Chukharev, Konstantin; Suvorov, Dmitrii; Chivilikhin, Daniil; Vyatkin, Valeriy

SAT-Based Counterexample-Guided Inductive Synthesis of Distributed Controllers

Published in:
IEEE Access

DOI:
10.1109/ACCESS.2020.3037780

Published: 01/01/2020

Document Version
Publisher's PDF, also known as Version of record

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 30, 2020, accepted October 29, 2020, date of publication November 16, 2020,
date of current version November 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3037780

SAT-Based Counterexample-Guided Inductive Synthesis of Distributed Controllers

KONSTANTIN CHUKHAREV[1,2], DMITRII SUVOROV[1,2], DANIIL CHIVILIKHIN[1,2], AND VALERIY VYATKIN[2,3,4], (Senior Member, IEEE)

[1]Sirius University of Science and Technology, 354340 Sochi, Russia
[2]Computer Technologies Laboratory, ITMO University, 197101 Saint Petersburg, Russia
[3]Department of Electrical Engineering and Automation, Aalto University, 02150 Espoo, Finland
[4]Department of Computer Science, Electrical and Space Engineering, Luleå University of Technology, 971 87 Luleå, Sweden

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

Funding: The reported study was funded by RFBR, project number 19-37-51066.

ABSTRACT This article proposes a new method for automatic synthesis of distributed discrete-state controllers from given temporal specification and behavior examples. The proposed method develops known synthesis methods to the distributed case, which is a fundamental extension. This method can be applied for automatic generation of correct-by-design distributed control software for industrial automation. The proposed approach is based on reduction to the Boolean satisfiability problem (SAT) and has Counterexample-Guided Inductive Synthesis (CEGIS) at its core. We evaluate the proposed approach using the classical distributed alternating bit protocol.

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

I. INTRODUCTION

The design of robust industrial automation systems is a complex problem. It is common to use deterministic finite-state models in order to reduce the complexity of engineering and benefit from automated software verification techniques [1]–[5]. Thus, controller logic is often designed as a set of interacting automata.

Usually, finite-state models of controllers are designed by hand. This is a tedious and error-prone process, especially when long-term maintenance is required. Often there is no access to the controller logic source code, leading to the problem of migrating to new automation standards, e.g. migration from the IEC 61131-3 [6] standard for programmable logic controllers (PLCs) to the modern IEC 61499 [5] for distributed industrial automation systems. Alternatively, one could employ finite-state automata synthesis techniques and infer controller models from behavioral examples and/or temporal specification [7]–[14].

Automatic synthesis of finite-state models is a well-known problem [3], [4], [7], [8], [11], [15]–[17]. Over the past years, various methods have been proposed that can be directly applied for controller model inference. However, these methods are mostly focused on inference of a monolithic controller, i.e. controller that is implemented as one single finite-state machine. In this work, we address the problem of synthesizing finite-state models for a set of independent controllers which communicate with each other via network. More specifically, we target systems implemented following the international standard for distributed automation systems development IEC 61499 [5], which defines control systems as networks of interacting *function blocks (FBs)*, specified by their *interfaces* and implementations (*control algorithms*). Essentially, control algorithms are finite-state automata.

The main contribution of this article is a method for synthesis of finite-state models for a *distributed* controller (set of interconnected individual controllers) from behavior examples and temporal specification. We implement the proposed method as an extension of the tool `fbSAT` [18], [19] and evaluate it using the Alternating Bit Protocol (ABP) [20] as an example.

The rest of this article is structured as follows. In Section II, we introduce the necessary definitions and notation. In Section III, we formally state the problem addressed in this article. Section IV contains a survey of related work. In Section V, we briefly describe the previous work done

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

on the synthesis of monolithic controllers [18]. Then, in Section VI, we describe the main contribution of this article: extension of monolithic controller synthesis for distributed systems. Section VII describes our experiments with ABP. The results are discussed in Section VIII, and Section IX concludes the paper.

II. PRELIMINARIES

A. FUNCTION BLOCKS AND AUTOMATA

In this work we consider *function blocks (FBs)*, defined in the IEC 61499 standard, as a target implementation for synthesized systems. A function block is defined by an *interface* and a *control algorithm*. An interface (see e.g. Fig. 1) specifies interaction between function block and environment, and consists of the sets of input and output events (denoted with $\mathcal{I}$ and $\mathcal{O}$ respectively), and the sets of input and output variables (denoted with $\mathcal{X}$ and $\mathcal{Z}$ respectively). We only consider Boolean input/output variables.

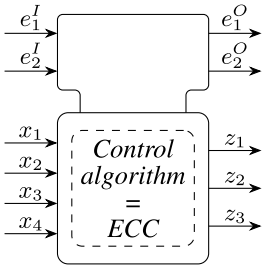


FIGURE 1. A function block.

A control algorithm is a Moore-type finite-state machine, called *execution control chart* (ECC). Later we will refer to such ECC simply as *automaton*. A rigorous description of FB semantics including a formal definition of ECC can be found in [21]. In this article we only consider *canonical* ECCs with *logical* inputs and outputs. Formally, an *automaton* $\mathcal{A}$ is a tuple $(Q, q_{\text{init}}, \mathcal{I}, \mathcal{O}, \mathcal{X}, \mathcal{Z}, \lambda, \psi, \omega)$, where:

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

The states of an automaton are marked with output events and *algorithms*. Algorithm is a function that changes the values of output variables $\mathcal{Z}$. In this article we consider algorithms of form $\mathbb{B}^{|\mathcal{Z}|} \rightarrow \mathbb{B}^{|\mathcal{Z}|}$, where each output variable only depends on its previous value. The transitions of an automaton are marked with input events and *guard conditions* – functions of a form $\mathbb{B}^{|\mathcal{X}|} \rightarrow \mathbb{B}$ which enables/disables a given transition for current input variables values. We label a transition with $\mathtt{R}$ & x to indicate it is marked with an input event $\mathtt{R}$ and a guard x.

An automaton reads a sequence of *input actions* and transforms it into a sequence of *output actions*. An input action $i[\bar{x}]$ is a pair of an input event $i \in \mathcal{I}$ and a vector of input variable values $\bar{x} = \langle x_1, \ldots, x_{|\mathcal{X}|} \rangle$, where $x_i \in \mathcal{X}$. Similarly, an output action $o[\bar{x}]$ is pair of an output event $o \in \mathcal{O} \cup \{\varepsilon\}$ and a vector of output variable values $\bar{z} = \langle z_1, \ldots, z_{|\mathcal{Z}|} \rangle$, where $z_i \in \mathcal{Z}$. An empty output event ε is produced when automaton ignores an input action and does not make a transition. Later we will refer to $\bar{x}$ and $\bar{z}$ simply as *input* and *output*. The semantics of an execution step is as follows. An automaton receives an input action $i[\bar{x}]$ and changes its state to $q' = \lambda(q, i, \bar{x})$, where q is the current state and $\bar{x}$ are current input variable values. Next, it produces an output event $o' = \psi(q, i, \bar{x})$ changes the output variable values to $\bar{z}' = \omega(q, i, \bar{x}, \bar{z})$, where $\bar{z}$ are current output variable values. Note that the automaton may ignore some input actions, in that case the output event is an empty event: $o' = \varepsilon$, and the state and the values of output variables do not change: $q' = q, \bar{z}' = \bar{z}$.

B. EXECUTION SCENARIOS

An *execution scenario* is a sequence of *scenario elements* s_i. A scenario element is a pair of an input action and an output action: $s_i = \langle i[\bar{x}], o[\bar{z}] \rangle$. An automaton $\mathcal{A}$ *satisfies* a scenario s, if after processing a sequence of input actions in s it produces the exact same sequence of output actions as in s.

A *positive scenario* is an execution scenario, that represents a desired behavior of the automaton. Positive scenarios can either be obtained by running a simulation of some model or by accessing a real automation system. For example, in [22], execution scenarios are obtained directly from the legacy automation system: each legacy PLC is physically connected with a data collection PLC that observes and logs all input/output signals of the legacy PLC, thus producing a positive execution scenario. A separate problem addressed in [22] is denoising the hardware-derived execution scenarios that may exhibit various types of errors. In this work we assume that execution scenarios do not contain errors.

C. FORMAL VERIFICATION WITH MODEL CHECKING

In this work we specify a system with a set of linear temporal logic (LTL) [23] formulas. An LTL formula is defined over execution paths and describes some temporal properties of these paths. LTL formulas include atomic propositions (some elementary statements about the system), propositional logic connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$), and temporal operators (*e.g.*, $\mathbf{X}$ – "next", $\mathbf{U}$ – "until", $\mathbf{G}$ – "always", $\mathbf{F}$ – "eventually"). With LTL formulas one can specify safety ("something bad never happens") and liveness ("something good will eventually happen") properties of a given system. An example of a safety property is a formula $\mathbf{G}\,\neg P$, which states that some predicate P is always false. An example of a liveness property is a formula $\mathbf{G}(P \rightarrow \mathbf{F}\,Q)$, which states that if a predicate P is true, then a predicate Q will eventually become true.

Model checking [24] is a technique that can be used to verify a given finite-state model *w.r.t.* a given specification

(here, set of LTL formulas) and obtain a counterexample execution path if this specification is violated. Counterexamples can further be translated to *negative scenarios* – execution scenarios representing undesired behavior of an automaton or a system of automata. This process is described in detail in Section V-A2.

D. BOOLEAN SATISFIABILITY PROBLEM

The SAT problem consists in determining whether a Boolean formula in Conjunctive Normal Form (CNF) is satisfiable. A Boolean variable has values from the set $\mathbb{B} = \{0, 1\}$. Boolean variable x and its negation $\neg x$ are called *literals*; literals x and $\neg x$ are called contrary. A *clause* is a disjunction of non-contrary literals, *e.g.*, $x_1 \vee x_2 \vee \neg x_3$. A Boolean formula in CNF is a conjunction of clauses. A formula is called *satisfiable* if there exists an assignment of variable values such that all clauses evaluate to 1; such an assignment is called *satisfying*. If a satisfying assignment does not exist, the formula is called *unsatisfiable*. The problem is to determine if a given formula is satisfiable, and if it is, to find a satisfying assignment. SAT is the classical NP-complete problem [25], [26]. Extensive research in SAT theory and algorithms resulted in modern SAT solvers based on the conflict-driven clause learning (CDCL) algorithm [27], which are now considered to be a standard computational tool used in various application domains [28].

