference time for MiniSat/Glucose can be explained as follows: CEGIS is basically a model guessing process, and any deviations during it could result in different number of iterations (thus, in different running time, since they are generally proportional), especially when a small set of positive scenarios is used, *e.g.*, $\mathcal{S}^{(1)}$. In contrast, the low variance of Cadical results indicates that probably either Cadical is not employing randomness too much, or the models it produces are not heavily dependent on this randomness, so CEGIS converges in a constant number of iterations.

VI. CASE STUDY: SYNTCOMP BENCHMARK

As another case study and to demonstrate the applicability of FBSAT to problems other than the synthesis of function block models, in this section we show an example of FBSAT being used to synthesize transition systems using benchmakrs from the SYNTCOMP reactive synthesis competition [38].

One of the SYNTCOMP competition tracks, the sequential synthesis track, is devoted to the synthesis of a transition system from a given LTL specification – the so-called LTL synthesis. There is a large variety of LTL synthesis tools available including BoSy [7] and Strix [25]. To the best of our knowledge, out of all LTL synthesis tools, only BoSy limits the size of generated transition systems (number of states), but it does not try to minimize the size of generated guard conditions, which tend to be large and incomprehensible.

Note that FBSAT in its current version is not directly applicable to the LTL synthesis problem, as it requires positive execution scenarios, which LTL synthesis does not use. However, here we show how to apply FBSAT for *minimization* of transition systems generated by LTL synthesis tools, *e.g.*, BoSy.

A transition system $\mathcal{T}$ [7] is a tuple (T, t_0, Σ, τ), where T is a finite set of states, $t_0 \in T$ is the initial state, $\Sigma = I \cup O$ is an input/output interface, I is a finite set of propositional variables controllable by the environment (*inputs*), O is a finite set of propositional variables controllable by system (*outputs*), and $\tau : T \times 2^I \to 2^O \times T$ is a transition function, which maps a state $t \in T$ and a valuation of an input vector $\boldsymbol{i} \in 2^I$ to a valuation of an output vector $\boldsymbol{o} \in 2^O$ and a next state t'.

From analysis of this definition, one may note the similarity between the transition system and the ECC of the basic function block. Overall, if the transition system is of the Moore type (*i.e.* for any $t \in T$ and $\boldsymbol{i} \neq \boldsymbol{i'} \in 2^I$ with $\tau(t, \boldsymbol{i}) = (\boldsymbol{o}, \_)$ and $\tau(t, \boldsymbol{i'}) = (\boldsymbol{o'}, \_)$ it holds that $o = o'$), it can be easily modeled using an ECC.

In this case study, we started from the `full_arbiter_3` instance from SYNTCOMP-2018 and used BoSy (input-symbolic QBF-based version) to generate a transition system satisfying the specification. The resulting transition system $\mathcal{A}_{\text{original}}$ was saved in the NuSMV format and had $C = 8$ states, $T = 28$ transitions with a total size of guard conditions $N = 147$. The graphical representation of the resulting transition system is shown in Fig. 12. Then we used NuSMV to automatically generate 20 execution scenarios, each of length 20, covering all transition system states. The goal was to minimize the size of guard conditions while preserving the transition system's compliance with the LTL specification.

Note that the definition of the transition systems does not constrain the transition function to be deterministic. However, FBSAT is currently only applicable to the synthesis of deterministic state machines. Therefore, for this case study we needed deterministic execution scenarios. In order to obtain them, we augmented BoSy with corresponding constraints to make the transition function deterministic.

Two experiments were conducted. In the experiments, FBSAT was given as input the generated execution scenarios and the LTL specification of the SYNTCOMP instance, and the CEGIS-MIN algorithm was used. In the first experiment, we ensured that the minimized transition system generated by FBSAT is deterministic. Since the definition of the transition system does not include a transition priority function (as in the ECC), it implies that guard conditions of transitions originating from one state must not have a common satisfying assignment. This additional constraint was added to FBSAT for the purpose of this case study. As a result, we were able to find a solution $\mathcal{A}_{\text{deterministic}}$ shown in Fig. 13 with the same number of states and transitions, but with a smaller total size of guard conditions $N = 105$.