Instead of developing an ad-hoc algorithm for a hard problem at hand, in many cases one may develop a reduction of the problem to SAT: a function that, given a problem instance, produces a CNF formula that is satisfiable if and only if the original problem has a solution. The resulting CNF formula is then fed as input to a SAT solver tool. The outcome of SAT solving process is either a satisfying assignment, or UNSAT – an indication that the formula is unsatisfiable.

III. DISTRIBUTED CONTROLLER SYNTHESIS PROBLEM

In this work we consider a setup of a distributed controller, in which multiple individual controllers, also called *modules*, communicate with each other via some *environment*. We assume that the controllers are implemented (or could be implemented) with FBs, hence the communication is done by exchanging messages and setting Boolean variables. We do not formalize what an environment is, but instead require that it can be modeled with an input language of some model checker (in our experiments we use NuSMV [29]). By avoiding a more rigorous definition of an environment we gain more flexibility in the application of the proposed approach. For instance, we can model a synchronous setup, when controllers are attached to a common bus, like in [22], or we can model a true distributed system, in which modules communicate via unreliable channels.

We assume that we can observe the system and gather behavior examples, and that there is some temporal specification of the system. Our goal is to infer a set of FB models, one for each module of the distributed controller, such that the distributed system complies with the given positive scenarios and LTL/specification. Denote the number of modules with M, and let $m \in [1 \,..\, M]$ be the index of a module in the rest of this article. Ultimately, the problem addressed in this work is to infer a set of automata $\{\mathcal{A}^{(m)}\}$ for all modules, that comply with a given set of individual positive scenarios $\mathcal{S}^{+(m)}$ for each module, and an LTL/specification $\mathcal{L}$ for the whole system. The module interfaces (*i.e.* sets $\mathcal{X}^{(m)}$, $\mathcal{Z}^{(m)}$, $\mathcal{I}^{(m)}$, and $\mathcal{O}^{(m)}$) are known beforehand, and LTL formulas use variables from these sets as atomic propositions.

IV. RELATED WORK

Our work is mainly inspired by [16], where an approach for automatic synthesis of distributed *protocols* has been proposed. Distributed protocols are similar to distributed industrial control systems in the sense that they too can be modeled with a set of communicating finite-state machines. Correctness of protocols is also specified with *safety* and *liveness* temporal properties. The approach proposed in [16] uses the idea of *program sketching* [30], in which the developer of the protocol specifies an incomplete implementation, and the synthesis tool automatically *completes* the manual implementation by means of search and verification of temporal properties. This approach of protocol *completion* deals with the known undecidability of the distributed synthesis problem [31]: limiting the sizes of the synthesized finite-state machines and their number of transitions makes the completion problem decidable [16]. The approach of [16], similar to our work, is based on counterexample-guided inductive synthesis (CEGIS) [32]. However, it uses the following strong assumptions.

1) An incomplete protocol process (automaton or module) is defined as an automaton in which some transitions are missing (in comparison with a complete process). Thus, it is assumed that the protocol developer provides a partial implementation, in which some needed transitions are missing, and, more importantly, all present transitions are correct. In this assumption, the synthesis (or rather, completion) algorithm only needs to add some missing transitions. This scenario does not appear to be very practical: an error in the manually prepared transitions will prevent the algorithm from finding the solution.
2) Following from the first assumption, the input scenarios in [16] are assumed to cover all necessary states of each protocol. If this assumption is violated, the algorithm will fail to complete the distributed protocol.
3) Due to previous assumptions, the learner part of the CEGIS in [16], the one that is supposed to provide candidate completions for verification, is a simple enumeration algorithm. Though the use of enumeration algorithms is quite a valid approach when the search space is small, in practice it quickly becomes infeasible. A better approach employed in our research is delegation of enumeration to a SAT solver, which performs it in a more intelligent and efficient way.

4) Finally, the algorithm used in [16] to construct incomplete automata from input scenarios heavily depends on so-called labels that annotate messages [33]. Labels affect the way that the scenario elements are merged, thus having an effect on the produced automata. In fact, labels mostly define the mapping of scenario elements to states of the incomplete process.

In contrast to the above, in our approach we do not make any assumptions about the input scenarios (apart that they must define a deterministic system), do not require scenarios to cover all states of the distributed controller, use an intelligent SAT encoding to construct state machines from input scenarios and provide candidate distributed controllers for verification.

Use of SAT solvers is a common approach in automatic passive synthesis of automata [7]–[9], [12], [17], [34], as well as in many other application domains [28]. A related paper that is based on the use SAT solvers is [22], where a *modular* controller is synthesized from one set of positive scenarios, not considering temporal properties. This is in contrast with the problem addressed in the present paper, where each part of the distributed controller (each automaton) is synthesized from a separate set of positive scenarios, and the functioning of the distributed system as a whole is bounded by temporal properties, producing a set of negative scenarios for the entire system.

V. OVERVIEW OF MONOLITHIC CONTROLLER SYNTHESIS WITH fbSAT

In this section we briefly describe the `fbSAT` approach for the synthesis of a single monolithic finite-state function block model from a set of positive scenarios and an LTL/specification, proposed in [18]. The method is based on a reduction to SAT. First, `fbSAT` builds an automaton that satisfies the positive scenarios, that are represented with a (positive) scenario tree. This is done with a SAT encoding that ensures correspondence of scenario elements with corresponding states of the automaton.

Second, the solution constructed from positive scenarios is refined by means of counterexample-guided inductive synthesis: on each iteration a model checker is used to verify the compliance of the current automaton with the LTL/ specification; if counterexamples are generated, then SAT formula is appended with additional clauses that encode that the sought solution must not demonstrate the behavior represented by the counterexample.

A. SCENARIO TREE

1) POSITIVE SCENARIO TREE

A *scenario tree* $\mathcal{T}$ is a prefix tree constructed from the set of execution scenarios. Each execution scenario is prepended with an empty element $\varepsilon[\langle 0 \ldots 0 \rangle]$ so that they share a common prefix. Each node of a scenario tree is marked with an output action, and each incoming edge is marked with an input action. A node with its incoming edge correspond to a scenario element. A scenario tree constructed from a set of positive scenarios is called a *positive scenario tree* and denoted with $\mathcal{T}^+$.

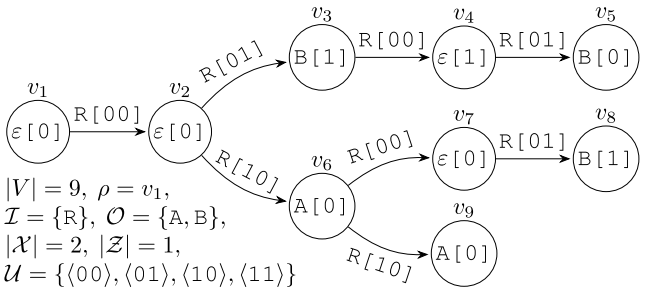


FIGURE 2. Positive scenario tree.

The following notation will be used throughout the paper. The set of all unique inputs in the scenario tree is denoted with $\mathcal{U} \subseteq \mathbb{B}^{|\mathcal{X}|}$. The set of scenario tree nodes is denoted with V, $\rho \in V$ is the root of the tree. The following functions are used to describe the scenario tree:

- $\mathrm{tp}(v) \in V$ is the parent of node $v \in V$, $v \neq \rho$;
- $\mathrm{tie}(v) \in \mathcal{I}$ is the input event on the incoming edge of node $v \in V$, $v \neq \rho$;
- $\mathrm{toe}(v) \in \mathcal{O} \cup \{\varepsilon\}$ is the output event in node $v \in V$;
- $\mathrm{tin}(v) \in \mathcal{U}$ is the input on the incoming edge of node $v \in V$, $v \neq \rho$;
- $\mathrm{tov}(v, z) \in \mathbb{B}$ is the value of the output variable $z \in \mathcal{Z}$ in node $v \in V$.

Tree nodes, other than the root, are divided into *active* $V_{(\mathrm{act})}$ and *passive* $V_{(\mathrm{pass})}$ nodes, those with a non-empty and empty output event respectively:

- $V_{(\mathrm{act})} = \{v \in V \setminus \{\rho\} \mathrm{toe}(v) \neq \varepsilon\}$;
- $V_{(\mathrm{pass})} = \{v \in V \setminus \{\rho\} \mathrm{toe}(v) = \varepsilon\}$.

The scenario tree for the following positive scenarios is shown in Fig. 2:

$[\langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[01]}, \mathtt{B[1]}\rangle; \langle \mathtt{R[00]}, \varepsilon\mathtt{[1]}\rangle;$
$\langle \mathtt{R[01]}, \mathtt{B[0]}\rangle],$
$[\langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[10]}, \mathtt{A[0]}\rangle; \langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle;$
$\langle \mathtt{R[01]}, \mathtt{B[1]}\rangle],$
$[\langle \mathtt{R[00]}, \varepsilon\mathtt{[0]}\rangle; \langle \mathtt{R[10]}, \mathtt{A[0]}\rangle; \langle \mathtt{R[10]}, \mathtt{A[0]}\rangle].$

2) NEGATIVE SCENARIO TREE

An automaton can be verified against a given LTL/ specification with a model checker tool, such as NuSMV [29]. A model checker finds a *counterexample* for each violated LTL property in the form of an execution state sequence. Note that execution states of NuSMV models should not be confused with states of automata.

Counterexamples are translated into *negative scenarios*. A counterexample for a safety property is a finite sequence of execution states. Consider the system depicted in Fig. 3. Every state produces an output event `C` and it is omitted in the picture. A safety LTL property $\mathbf{G}\,\neg z$ is violated as shown by

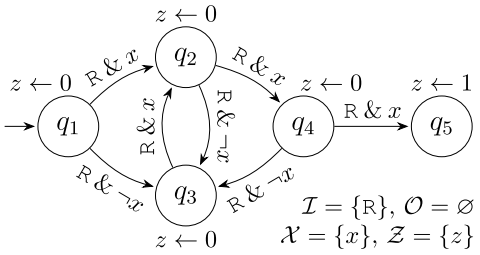


FIGURE 3. Example automaton with a looping behavior.

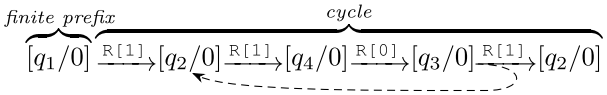
the following counterexample (the notation $[q_1/0]$ indicates the execution state, in which the automaton is in the state q_1, and the current value of z is 0):

$$[q_1/0] \xrightarrow{\mathtt{R[1]}} [q_2/0] \xrightarrow{\mathtt{R[1]}} [q_4/0] \xrightarrow{\mathtt{R[1]}} [q_5/1].$$

This counterexample is translated into the following loopless negative scenario:

$$[\langle \mathtt{R[1],C[0]} \rangle; \langle \mathtt{R[1],C[0]} \rangle; \langle \mathtt{R[1],C[1]} \rangle].$$

A counterexample for a liveness property is an infinite sequence, which can be represented as a finite prefix followed by a loop. Consider a liveness property $\mathbf{F}\, z$ for the same system. A counterexample for it would be the following:



A negative execution scenario obtained from this counterexample contains a loopback and looks like this (the beginning of the loop is underlined):

$$[\underline{\langle \mathtt{R[1],A} \rangle}; \langle \mathtt{R[1],C[0]} \rangle; \langle \mathtt{R[0],C[0]} \rangle; \langle \mathtt{R[1],C[0]} \rangle].$$

A *negative scenario tree* $\mathcal{T}^-$ is a prefix tree constructed from a set of negative scenarios. It contains auxiliary edges that represent loopbacks in negative scenarios. A set of tree nodes connected via a loopback edge with a node $\widehat{v}$ is denoted with $\widehat{\mathrm{tbe}}(\widehat{v}) \subseteq \widehat{V}$. The notation for negative trees is the same as for positive trees, but all symbols are marked with a hat: $\widehat{v} \in \widehat{V}$, $\widehat{\mathrm{tp}}(v)$, $\widehat{V}^{(\mathrm{act})}$, *etc.*

B. FB MODEL INFERENCE USING SAT SOLVER

The described approach is based on a reduction to SAT: a Boolean formula is constructed which is satisfiable if and only if there exists an automaton $\mathcal{A}$ of a predefined size that satisfies positive scenarios and does not satisfy negative scenarios. For non-Boolean variables with finite domains the standard pairwise [35] encoding is used. Tseytin transformations [36] are employed to convert all constraints to CNF. The encoding consists of four parts which are covered in subsequent sections.

The following notation is used hereinafter. The number $C = |Q|$ is the number of states in automaton $\mathcal{A}$, each state has at most $K \leq C$ outgoing transitions. It is also assumed that $b \in \mathbb{B} = \{0, 1\}$, $q \in Q$, $k \in [1\,..\,K]$, $i \in \mathcal{I}$, $o \in \mathcal{O}$, $u \in \mathcal{U}$, $v \in V$, unless stated otherwise.

1) AUTOMATON STRUCTURE ENCODING

The first part of the encoding defines automaton states and transitions. Variable $\phi_q \in \mathcal{O} \cup \{\varepsilon\}$ denotes the output event associated with state q. Variable $\gamma_{q,z,b} \in \mathbb{B}$ denotes the new value for the output variable z in dependence from its old value b and state q. Thus, variable γ encodes the algorithm associated with state q.

A transition is characterized by its destination state, input event, and guard condition. Variable $\tau_{q,k} \in Q_0$, where $Q_0 = Q \cup \{q_0\}$, denotes the destination state of the k-th transition from state q. Some states may have less than K transitions – to deal with it, an auxiliary state q_0 is introduced: $\tau_{q,k} = q_0$ indicates the absence of the k-th transition from the state q. Variable $\xi_{q,k} \in \mathcal{I} \cup \{\varepsilon\}$ denotes the input event associated with the k-th transition from the state q. To encode guard conditions the following variables are introduced. $\theta_{q,k,u} \in \mathbb{B}$ denotes the value of the guard condition associated with the k-th transition from the state q for the input u, and $\delta_{q,k,i,u} \in \mathbb{B}$ is true if and only if that guard condition fires for an input action $i[u]$.

Finally, to represent how the automaton changes states when processing the positive scenarios, variable $\lambda_{q,i,u} \in Q_0$ is introduced. It denotes the state to which the automaton switches after processing the input action $i[u]$ in the state q. $\lambda_{q,i,u} = q_0$ denotes the situation when the automaton ignores the input action $i[u]$ and remains in the state q.

To reduce the search space size, symmetry-breaking constraints [10], [37] are declared. These constraints enforce that the states of the automaton are enumerated in the breadth-first search (BFS) traversal order. Variable $\tau^{\mathrm{bfs}}_{q_i,q_j} \in \mathbb{B}$ $(q_i, q_j \in Q)$ encodes whether there is a transition from state q_i to state q_j. Variables $\tau^{\mathrm{bfs}}_{q_i,q_j} \in \mathbb{B}$ are defined through τ in the following way:

$$\tau^{\mathrm{bfs}}_{q_i,q_j} \leftrightarrow \bigvee_{k \in [1..K]} (\tau_{q_i,k} = q_j).$$

Variable $\pi^{\mathrm{bfs}}_{q_j} \in \{q_1, \ldots, q_{j-1}\}$ $(j \in [2\,..\,C])$ denotes the parent of the state q_j in the BFS traverse tree:

$$(\pi^{\mathrm{bfs}}_{q_j} = q_i) \leftrightarrow \tau^{\mathrm{bfs}}_{q_i,q_j} \wedge \bigwedge_{r<i} (\neg \tau^{\mathrm{bfs}}_{q_r,q_j}).$$

Finally, the symmetry-breaking constraint requires that each state has a parent in the BFS traverse tree with a smaller number:

$$(\pi^{\mathrm{bfs}}_{q_j} = q_i) \rightarrow \bigwedge_{r<i} (\pi^{\mathrm{bfs}}_{q_{j+1}} \neq q_r).$$

2) GUARD CONDITIONS STRUCTURE ENCODING

In the approach described so far, guard conditions are represented as truth tables (via variable θ). This representation is inconvenient and is not human-readable. In [18], guard conditions are represented with parse trees of corresponding Boolean formulas and explicitly encoded in SAT. In this encoding, the parameter P is the guard condition parse tree size, and the parameter N is the upper bound for a total

number of parse tree nodes in all guard conditions. We do not describe this encoding here, since it is not necessary for the understanding of the rest of the paper.

3) POSITIVE SCENARIO TREE MAPPING ENCODING

In order to make the automaton $\mathcal{A}$ comply with a set of positive scenarios represented as scenario tree $\mathcal{T}^+$, a mapping $\mu : V \to Q$ is organized between the tree nodes and the automaton states. The variable $\mu_v \in Q$ denotes the *satisfying state* (also called "mapped") in which the automaton finishes processing a sequence of input actions on the path from ρ to $v \in V$.

The root of the tree maps to the initial automaton state: $\mu_\rho = q_{\text{init}}$. A passive node in the scenario tree represents the case when the automaton ignores an input action and does not change state. Thus, passive nodes map to the same state as their parents:

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

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

Active nodes, on the contrary, represent the situation when the automaton reacts to an input action $i[u]$ and follows a transition – exactly to the *satisfying state*, where the automaton must produce exactly the same output action as specified in the active node:

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

where $v \in V_{(\text{act})}$, $p = \text{tp}(v)$, $q = \mu_p$, $q' \in Q$, $i = \text{tie}(v)$, $u = \text{tin}(v)$, $o = \text{toe}(v)$, $z \in \mathcal{Z}$, $b = \text{tov}(p, z)$, $b' = \text{tov}(v, z)$.

4) NEGATIVE SCENARIO TREE MAPPING ENCODING

Similar to the positive tree mapping, a mapping $\hat{\mu} : \widehat{V} \to Q_0$ is organized between the negative tree nodes and the automaton states. Variable $\hat{\mu}_{\widehat{v}} \in Q_0$ denotes the *satisfying state* (or its absence) in which the automaton finishes processing a sequence of input actions on the path from $\widehat{\rho}$ to $\neg \in \widehat{V}$. Note that it is acceptable for the automaton not to behave in the way specified in the negative scenario tree, *e.g.*, it might produce mismatching output actions – in such cases the corresponding nodes will be *unsatisfied* (also called *unmapped*), which is denoted by $\hat{\mu}_{\widehat{v}} = q_0$, where q_0 is an auxiliary automaton state.

The root $\widehat{\rho}$ of the negative scenario tree $\mathcal{T}^-$ maps to the initial state of the automaton: $\hat{\mu}_{\widehat{\rho}} = q_{\text{init}}$.

Passive nodes either (1) map to the same state as their parent, or (2) become unmapped (this happens when the automaton does not exhibit the passive behavior):

$$(\hat{\mu}_{\widehat{v}} = \hat{\mu}_{\widehat{p}}) \vee (\hat{\mu}_{\widehat{v}} = q_0),$$

where $\widehat{v} \in \widehat{V}^{(\text{pass})}$, $\widehat{p} = \widehat{\text{tp}}(\widehat{v})$. In the first case, the automaton ignores the input action and remains in the same state (*i.e.* does not follow any transition):

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

where $\widehat{v} \in \widehat{V}^{(\text{pass})}$, $q' \in Q$, $i = \widehat{\text{tie}}(\widehat{v})$, $u = \widehat{\text{tin}}(\widehat{v})$. And in the second case, the automaton does the inverse – follows some transition:

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

where $\widehat{v} \in \widehat{V}^{(\text{pass})}$, $\widehat{p} = \widehat{\text{tp}}(\widehat{v})$, $q = \hat{\mu}_{\widehat{p}}$, $i = \widehat{\text{tie}}(\widehat{v})$, $u = \widehat{\text{tin}}(\widehat{v})$.