In the second experiment we removed the determinism constraint described above. We can do this because, as mentioned above, the transition system definition does not require determinism. This way, the generated transition system will be deterministic as an ECC (due to transition priority), but will be non-deterministic as a transition system. As a result, for the generated solution $\mathcal{A}_{\text{non-deterministic}}$ shown in Fig. 14 the size of the guard conditions was greatly reduced down to $N = 52$.

These results indicate that our approach of explicit tree representation of guard conditions allows to sufficiently minimize the guard conditions size. It can potentially be applied not only after LTL synthesis: it should be possible to augment BoSy encodings [7] with our encodings for guard conditions and minimize them during synthesis.

VII. CASE STUDY: RANDOM AUTOMATA

In order to test our developed tool FBSAT on more instances, we perform its evaluation on randomly generated automata in this case study.

The first step is to generate random automata. Initially, we wanted to choose the automaton parameters similar to the parameters of the model inferred in "Case Study: PnP Manipulator" (Section V): number of states $C = 8$, one input and one output event, $|\mathcal{X}| = 10$ input and $|\mathcal{Z}| = 7$ output variables. However, our preliminary experiments with these parameters has shown the predictable strong hardness of the random automata synthesis problem. Hence, we chose to study simpler models with the following parameters: number of states $C \in [4..7]$, number of transitions $T = C^2$ (each state has a transition to every other state), guard condition max size $P = 5$, and number of input and output variables $|\mathcal{X}| \in \{5, 10\}$ and $|\mathcal{Z}| = 5$, respectively. We generate up to 100 random automata for each combination of C and $|\mathcal{X}|$. Our preliminary experiments with more complex automata showed the predictable strong hardness of the random automata synthesis problem, so we chose to study simpler models instead. The goal of this case study is to explore the problem landscape and determine the general applicability of FBSAT as a model synthesis tool.

The second step is to simulate execution scenarios. We start at the initial automaton state and sequentially select random inputs. The automaton reacts to these input actions and produces output actions, forming an execution scenario. Note that this random walk corresponds to a situation when the plant has random dynamics. Hence, these randomly simulated instances are most likely harder than real-world instances, since real-world plants (such as the PnP manipulator) do not have random dynamics. We simulate sets of scenarios of the following sizes: 10×50, 20×50, 20×100, 30×100 ("count" $\times$ "length").

The next step is to infer the minimal automaton from the simulated scenarios. For this, we employ the EXTENDED-MIN-UB algorithm with a value of $w = 0$, along with the Glucose SAT solver with the following options: `random_var_freq=0.01`, `rnd_pol=true`,