Active nodes map to the state in which the automaton finishes processing the input action $i[u]$:

$$(\hat{\mu}_{\widehat{v}} = q') \leftrightarrow (\hat{\lambda}_{q,i,u} = q') \wedge (\hat{\phi}_{q'} = o) \wedge \bigwedge_{z \in \mathcal{Z}} (\hat{\gamma}_{q',z,b} = b'),$$

where $\widehat{v} \in \widehat{V}^{(\text{act})}$, $\widehat{p} = \widehat{\text{tp}}(\widehat{v})$, $q = \mu_{\widehat{p}}$, $q' \in Q$, $i = \widehat{\text{tie}}(\widehat{v})$, $u = \widehat{\text{tin}}(\widehat{v})$, $o = \widehat{\text{toe}}(\widehat{v})$, $z \in \mathcal{Z}$, $b = \widehat{\text{tov}}(\widehat{p}, z)$, $b' = \widehat{\text{tov}}(\widehat{v}, z)$. Note that this constraint, compared to positive mapping, uses equivalence ($\leftrightarrow$) instead of implication ($\to$): a node maps to some state *if and only if* the automaton has matching behavior. This allows to declare the definitions of $\hat{\mu}_{\widehat{v}} = q'$ only for $q' \in Q$, without explicitly considering the case $\hat{\mu}_{\widehat{v}} = q_0$ (which would require a large number of long clauses, which generally makes the SAT problem harder for a SAT solver [28]). When the automaton does not exhibit the behavior specified in the scenario tree node, this node remains unmapped (*i.e.* maps to q_0).

If a negative tree node is unmapped, then all its descendants are unmapped as well:

$$(\hat{\mu}_{\widehat{\text{tp}}(\widehat{v})} = q_0) \to (\hat{\mu}_{\widehat{v}} = q_0).$$

Finally, for undesired behavior prohibition, it is required that the first and the last nodes of the loop in the negative tree either map to different states, or both are unmapped (*i.e.* map to q_0):

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

C. COUNTEREXAMPLE-GUIDED INDUCTIVE SYNTHESIS

The approach described so far accepts positive and negative scenarios as input, without specifying the mechanism of producing the negative scenarios. This is done via counterexample-guided inductive synthesis [32]. A CEGIS iteration consists of three steps. First, some automaton $\mathcal{A}$ is inferred based on current positive and negative scenario trees. Second, the inferred automaton is checked against the specification $\mathcal{L}$ with a model checker, and, possibly, some counterexamples describing incorrect behavior are obtained. Finally, these counterexamples are translated into negative scenarios, which are added to the negative tree, and the procedure repeats until there are no more counterexamples, or there is no such automaton that satisfies positive scenarios and does not satisfy negative scenarios. In the first case the inferred automaton complies with the given temporal specification. A CEGIS loop is schematically shown in Fig. 4.

VI. FROM MONOLITHIC TO DISTRIBUTED SYNTHESIS

This article proposes an extension of the `fbSAT` approach [18] described in the previous section. The proposed extension allows for synthesis of distributed

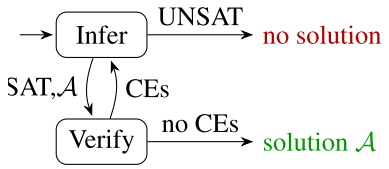


FIGURE 4. CEGIS loop.

controllers, and is based on two ideas. The first one is to declare a reduction for monolithic case for each module independently. This is described in Section VI-A. The second idea has to do with the adaptation of CEGIS developed in [18] to make it applicable for distributed synthesis, and is the main idea of the paper. The difficulty of using CEGIS for distributed synthesis is that since the specification is written for the system as a whole, negative scenarios derived from counterexamples are also written for the system as a whole. Section VI-B describes the structure of these scenarios. The difficulty is that it is not known in advance, which modules behave incorrectly in some particular negative scenario, and thus we do not know, in which module do we need to prohibit the negative scenario.

The naïve approach would be to prohibit each negative scenario for all modules, but it would be incorrect to do so: a behavior that is incorrect for one of the modules may be completely normal for another one. Thus, this naïve approach would lead to unsatisfiability of corresponding Boolean formulas, and the sought distributed controller would not be synthesized.

The key idea to cope with this difficulty is to make sure that each negative scenario is prohibited for at least one module. This is described in Section VI-C. Finally, in Section VI-D we discuss the algorithm for minimization of inferred automata in terms of their parameters, *e.g.*, number of states and total size of guard conditions.

A. REDUCTION FOR DISTRIBUTED SYNTHESIS FROM POSITIVE SCENARIOS

We declare the reduction described in Section V for each module independently, adding the module index to every variable. Thus, respective constraints remain the same, but are now quantified over all modules. For instance, variable $\tau^{(m)}_{q,k} \in Q_0$ denotes the destination state of the k-th transition from state q in the automaton for the m-th module, and variable $\xi^{(m)}_{q,k} \in \mathcal{I} \cup \{\varepsilon\}$ denotes the input event associated with that transition. With the constraint $(\tau^{(m)}_{q,k} = q_0) \leftrightarrow (\xi^{(m)}_{q,k} = \varepsilon)$ we enforce that only transitions to q_0 are marked with the ε input event. We also add module indices to the reduction parameters. For example, $C^{(m)}$ denotes the number of states in $\mathcal{A}^{(m)}$ – the automaton for the m-th module. All other parts of the encoding from Section V are extended in a similar way, except for the negative mapping, which we discuss below.

The difference with positive scenarios is that now we have a set of positive scenario trees $\{\mathcal{T}^+_m\}_{m\in[1..M]}$, one for each module of the distributed controller. Variable $\mu^{(m)}_v$ denotes a state in which the automaton for the m-th module finishes processing a sequence of inputs formed by following the positive tree from the root till the node v (since the tree is deterministic, for each node v there is only one path from the root). This variable represents a mapping $\mu^{(m)} : V^{(m)} \to Q^{(m)}$ between the nodes of $\mathcal{T}^+_m$ and the states of an automaton $\mathcal{A}^{(m)}$, and the constraints on this mapping are the same to those described in Section V-B3.

B. COMPOUND NEGATIVE SCENARIO TREE

A *compound negative scenario* is a sequence of *compound scenario elements* c_i, where c_i is an M-tuple of scenario elements for each synthesized module: $c_i = (s^m_i)$, where $s^m_i = \langle i[\bar{x}], o[\bar{z}]\rangle$.

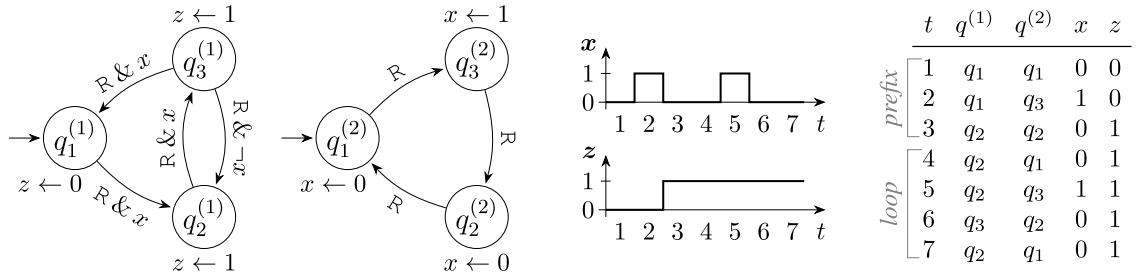
Consider a system of two interconnected modules depicted in Fig. 5. The graph and the table in Fig. 5 show the execution state sequence of this system, illustrating the violation of the LTL property: $\mathbf{GF}\,\neg z$. A compound negative scenario obtained from this execution sequence is shown below (the beginning of the loop is marked with an underline).

$$
\begin{aligned}
&[(\langle \mathtt{R[0]}, \varepsilon\,\mathtt{[0]}\rangle, \langle \mathrm{R}, \mathtt{C[1]}\rangle);\\
&\underline{(\langle \mathtt{R[1]}, \mathtt{C[1]}\rangle, \langle \mathrm{R}, \mathtt{C[0]}\rangle);}\\
&(\langle \mathtt{R[0]}, \varepsilon\,\mathtt{[1]}\rangle, \langle \mathrm{R}, \mathtt{C[0]}\rangle);\\
&(\langle \mathtt{R[0]}, \varepsilon\,\mathtt{[1]}\rangle, \langle \mathrm{R}, \mathtt{C[1]}\rangle);\\
&(\langle \mathtt{R[1]}, \mathtt{C[1]}\rangle, \langle \mathrm{R}, \mathtt{C[0]}\rangle);\\
&(\langle \mathtt{R[0]}, \mathtt{C[1]}\rangle, \langle \mathrm{R}, \mathtt{C[0]}\rangle)]
\end{aligned}
$$

One important difference from the monolithic case is that in distributed synthesis we assume that the input event might be empty: $i \in \mathcal{I} \cup \{\varepsilon\}$. This corresponds to the case when one of the modules does not receive any input at some point in time. We also assume that in valid scenarios $i = \varepsilon$ implies $o = \varepsilon$, *i.e.* if a module does not receive any input event, then it remains idle. Note that in negative traces derived from counterexamples obtained from the model checker, it might be the case that all synthesized modules do not receive an input event. It happens when all modules are idle, awaiting some input events from the environment. We filter out such elements, since they do not carry any information useful for the synthesis.

A *compound negative scenario tree* (or *compound tree*) $\mathcal{CT}^-$ is a negative scenario tree built from a set of compound negative scenarios in a similar way as described in Section V-A. Each node of $\mathcal{CT}^-$ and its incoming edge correspond to a compound scenario element. We denote the set of nodes of tree $\mathcal{CT}^-$ with $\widehat{\mathcal{CV}}$ and reuse the negative tree auxiliary functions ($\widehat{\mathrm{tp}}(\widehat{v})$, $\widehat{\mathrm{tie}}(\widehat{v})$, *etc.*) for the compound tree.