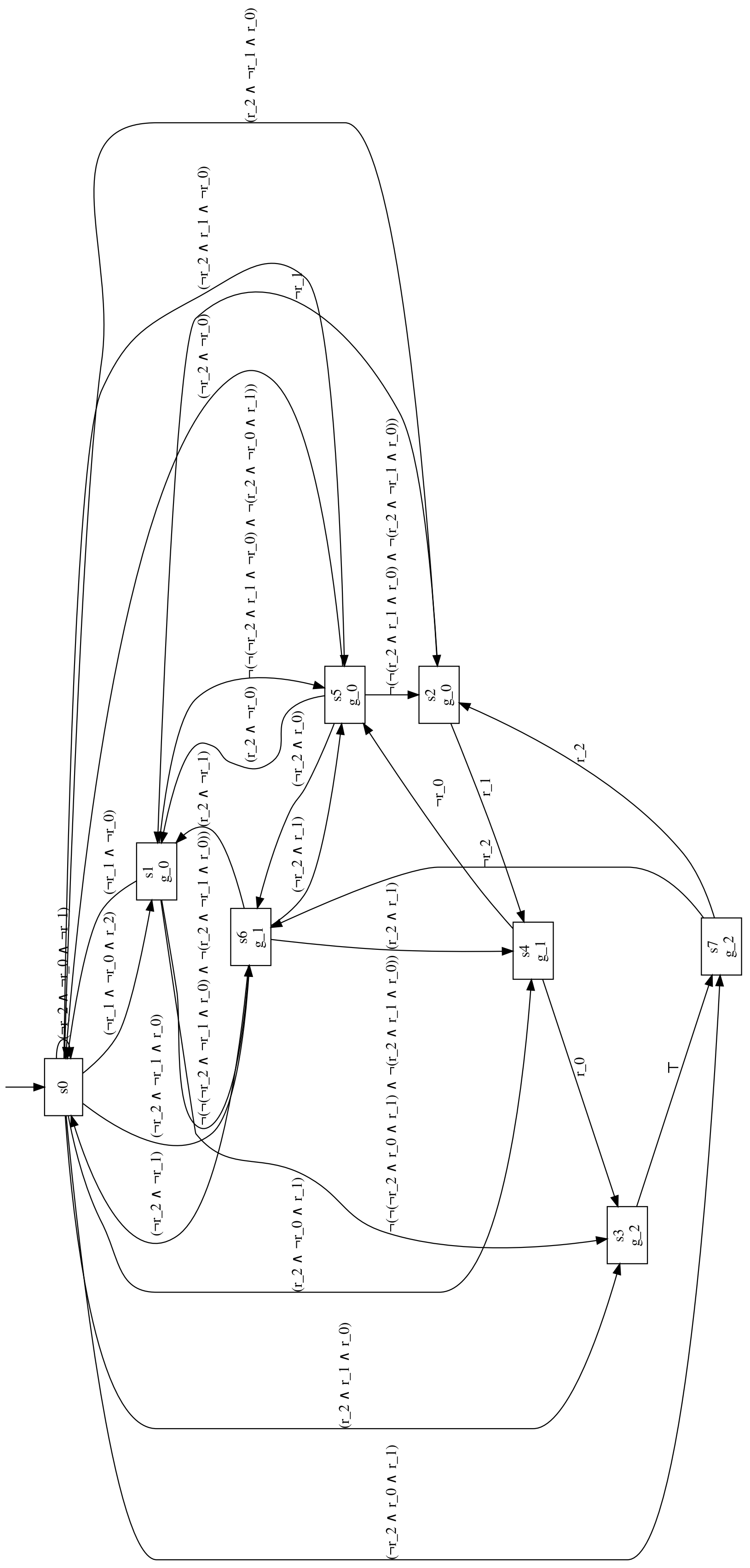


FIGURE 12. Transition system $\mathcal{A}_{\text{original}}$ generated by BoSy for the `full_arbiter_3` instance has $C = 8$ states, $T = 28$ transitions with guard conditions of size $N = 147$.

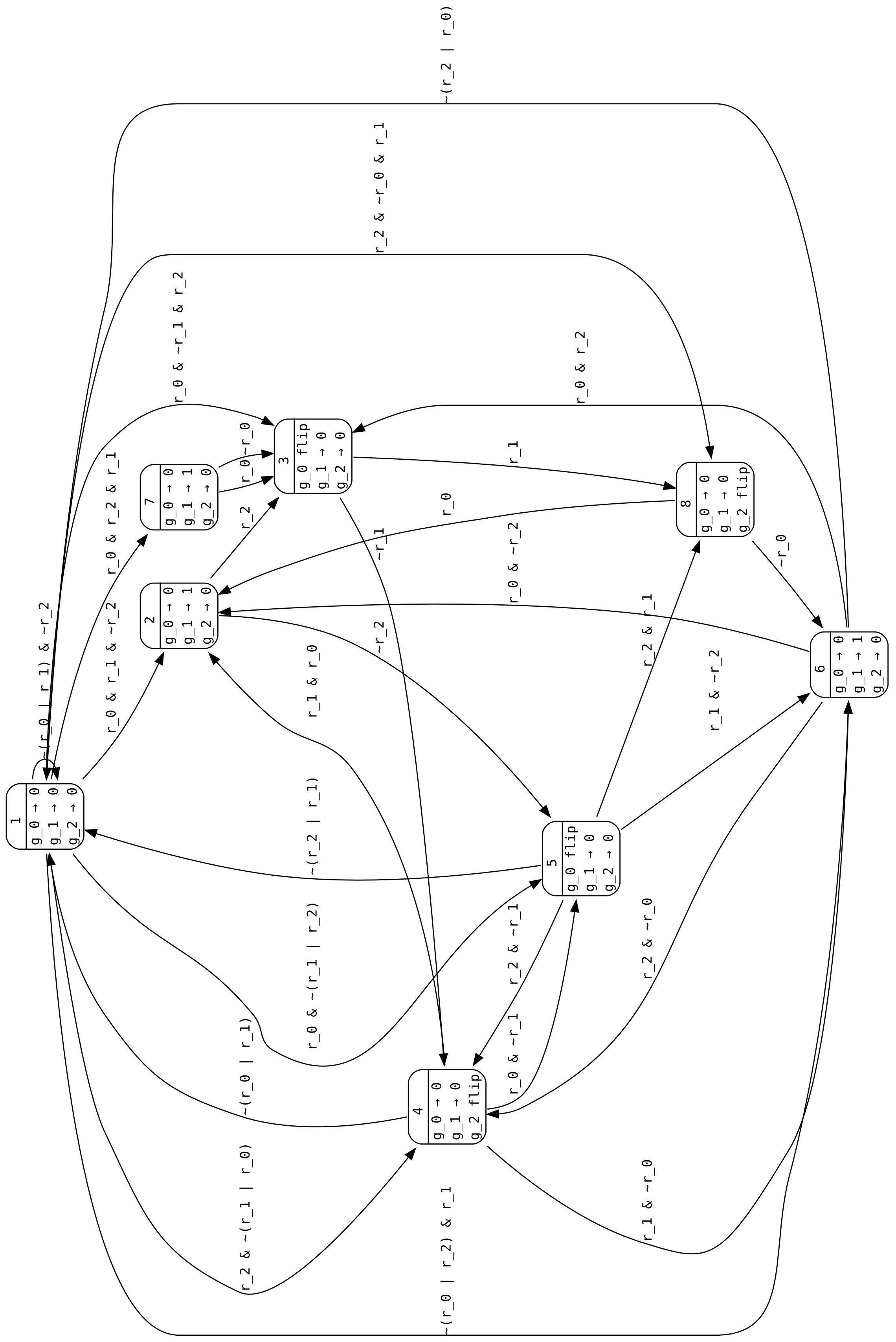


FIGURE 13. Deterministic minimized transition system $\mathcal{A}_{\text{deterministic}}$ generated by FBSAT has smaller guard conditions of size $N = 105$.