A *projection* of a compound scenario element to module m is simply its m-th element. A *projection* $\mathcal{T}^-_m$ of a compound tree $\mathcal{CT}^-$ to module m is a tree isomorphic to $\mathcal{CT}^-$, each node of which is a projection of the scenario element in the respective node of $\mathcal{CT}^-$ to module m.



	t	$q^{(1)}$	$q^{(2)}$	x	z
prefix	1	q_1	q_1	0	0
	2	q_1	q_3	1	0
	3	q_2	q_2	0	1
loop	4	q_2	q_1	0	1
	5	q_2	q_3	1	1
	6	q_3	q_2	0	1
	7	q_2	q_1	0	1

FIGURE 5. Looping distributed system.

C. COMPOUND NEGATIVE SCENARIO TREE MAPPING

Variable $\widehat{\mu}_{\widehat{v}}^{(m)}$ denotes the satisfying state in automaton $\mathcal{A}^{(m)}$ for node $\widehat{v} \in \widehat{\mathcal{CV}}$, with $\widehat{\mu}_{\widehat{v}}^{(m)} = q_0$ in case when the automaton does not exhibit this particular behavior, and hence there is no satisfying state for $\widehat{v}$ (auxiliary state q_0 is common for all modules). For each module, the root node of the compound negative tree maps to the initial state of the corresponding automaton: $\widehat{\mu}_{\widehat{\rho}}^{(m)} = {q_{\text{init}}}^{(m)}$.

The projection of a compound tree may contain nodes with $i = \varepsilon$ on an incoming edge – we call such nodes *idle*. Among non-idle nodes, we recognize passive (with $\widehat{\text{toe}}(\widehat{v}) = \varepsilon$) and active nodes (with $\widehat{\text{toe}}(\widehat{v}) \neq \varepsilon$) with the same properties as for regular negative trees. We denote the sets of idle, passive and active nodes of the compound negative tree as $\widehat{\mathcal{CV}}_{(\text{idle})}$, $\widehat{\mathcal{CV}}_{(\text{pass})}$, and $\widehat{\mathcal{CV}}_{(\text{act})}$ respectively.

The constraints for passive and active nodes are the same as in the reduction for monolithic synthesis. Each idle node maps to the same state as its parent. The only difference from passive nodes is that we do not need to encode that the automaton ignores an input action, since, in fact, it does not receive any: $(\widehat{\mu}_{\widehat{v}}^{(m)} = \widehat{\mu}_{\widehat{\text{tp}}(\widehat{v})}^{(m)})$, where $\widehat{v} \in \widehat{\mathcal{CV}}_{(\text{idle})}$.

Similar to the monolithic case, we ensure that if the node $\widehat{v} \in \widehat{\mathcal{CV}}$ is unmapped (*i.e.* mapped to q_0), then this property is propagated down the tree:

$$(\widehat{\mu}_{\widehat{\text{tp}}(\widehat{v})}^{(m)} = q_0) \rightarrow (\widehat{\mu}_{\widehat{v}}^{(m)} = q_0).$$

Finally, we need to prohibit the looping behavior described in counterexamples, but in such a way that each loop is prohibited in at least one module. This is done by adding a disjunction over $m \in [1\,..\,M]$ for each loop:

$$\bigwedge_{\widehat{v}' \in \widehat{\text{tbe}}(\widehat{v})} \bigvee_{m \in [1..M]} \left[(\widehat{\mu}_{\widehat{v}}^{(m)} \neq \widehat{\mu}_{\widehat{v}'}^{(m)}) \vee (\widehat{\mu}_{\widehat{v}}^{(m)} = \widehat{\mu}_{\widehat{v}'}^{(m)} = q_0) \right].$$

Thus, on each CEGIS iteration we accumulate clauses in the SAT formula to encode that this particular negative scenario is prohibited for at least one module. Essentially, this approach is an implicit search delegated to a SAT solver.

D. FINDING A MINIMAL DISTRIBUTED CONTROLLER

Ultimately, we want to infer the *minimal* distributed controller complying with the given specification $\mathcal{L}$. Minimal models have more value in practice – they are much more comprehensible for humans and more efficient in the hardware. However, if we start minimizing (*e.g.*, by adding additional constraints) the solution – set of modules $\{\mathcal{A}^{(m)}\}$ – found by a CEGIS, the resulting minimal solution may (and, in practice, will) not comply with the specification $\mathcal{L}$, though the already obtained negative scenarios will still not be satisfied, as expected. Therefore, the minimization step should be included into the CEGIS loop. Moreover, producing minimal models on each CEGIS iteration allows to perform model checking much faster, because smaller models have a smaller state space. Later we will refer to CEGIS with minimization included as CEGIS-min.

In order to enable the minimization of the synthesized automata in terms of total number of states we perform the following extensions of the reduction to SAT. To begin with, observe that parameters $C^{(m)}$ – maximum number of states in each module $m \in [1\,..\,M]$ – have to be known before the beginning of inference process, and the solution (if found) will have exactly the specified size (number of states). The simplest strategy for finding a satisfying solution with a minimal total number of states would be an iterative strategy – linear bottom-up search. However, in case of $M > 1$ (*i.e.* any non-monolithic synthesis) it is unclear how to perform the iteration of parameters – efficiently, whereas maintaining the minimality – and this becomes a problem by itself. One possible way is to iterate all $C^{(m)}$ altogether until we find some solution (all "too small" values result in UNSAT), and then try to decrease some of $C^{(m)}$. However, the latter – inference with different $C^{(m)}$ values – requires *restarts* of the SAT solver, since the declared variables (those with index $q \in Q^{(m)}$) and constraints depend on the $C^{(m)}$ *statically*, *i.e.* this parameter cannot be easily changed while preserving the satisfiability. Hence, we need a *dynamic* way to gradually "disable" some states in the modules. This can be done using the "used states" approach proposed in [22]. The main idea is to mark each state *used*/*unused*, which allows us to minimize the total number of *used* states by encoding it with the *totalizer* [38].

Variable $\zeta_q \in \mathbb{B}$ denotes whether the state q is *used* by an automaton, *i.e.* is reachable from the initial state q_{init}. Clearly, the initial state is always *used*. A (non-initial) state is *used* if and only if it has an incoming transition:

$$\zeta_{q_{\text{init}}} \wedge \bigwedge_{q \in Q \setminus \{q_{\text{init}}\}} \left[\zeta_q \leftrightarrow \bigvee_{\substack{q' \in Q \\ k \in [1..K]}} (\tau_{q',k} = q) \right].$$

Additionally, the following constraint ensures that unused states have the largest indices:

$$\neg\zeta_{q_i} \rightarrow \neg\zeta_{q_{i+1}}.$$

In the context of distributed synthesis, this variable is declared for each module $m \in [1\,..\,M]$: $\zeta_q^{(m)}$. The total number of used states in all modules is denoted C^{used} and encoded using the *totalizer* [38].

Symmetry-breaking constraints declared earlier (Section V-B1 require the states of the automaton to be enumerated in the BFS traversal order, *i.e.* to be included in the BFS traverse tree. This implies that all states (except the initial one) have an incoming transition, due to the definition of π_q^{bfs} variable – parent of the state q in the BFS traverse tree. Consequently, this implies that all states are *used*, by definition. In order to both reduce the search space by using the BFS constraints *and* allow the states to be *unused*, the former have to be modified in the following way. Modified variable $\pi_{q_j}^{\text{bfs-mod}} \in \{q_0, q_1, \ldots, q_{j-1}\}$ $(j \in [2\,..\,C])$ includes q_0 in its domain, which represents the absence of a parent for the state $q_j \in Q$. Only *unused* states do not have a parent:

$$\neg\zeta_{q_j} \leftrightarrow (\pi_{q_j}^{\text{bfs-mod}} = q_0).$$

Variable $\tau_{q_i,q_j}^{\text{bfs}} \in \mathbb{B}$ and all other constraints – definitions of τ^{bfs} and $\pi^{\text{bfs-mod}}$, and the actual BFS constraint – do not require any additional changes.

The main procedure implementing the CEGIS-min is shown in Algorithm 1. First, we estimate parameter D: the *common* upper bound for the number of states in each module. The estimation is done by a simple bottom-up linear search. On each iteration we call the function `infer(***)`, where *** denotes the common arguments $(\{\mathcal{S}^{+(m)}\}, \mathcal{CS}^{-}, \{C^{(m)} = D\}, P)$. The `infer` procedure is responsible for:

1) building positive scenario trees for $\{\mathcal{S}^{+(m)}\}$ and a negative compound scenario tree for $\mathcal{CS}^{-}$;
2) reducing the inference problem to SAT by declaring the constraints described in this article;
3) delegating to a SAT solver to actually solve the SAT problem;
4) building a solution – set of automata (modules) $\{\mathcal{A}^{(m)}\}$ satisfying given scenarios – from the satisfying assignment found by the SAT solver.

Next, we start an actual CEGIS loop. First, we minimize C^{used} using a top-down approach. Here, we call `infer` with an extra predicate/argument "$C^{\text{used}} <$ `UB`", where `UB` denotes the upper bound for a total number of used (reachable) states in all synthesized automata. This predicate represents a *cardinality constraint* which is encoded using a *totalizer* technique [38]. Next, we minimize N^{used} using the same approach. After that, we perform model checking of the synthesized distributed controller represented by automata $\{\mathcal{A}^{(m)}\}$ for compliance with the given LTL/ specification $\mathcal{L}$ using NuSMV [29] model checker, and obtain a set of negative compound scenarios $\mathcal{CS}^{-}_{\text{new}}$ – if there are any, they will be taken into account on the next iteration, else the CEGIS is done and $\{\mathcal{A}^{(m)}\}$ is the satisfying solution.