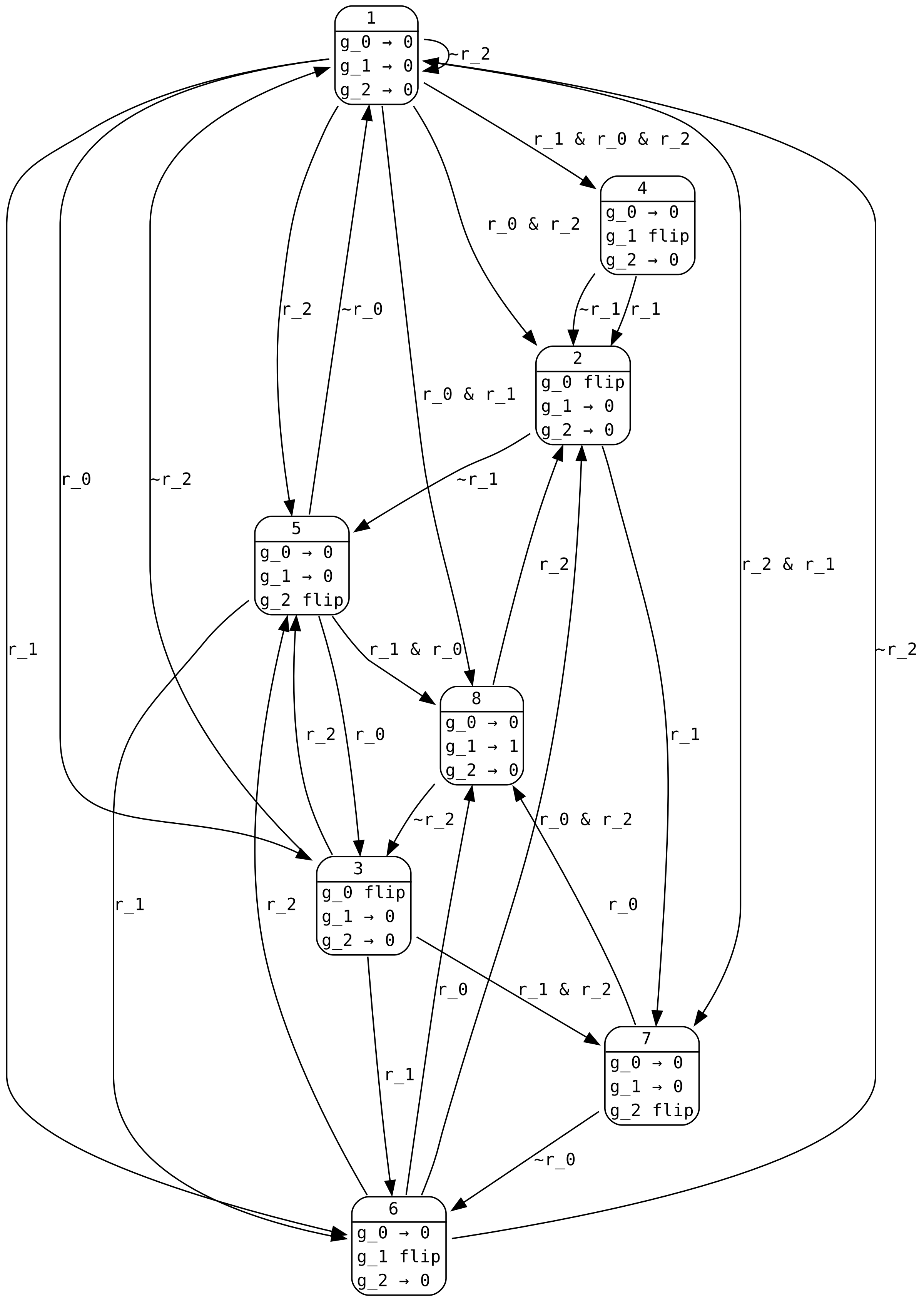


FIGURE 14. Non-deterministic minimized transition system $\mathcal{A}_{\text{non-deterministic}}$ generated by FBSAT has much smaller guard conditions of size $N = 52$.

`rnd_init_act=true`, and `timeout=1800s` (30 minutes). In addition, since we observed in Section V-C that different random seeds can greatly affect the results, we rerun the inference 3 times using different random seeds for the solver. It should be noted that we do not currently aim to study the effect of using random seeds.

The final step is to validate the inferred automaton using the "forward check" validation approach from [39]. This approach involves generating a large set of validation scenarios, denoted by $\mathcal{S}^{100\times100}_{\text{validation}}$, and checking whether the inferred automaton satisfies them. The metric used to evaluate the results of the validation is the percentage of scenarios that the automaton satisfies. It is expected that a high coverage of target automata by scenarios will result in good validation results.