Algorithm 1: Distributed-CEGIS-min($\{\mathcal{S}^{+(m)}\}$, $\mathcal{L}$, P)

Input: set of positive scenario sets $\{\mathcal{S}^{+(m)}\}_{m\in[1..M]}$, LTL-specification $\mathcal{L}$, parse tree size P.
Output: automata $\{\mathcal{A}^{(m)}\}_{m\in[1..M]}$ complying with $\mathcal{L}$.

```
// Compound negative scenarios
CS⁻ ← ∅
// Common arguments (for "infer")
*** := ({S⁺⁽ᵐ⁾}, CS⁻, {C⁽ᵐ⁾ = D}, P)
label start
// Estimate D
for D ← 1 to ∞ do
    {A⁽ᵐ⁾} ← infer(***)
    if {A⁽ᵐ⁾} ≠ null then break
while true do
    // Minimize C^used
    {A⁽ᵐ⁾} ← infer(***, C^used < ∞)
    if {A⁽ᵐ⁾} = null then
        goto start
    while {A⁽ᵐ⁾} ≠ null do
        C^used_min ← numberOfUsedStates({A⁽ᵐ⁾})
        {A⁽ᵐ⁾} ← infer(***, C^used < C^used_min)
    // Minimize N^used
    {A⁽ᵐ⁾} ← infer(***, C^used = C^used_min, N^used < ∞)
    while {A⁽ᵐ⁾} ≠ null do
        N^used_min ←
          numberOfParseTreeNodes({A⁽ᵐ⁾})
        {A⁽ᵐ⁾} ← infer(***, C^used = C^used_min, N^used <
          N^used_min)
    {A⁽ᵐ⁾} ← infer(***, C^used = C^used_min, N^used =
      N^used_min)
    CS⁻_new ← performModelChecking({A⁽ᵐ⁾}, L)
    if CS⁻_new = ∅ then // No counterexamples
        return {A⁽ᵐ⁾}
    CS⁻ ← CS⁻ ∪ CS⁻_new
```

VII. CASE STUDY: ALTERNATING-BIT PROTOCOL

We evaluate the proposed approach on the example of the classical Alternating-bit protocol (ABP) [20]. It is widely used to illustrate the concepts of formal verification and model checking, *e.g.* [16], [39], [40].

A. ABP DESCRIPTION

The purpose of the ABP is to allow for a communication over an unreliable channel. The system shown in Fig. 6 consists of several modules, which interact with each other via message passing. The *Sender* and the *Receiver* modules

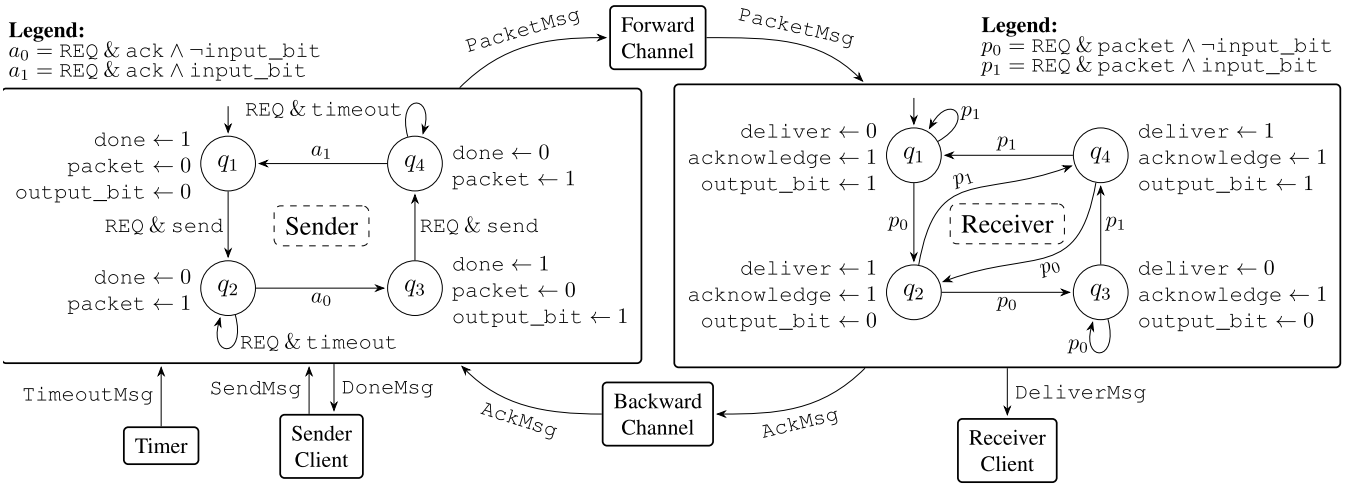


FIGURE 6. Alternating-bit protocol scheme.

implement the protocol, and our goal is to infer finite-state FB models for them. There are also several environment modules, which we model with NuSMV. Environment modules have the following semantics.

- *Forward Channel* and *Backward Channel* transmit messages from *Sender* to *Receiver* and vice versa, respectively. Channels model unreliable transmission: messages can either be lost or duplicated.
- *Timer* sends timeouts to *Sender* in order to force a message retransmission to cope with a lost message.
- *Sender Client* and *Receiver Client* model protocol clients, which communicate with each other via the protocol and expect reliable message transmission.

More specifically, the modules communicate as follows.

- *Sender* receives a `SendMsg` from *Sender Client* whenever the latter wants to send a message, and replies with `DoneMsg` upon a successful message transmission.
- *Sender* receives a `TimeoutMsg` from *Timer* and uses this information to retransmit messages over *Forward Channel*.
- *Sender* sends `PacketMsg` to *Receiver* via a *Forward Channel*. This message is annotated with a single bit of information.
- *Receiver* reports with `DeliverMsg` to *Receiver Client* upon a successful packet deliver.
- *Receiver* sends `AcknowledgeMsg` to *Sender* via a *Backward Channel*. This message is annotated with a single bit of information and acknowledges the delivery of a `PacketMsg` annotated with the same bit.

For technical reasons we represent messages with input/output variables instead of direct representation with events. To store the bit annotating `PacketMsg`, we introduce variables `input_bit` and `output_bit` for both *Sender* and *Receiver*. We also introduce a special input event `REQ` and a special output event `CNF` (widely used in IEC 61499 [5]) with the following semantics. A module receives a `REQ` (request) event from the environment whenever some input variables of this module have changed value. And it produces a `CNF` (confirmation) output event to notify the system that it has reacted on the request and modified values of its output variables accordingly. In fact, every event-oriented system can be transformed into a `REQ`/`CNF`-system, and the reverse transformation is also possible. Thus, *Sender* has the following interface:

- $\mathcal{I}^{\mathrm{s}} = \{\texttt{REQ}\}$;
- $\mathcal{O}^{\mathrm{s}} = \{\texttt{CNF}\}$;
- $\mathcal{X}^{\mathrm{s}} = \{\texttt{send}, \texttt{timeout}, \texttt{acknowledge}, \texttt{input\_bit}\}$;
- $\mathcal{Z}^{\mathrm{s}} = \{\texttt{done}, \texttt{packet}, \texttt{output\_bit}\}$.

The *Receiver* module has the following interface:

- $\mathcal{I}^{\mathrm{r}} = \{\texttt{REQ}\}$;
- $\mathcal{O}^{\mathrm{r}} = \{\texttt{CNF}\}$;
- $\mathcal{X}^{\mathrm{r}} = \{\texttt{packet}, \texttt{input\_bit}\}$;
- $\mathcal{Z}^{\mathrm{r}} = \{\texttt{deliver}, \texttt{acknowledge}, \texttt{output\_bit}\}$.

We transform safety and liveness monitors that are used in [16] to formally specify the ABP system into the following LTL properties.

First, we introduce four safety properties $\mathcal{L}_{safe}$ which ensure the correct order between `send` and `deliver`, and between `deliver` and `done`.

1) $\neg(\neg\texttt{send}\,\mathbf{U}\,\texttt{deliver})$: `deliver` can only happen after `send`.
2) $\mathbf{G}(\texttt{send} \rightarrow \mathbf{X}\neg(\neg\texttt{deliver}\,\mathbf{U}\,\texttt{send}))$: two `send`s cannot occur in a row without a `deliver` in between.
3) $\neg(\neg\texttt{deliver}\,\mathbf{U}\,\texttt{done})$: `done` can only happen after `deliver`.
4) $\mathbf{G}(\texttt{deliver} \rightarrow \mathbf{X}\neg(\neg\texttt{done}\,\mathbf{U}\,\texttt{deliver}))$: two `deliver`'s cannot occur in a row without a `done` in between.

Next, we augment the specification with three liveness properties $\mathcal{L}_{live}$. With P we denote the formula for fairness assumption under which we verify liveness properties: we do not take into account infinite execution paths with no `timeout`, and we ensure that the channels cannot lose

packets infinitely often. The first two properties guarantee the message delivery. The third one ensures that the *Sender Client* is not stale and at any moment in time it will eventually send a packet.

1) $P \rightarrow \mathbf{G}(\texttt{send} \rightarrow \mathbf{F}\,\texttt{deliver})$: any `send` is eventually followed by `deliver`.
2) $P \rightarrow \mathbf{G}(\texttt{send} \rightarrow \mathbf{F}\,\texttt{done})$: any `send` is eventually followed by `done`.
3) $P \rightarrow \mathbf{GF}\,\texttt{send}$: `send` occurs infinitely often.

Manually designed automata for *Sender* and *Receiver* complying with the described LTL/specification are shown in Fig. 6.

B. EXPERIMENTAL EVALUATION

We apply the proposed Distributed-Cegis-min method to infer a minimal distributed controller complying with the given positive scenarios $\{\mathcal{S}^{+(m)}\}$ and the LTL/specification $\mathcal{L}$.

For a model checker we used NuSMV [29]. For a SAT solver we used MiniSAT [41] through the native interface. Experiments were conducted on a computer with an Intel(R) Core(TM) i7-7700HQ CPU @ 3.40GHz and 32GB of RAM.

Since our designed liveness properties contain fairness assumptions, their verification – even for ''small'' models – takes an excessive amount of time (typically, around 20 seconds, but sometimes it spans up to 3 minutes). This should be taken into an account, because the verification step is performed on each iteration of CEGIS. Hence, we split the specification and additionally perform the further described experiments using safety properties only. However, the results of such evaluations are less informative and can only be regarded as proof of CEGIS convergence.