The experimental results are shown in Fig. 10. For $C \in [4..6]$ and $|\mathcal{X}| = 5$, all $100 \cdot 20 \cdot 3 = 6000$ runs finished with a SAT result. For other parameter values, some runs timed out after 30 minutes. Each boxplot represents the inference time distribution only for SAT instances finished within a timeout. The numbers below each boxplot (*e.g.*, "5959/6000") denote the number of SAT runs out of the total, whereas a single number denotes the absence of timeouts. Additionally, each boxplot is supplied with (up to) 100 random sampled points to render the overall distribution. We observe that the inference time depends at least exponentially on the size of the input data, specifically on the number of states C, the number of input variables $|\mathcal{X}|$, and the size of execution scenarios.

The validation results are shown in Fig. 11. Each boxplot represents the distribution of "forward check" in percent, and the numbers below each boxplot denote the number of 100%-validated runs out of the total SAT instances. As expected, larger scenarios provide higher model coverage, thence, higher "forward check". Additionally, the larger the model, the larger the size of scenarios necessary to capture the original model behavior. These results can be extrapolated to non-random models and used as a reference, for example, when inferring a model with $C = 6$ states, it is best to use scenarios of size at least 20×100.

VIII. CONCLUSION AND FUTURE WORK

We have proposed a SAT-based approach for inference of minimal FB models from execution scenarios and LTL properties, and implemented it in the tool FBSAT. The proposed approach is the only one that allows direct minimization of guard conditions complexity of synthesized automata. In particular, the EXTENDED-MIN-UB algorithm is guaranteed to find the solution with globally minimal complexity of guard conditions. Experiments showed that the suggested approach outperforms existing ones and demonstrates predictable scalability on random instances.

The proposed methods allow to infer minimal finite-state models of function blocks from a given specification: execution scenarios and LTL properties. The algorithms have an exponential worst-case complexity, since they rely on SAT solvers to solve an NP-hard problem. The scenarios are either derived from existing systems under learning (SULs) or generated by simulating existing system models, while the LTL properties are written manually. It is important to note that synthesized models are not always equivalent to the original SUL and may exhibit different behavior in situations not covered by the specification. We assume that the given specification covers the necessary behavior. Note that it is not possible to formally verify the conformance of the SUL to the LTL specification or the equivalence of the synthesized model to the SUL, since we do not have explicit access to the SUL (*i.e.* to its internal structure and source code) due to the purely "passive learning" industrial problem statement.

Future research may include synthesizing modular automata and exploring other encodings for integer variables and cardinality constraints. Additionally, the developed encodings of tree-form arbitrary guard conditions can be applied to other SAT-based methods for state machine synthesis, such as augmenting BoSy [7] to reduce the size of generated transition systems. Another possible direction is parallelizing the proposed approach. For example, it is possible to perform iterations (such as those in BASIC-MIN) in parallel by launching several independent instances of the algorithm for each value of C in a specified range, or by using an incremental parallel SAT solver.

APPENDIX FIGURES

The last pages of this manuscript contain several large figures (namely, 12, 13, and 14) referenced throughout the paper.

REFERENCES

[1] L. Apfelbaum and J. Doyle, "Model based testing," in *Proc. Softw. Quality Week Conf.*, 1999, pp. 296–300.

[2] L. Marsso, R. Mateescu, and W. Serwe, "TESTOR: A modular tool for on-the-fly conformance test case generation," in *Tools and Algorithms for the Construction and Analysis of Systems*. Cham, Switzerland: Springer, 2018, pp. 211–228.

[3] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1521–1530, Aug. 2017.

[4] E. Lee, Y.-G. Kim, Y.-D. Seo, K. Seol, and D.-K. Baik, "RINGA: Design and verification of finite state machine for self-adaptive software at runtime," *Inf. Softw. Technol.*, vol. 93, pp. 200–222, Jan. 2018.

[5] 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, Nov. 2011.

[6] M. J. H. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Grammatical Inference: Theoretical Results and Applications*. Berlin, Germany: Springer, pp. 66–79.

[7] P. Faymonville, B. Finkbeiner, and L. Tentrup, "BoSy: An experimentation framework for bounded synthesis," in *Computer Aided Verification*. Cham, Switzerland: Springer, 2017, pp. 325–332.

[8] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *Int. J. Softw. Tools Technol. Transf.* vol. 20, no. 1, pp. 35–55, Feb. 2018.

[9] V. Ulyantsev, I. Zakirzyanov, and A. Shalyto, "BFS-based symmetry breaking predicates for DFA identification," in *Language and Automata Theory and Applications*. Cham, Switzerland: Springer, 2015, pp. 611–622.

[10] G. Giantamidis and S. Tripakis, "Learning Moore machines from input–output traces," in *Formal Methods*. Cham, Switzerland: Springer, 2021, pp. 291–309.

[11] F. Avellaneda and A. Petrenko, "FSM inference from long traces," in *Formal Methods*. Cham, Switzerland: Springer, pp. 93–109.

[12] C.-H. Cheng, C.-H. Huang, H. Ruess, and S. Stattelmann, ''G4LTL-ST: Automatic generation of PLC programs,'' in *Computer Aided Verification*. Cham, Switzerland: Springer, pp. 541–549.

[13] R. Smetsers, P. Fiterău-Broştean, and F. Vaandrager, ''Model learning as a satisfiability modulo theories problem,'' in *Language and Automata Theory and Applications*. Cham, Switzerland: Springer, 2018, pp. 182–194.

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

[15] *NxtControl*. Accessed: Dec. 11, 2022. [Online]. Available: https://www.nxtcontrol.com/en/engineering

[16] E. M. Gold, ''Complexity of automaton identification from given data,'' *Inf. Control*, vol. 37, no. 3, pp. 302–320, 1978.

[17] R. Rosner, ''Modular synthesis of reactive systems,'' Ph.D. thesis, Wiezman Inst. Sci., Rehovot, Israel, 1991, p. 104.

[18] D. Neider and U. Topcu, ''An automaton learning approach to solving safety games over infinite graphs,'' in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Germany: Springer, 2016, pp. 204–221.

[19] F. Coste and J. Nicolas, ''Regular inference as a graph coloring problem,'' in *Proc. Workshop Grammatical Inference, Automata Induction, Lang. Acquisition*, 1997, pp. 1–6.

[20] I. Zakirzyanov, A. Morgado, A. Ignatiev, V. Ulyantsev, and J. Marques-Silva, ''Efficient symmetry breaking for SAT-based minimum DFA inference,'' in *Tools and Algorithms for the Construction and Analysis of Systems*. Cham, Switzerland: Springer, 2019, pp. 159–173.

[21] F. Tsarev and K. Egorov, ''Finite state machine induction using genetic algorithm based on testing and model checking,'' in *Proc. 13th Annu. Conf. Companion Genetic Evol. Comput. (GECCO)*, 2011, pp. 759–762.

[22] A. Petrenko, F. Avellaneda, R. Groz, and C. Oriat, ''FSM inference and checking sequence construction are two sides of the same coin,'' *Softw. Qual. J.*, pp. 651–674, 2019, doi: 10.1007/s11219-018-9429-3.

[23] 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, 2015.

[24] B. Finkbeiner and F. Klein, ''Bounded cycle synthesis,'' in *Computer Aided Verification*. Cham, Switzerland: Springer, 2016, pp. 118–135.

[25] P. J. Meyer, S. Sickert, and M. Luttenberger, ''Strix: Explicit reactive synthesis strikes back!'' in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham, Switzerland: Springer, 2018, pp. 578–586.