A *sufficient* number of behavior examples allows inferring minimal models complying with the given LTL/specification in just several CEGIS iterations. In particular, 10 scenarios of length 25 or more are sufficient to cover all safety properties $\mathcal{L}_{safe}$ so that the initial inference only from those scenarios produces satisfying automata – in Table 1 this corresponds to ''#iter $= 1$''. The liveness properties $\mathcal{L}_{live}$ are much harder to cover by positive scenarios alone, so the inference process requires multiple CEGIS iterations, but the size and number of scenarios still contribute a lot. The key observation is the following: if the initial automata constructed only from positive scenarios have sizes equal (*e.g.*, 3+4 within this experiment) to the sizes of the sought automata – which means that the given scenarios are sufficient to render the overall structure – then the CEGIS process will converge in several iterations. On the other hand, when the given positive scenarios by themselves are insufficient (extreme case is when there are no positive scenarios at all) to capture the desired behavior, the CEGIS process resembles the enumerative LTL synthesis – in such cases, the proposed method is not very suitable. Example of such case is the evaluation on $\mathcal{S}_{1\times 50}$ – 1 scenario of length 50 – execution was aborted after 6 hours without a satisfying result. This confirms the clause in [16] that the problem of inference *from scratch* is quite challenging.

TABLE 1. Experimental results.

$\{\mathcal{S}^{+(m)}\}$	$\mathcal{L}$	DISTRIBUTED-CEGIS-MIN time, s	#iter	$C^{(m)}$	$N^{(m)}$
$\mathcal{S}_{1\times 50}$	$\mathcal{L}_{safe}$	Aborted after 6 hours			
	$\mathcal{L}_{all}$				
$\mathcal{S}_{1\times 75}$	$\mathcal{L}_{safe}$	63.3	21	$3+4$	$11+7$
	$\mathcal{L}_{all}$	393.2	17	$3+4$	$11+7$
$\mathcal{S}_{1\times 100}$	$\mathcal{L}_{safe}$	37.1	12	$3+4$	$11+7$
	$\mathcal{L}_{all}$	311.5	9	$3+4$	$13+7$
$\mathcal{S}_{10\times 25}$	$\mathcal{L}_{safe}$	3.4	1	$3+4$	$12+6$
	$\mathcal{L}_{all}$	71.2	6	$3+4$	$13+6$
$\mathcal{S}_{10\times 50}$	$\mathcal{L}_{safe}$	3.8	1	$3+4$	$13+7$
	$\mathcal{L}_{all}$	8.1	2	$3+4$	$13+7$

It should be noted that the sizes of scenarios in Table 1 denote the number of execution states in the NuSMV simulation traces, which are generated randomly and may contain plenty of idle steps, resulting in effectively poorly informative scenarios.

To sum up, the results in Table 1 indicate the effectiveness of the proposed Distributed-Cegis-min method on datasets of various sizes, especially if the given positive scenarios are sufficient to capture the desired behavior of the distributed system.

Models for *Sender* (left) and *Receiver* (right) modules of ABP synthesized from $\mathcal{S}_{10\times 50}$ with $\mathcal{L}_{\text{all}}$ using the proposed algorithm are shown in Fig. 7. By manually assessing the synthesized automata, we make sure that they indeed guarantee the packet delivery. Obviously, by construction they comply with the scenarios and the LTL/specification. The automaton for the *Sender* module contains only three states, whereas manually constructed has four. *Sender* and *Receiver* have guard conditions of sizes $N^{(\text{s})} = 13$ and $N^{(\text{r})} = 7$, as opposed to 11 and 28 in the manually prepared automata. The synthesized automata are not easily understandable by human and operate in such a way that might seem unnatural. For example, if *Receiver* receives a `PacketMsg` annotated with 0 in the state 3, it (1) produces an `AcknowledgeMsg` annotated with 1, and relies on the *Sender*, that it will handle this correctly (in fact it must handle this case, because messages may duplicate); (2) changes its state to 1, and awaits for another `PacketMsg` from the *Sender*. Thus, *Receiver* behaves suboptimally. If optimal behavior is desired, the specification could be supplemented with the following LTL properties:

$$\mathbf{G}(p_i \rightarrow \mathbf{X}\,a_i), \text{ where}$$
$$p_0 = \texttt{packet} \wedge \texttt{input\_bit},$$
$$p_1 = \texttt{packet} \wedge \neg\texttt{input\_bit},$$
$$a_0 = \texttt{acknowledge} \wedge \texttt{output\_bit},$$
$$a_1 = \texttt{acknowledge} \wedge \neg\texttt{output\_bit},$$

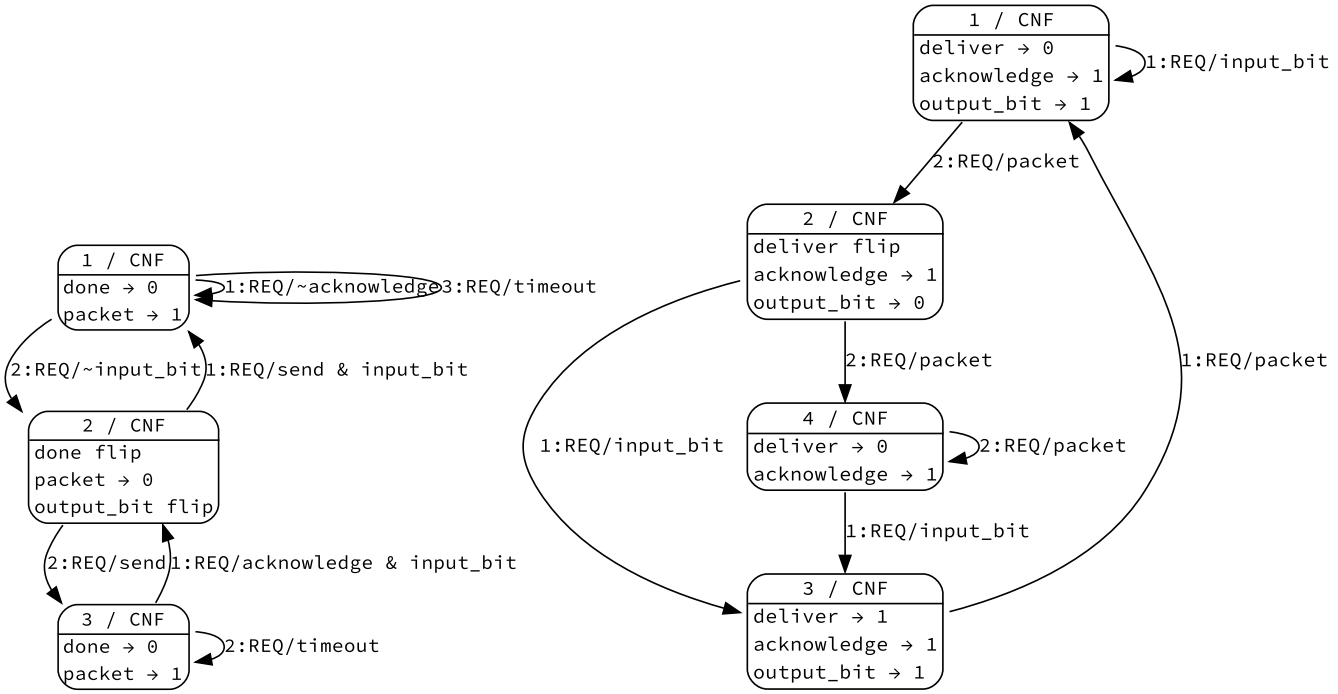


FIGURE 7. Models for *Sender* (left) and *Receiver* (right) modules of ABP synthesized from $\mathcal{S}_{10\times50}$ with $\mathcal{L}_{\text{all}}$.

which ensure that the *Receiver* reacts with immediate `AcknowledgeMsg` to every `PacketMsg`. However, this is not a drawback of the proposed method, and the question of what should be considered a complete specification is beyond the scope of this article.

C. COMPARISON WITH THE PROTOCOL COMPLETION APPROACH

Recall the protocol completion approach proposed by the authors of [16] for distributed synthesis. First, they use execution scenarios to build incomplete processes (automata), which comprise the distributed protocol. Here, they assume that the scenarios are sufficient to cover all process states. Next, they try to complete those incomplete processes by adding some missing transitions, basically using an enumerative CEGIS approach. The main downside of this approach is that the initially derived states and transitions are fixed throughout the CEGIS process, which may (and should) lead to unsatisfiability results in practice.

Ultimately, the goal of experiments reported in [16] is to synthesize ABP *from scratch*, by only specifying the number of states of the *Sender* and *Receiver*, *i.e.* by completing an incomplete protocol that has no defined transitions at all. Striving to reach this goal, authors of [16] performed two interesting sets of experiments.

In the first set of experiments they start with the manually designed ABP, and remove transitions one by one (though, not all), consecutively asking the completion tool to synthesize *all* possible completions.

In the second set of experiments they start with the incomplete protocol built from a sufficient set of execution scenarios, and remove its transitions one by one, each time running the completion tool to synthesize a single correct complete solution. The results reported by the authors indicate that the initial incomplete protocol could be completed in 65 iterations in 19 seconds. Further removal of transitions led to an increase of running time, and in the extreme case, when all 8 transitions from the incomplete protocol were removed, the tool execution was aborted after 4 hours.

We can perform a set of experiments resembling the second experiment set and compare the outcomes. However note that since the target models of our research and of [16] are quite different, the following cannot be considered a one-to-one comparison.

We will consider two sets of execution scenarios: $\mathcal{S}_{10\times50}$ (10 scenarios, each of length 50) will play a role of "sufficient" set of scenarios, and $\mathcal{S}_{1\times50}$ (1 scenario of length 50) will be considered "insufficient set of scenarios". Presumably, the larger scenario set size is, the more automaton state it will cover. Hence, using sufficient set of scenarios will result in a more efficient inferring with CEGIS. Inference of ABP only from "sufficient" scenarios results in the *Sender* with 3 states and the *Receiver* with 4 states. On the contrary, inference only from "insufficient" scenarios results in the *Sender* with only 2 states and the *Receiver* with only 3 states. Note that in both cases the resulting automata do not comply with the LTL/specification $\mathcal{L}_{\text{all}}$, *yet*. As already shown in Table 1, the application of the proposed Distributed-Cegis-min method to the scenarios $\mathcal{S}_{10\times50}$ allows inferring the satisfying ABP complying with $\mathcal{L}_{\text{all}}$ in just 2 iterations in 8 seconds.