[26] D. Chivilikhin, V. Ulyantsev, A. Shalyto, and V. Vyatkin, ''CSP-based inference of function block finite-state models from execution traces,'' in *Proc. IEEE 15th Int. Conf. Ind. Informat. (INDIN)*, Jul. 2017, pp. 714–719.

[27] D. Chivilikhin, I. Buzhinsky, V. Ulyantsev, A. Stankevich, A. Shalyto, and V. Vyatkin, ''Counterexample-guided inference of controller logic from execution traces and temporal formulas,'' in *Proc. IEEE 23rd Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2018, pp. 91–98.

[28] D. Chivilikhin, V. Ulyantsev, A. Shalyto, and V. Vyatkin, ''Function block finite-state model identification using SAT and CSP solvers,'' *IEEE Trans. Ind. Informat.*, vol. 15, no. 8, pp. 4558–4568, Aug. 2019.

[29] T. Klenze, S. Bayless, and A. J. Hu, ''Fast, flexible, and minimal CTL synthesis via SMT,'' in *Computer Aided Verification*. Cham, Switzerland: Springer, 2016, pp. 136–156.

[30] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, ''NUSMV: A new symbolic model checker,'' *Int. J. Softw. Tools Technol. Transf.*, vol. 2, no. 4, pp. 410–425, 2000.

[31] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[32] T. Walsh, ''SAT v CSP,'' in *Proc. 6th Int. Conf. Principles Practice Constraint Program.* Berlin, Germany: Springer, 2000, pp. 441–456.

[33] M. Björk, ''Successful SAT encoding techniques,'' *J. Satisfiability, Boolean Model. Comput.*, vol. 7, no. 4, pp. 189–201, Jul. 2009.

[34] O. Bailleux and Y. Boufkhad, ''Efficient CNF encoding of Boolean cardinality constraints,'' in *Principles and Practice of Constraint Programming*. Berlin, Germany: Springer, 2003, pp. 108–122.

[35] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, ''Combinatorial sketching for finite programs,'' *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 404–415, Nov. 2006.

[36] S. Patil, V. Vyatkin, and M. Sorouri, ''Formal verification of intelligent mechatronic systems with decentralized control logic,'' in *Proc. IEEE 17th Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2012, pp. 1–7.

[37] V. Vyatkin, H. M. Hanisch, C. Pang, and C. H. Yang, ''Closed-loop modeling in future automation system engineering and validation,'' *IEEE Trans. Syst., Man, C, Appl. Rev.*, vol. 39, no. 1, pp. 17–28, Jan. 2009.

[38] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker, ''The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results,'' 2019, *arXiv:1904.07736*.

[39] V. I. Ulyantsev and F. N. Tsarev, ''Extended finite-state machine induction using SAT-solver,'' *IFAC Proc. Volumes*, vol. 45, no. 6, pp. 236–241, May 2012.

KONSTANTIN CHUKHAREV received the bachelor's degree in control systems and informatics and the master's degree in applied mathematics and informatics from ITMO University, Saint Petersburg, Russia, in 2018 and 2020, respectively, and the one-year program ''algorithmic bioinformatics'' from the Bioinformatics Institute, Saint Petersburg, in 2017. He is currently pursuing the Ph.D. degree with the Computer Technologies Laboratory, ITMO University.

He is also a Junior Research Associate with the Computer Technologies Laboratory, ITMO University. He studies formal methods, software engineering, SAT solvers, and phylogenetics, while teaching students discrete math and working on his Ph.D.

DANIIL CHIVILIKHIN received the bachelor's and master's degrees in applied mathematics and informatics and the Ph.D. degree in technical sciences (mathematics and software for computing systems) from ITMO University, Saint Petersburg, Russia, in 2011, 2013, and 2015, respectively.

He is currently an Associate Professor with the Computer Technologies Laboratory, ITMO University. His research interests include program synthesis and verification, industrial informatics, evolutionary algorithms, and SAT solver applications.