The execution of the proposed method on $\mathcal{S}_{1\times50}$ was aborted after several hours without a result. However, since we know the parameters (number of states $C^{(m)}$ and parse trees nodes $N^{(m)}$) of the final ABP, we can specify them to the proposed method, which greatly reduces the search space. By initially specifying $C^{(m)} = 3 + 4$ and $N^{(m)} = 13 + 7$ we obtained ABP satisfying $\mathcal{S}_{1\times50}$ and complying with $\mathcal{L}_{\text{all}}$ in 622 iterations in 27867 seconds. Note that most of this time was spend on model checking, not on SAT solving – verification time ranged from 10 to 200 seconds. A similar run, but only with safety properties $\mathcal{L}_{safe}$, completed in 565 iterations in 1231 seconds.

VIII. DISCUSSION

The experiments described in the previous section show that the synthesis works most effectively when supplied with a sufficient set of positive scenarios (*e.g.* $\mathcal{S}_{10\times50}$), resulting in a single CEGIS iteration for $\mathcal{L}_{safe}$ and a few more for $\mathcal{L}_{\text{all}}$. The solving time increases as we decrease the total size of scenarios, resulting in a timeout for $\mathcal{S}_{1\times50}$.

An extreme case of the problem is to infer automata from LTL/specification only, which makes CEGIS practically infeasible, because it is much harder to describe the automaton with a set of negative scenarios only, than with a set of positive scenarios. In other words, positive scenarios describe what a system should do, while negative scenarios describe what it should not do. Naturally, the state space of undesired behaviors is much larger than that of desired ones.

Observe that in [16] the process completion strictly requires a set of execution scenarios that cover *all* states of protocol processes. The approach for distributed synthesis proposed in the present paper, on the other hand, requires *some* representative behavior examples: ideally, covering all states as in [16], but not necessarily.

Also note that in our experiments we observed that model checking is taking 90-95 % of run time in each CEGIS iteration. This can be partially explained by the fact that we used a symbolic model checker NuSMV in our implementation. Research (*e.g.* [42]) indicates that in many cases explicit-/state model checking (*e.g.*, using SPIN[1]) is much faster than symbolic model checking. Thus, if we switch to explicit-state model checking, the verification time may decrease, leading to better overall performance of our algorithm.

The presented experimental evaluation was conducted on rather small examples, however, we observe a relatively short execution time (especially the synthesis time compared to the verification time), which indicates that the proposed approach would scale well on larger data.

IX. CONCLUSION AND FUTURE WORK

We have developed a complete SAT-based counterexample-guided approach for the synthesis of a distributed finite-state controller from given behavior examples of a legacy system and formal specification in the form of LTL properties. The approach generates a set of finite-state machines implementing the distributed controller, one for each designated element (module) of the distributed control system. Thought the proposed method itself is quite general, our implementation reported in this article is specifically tailored to IEC 61499 function blocks, allowing direct application to synthesis of distributed controllers in an industrial format and their use in real industrial automation systems.

Future research may include speeding up CEGIS by extensively employing the *iterative* SAT solving, *e.g.*, by eliminating the solver restarts after the minimization on each CEGIS iteration. Also, LTL properties during the CEGIS can be taken into account *gradually*, one by one, in the order of their complexity, which might reduce the total execution time – due to not spending time on the model checking on early iterations. Another research direction is the support of *non-canonical* ECCs – those with multiple output events in automata states.

[1] http://spinroot.com/

ACKNOWLEDGMENT

(Konstantin Chukharev and Dmitrii Suvorov contributed equally to this work.)

REFERENCES

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

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

[3] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2008, pp. 248–257.

[4] 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, Jun. 2016.

[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] *Programmable Controllers—Part 1: General Information*, document IEC 61131-1:2003, 2003. [Online]. Available: https://webstore.iec.ch/publication/4550

[7] M. J. H. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Grammatical Inference: Theoretical Results and Applications*. Berlin, Germany: Springer, 2010, pp. 66–79.

[8] 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.

[9] 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.

[10] 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.

[11] G. Giantamidis and S. Tripakis, "Learning Moore machines from input–output traces," in *Formal Methods*. Cham, Switzerland: Springer, 2019, pp. 291–309.

[12] F. Avellaneda and A. Petrenko, "FSM inference from long traces," in *Formal Methods*. Cham, Switzerland: Springer, 2018, pp. 93–109.

[13] 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, 2014, pp. 541–549.

[14] 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.

[15] E. M. Gold, "Complexity of automaton identification from given data," *Inf. Control*, vol. 37, no. 3, pp. 302–320, Jun. 1978.
[16] R. Alur and S. Tripakis, "Automatic synthesis of distributed protocols," *ACM SIGACT News*, vol. 48, no. 1, pp. 55–90, Mar. 2017.
[17] 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.
[18] K. Chukharev and D. Chivilikhin, "FbSAT: Automatic inference of minimal finite-state models of function blocks using SAT solver," 2019, *arXiv:1907.03285*. [Online]. Available: http://arxiv.org/abs/1907.03285
[19] *fbSAT*. Accessed: Nov. 12, 2020. [Online]. Available: http://www.github.com/ctlab/fbSAT
[20] J. Kurose and K. Ross, *Computer Networking: A Top-down Approach*, 5th ed. Reading, MA, USA: Addison-Wesley, 2009, p. 862.
[21] 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.
[22] D. Chivilikhin, S. Patil, K. Chukharev, A. Cordonnier, and V. Vyatkin, "Automatic state machine reconstruction from legacy PLC using data collection and SAT solver," *IEEE Trans. Ind. Informat.*, vol. 16, no. 12, pp. 7821–7831, Dec. 2020.
[23] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Berlin, Germany: Springer-Verlag, 1995, p. 512.
[24] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999, p. 330.
[25] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. 3rd Annu. ACM Symp. Theory Comput. (STOC)*, 1971, pp. 151–158.
[26] L. A. Levin, "Universal sequential search problems," *Problems Inf. Transmiss.*, vol. 9, no. 3, pp. 265–266, 1973.
[27] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," in *Handbook of Satisfiability*, (Frontiers in Artificial Intelligence and Applications), vol. 185. Amsterdam, The Netherlands: IOS Press, 2009, pp. 131–153.
[28] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability* (Frontiers in Artificial Intelligence and Applications). vol. 185. Amsterdam, The Netherlands: IOS Press, 2009, p. 980
[29] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: A new symbolic model checker," *Int. J. Softw. Tools for Technol. Transf. (STTT)*, vol. 2, no. 4, pp. 410–425, Mar. 2000.
[30] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Trans.*, vol. 15, no. 5, pp. 475–495, Oct. 2013.
[31] A. Pneuli and R. Rosner, "Distributed reactive systems are hard to synthesize," in *Proc. 31st Annu. Symp. Found. Comput. Sci.*, vol. 2, Oct. 1990, pp. 746–757.
[32] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 404–415, Oct. 2006.
[33] R. Alur, M. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, and A. Udupa, "Synthesizing finite-state protocols from scenarios and requirements," in *Hardware and Software: Verification and Testing*. Cham, Switzerland: Springer, 2014, pp. 75–91.
[34] 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.*, vol. 27, no. 2, pp. 651–674, Jun. 2019.
[35] T. Walsh, "SAT υ CSP," in *Proc. 6th Int. Conf. Princ. Pract. Constraint Program.* Berlin, Germany: Springer, 2000, pp. 441–456.
[36] G. S. Tseytin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Berlin, Germany: Springer, 1983, pp. 466–483.
[37] I. Zakirzyanov, A. Morgado, A. Ignatiev, V. Ulyantsev, and J. Marques-Silva, "Efficient symmetry breaking for SAT-based minimum DFA inference," in *Language and Automata Theory and Applications*. Springer, 2019, pp. 159–173.
[38] 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.
[39] G. Holzmann, *The SPIN Model Checker: Primer Reference Manual*. Reading, MA, USA: Addison-Wesley, 2004, p. 608.
[40] N. Lynch, *Distributed Algorithms*. San Mateo, CA, USA: Morgan Kaufmann, 1996.
[41] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*. Berlin, Germany: Springer, 2003, pp. 502–518.
[42] I. Buzhinsky, A. Pakonen, and V. Vyatkin, "Explicit-state and symbolic model checking of nuclear I&C systems: A comparison," in *Proc. 43rd Annu. Conf. IEEE Ind. Electron. Soc.*, Oct./Nov. 2017, pp. 5439–5446.

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, where he is currently pursuing the Ph.D. degree with the Computer Technologies Laboratory.

He finished the one-year program "Algorithmic Bioinformatics" at the Bioinformatics Institute, Saint Petersburg, in 2017. He is currently a Junior Research Associate with the Computer Technologies Laboratory, ITMO University. He studies formal methods, software engineering, SAT and phylogenetics, while teaching students discrete mathematics and working on his Ph.D. degree.

DMITRII SUVOROV received the bachelor's and master's degrees in applied mathematics and informatics from ITMO University, Saint Petersburg, Russia, in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree with the Computer Technologies Laboratory.

He has worked as a Software Engineer. He is currently a Junior Research Associate with the Computer Technologies Laboratory, ITMO University. He also teaches discrete mathematics. His research interests include programming languages, formal verification, and distributed systems.

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.

VALERIY VYATKIN (Senior Member, IEEE) received the Ph.D. and Dr.Sc. degrees in applied computer science from the Taganrog Radio Engineering Institute, Russia, in 1992 and 1999, respectively, the Dr.Eng. degree from the Nagoya Institute of Technology, Japan, in 1999, and the Habilitation degree in Germany, in 2002.

He was a Visiting Scholar with Cambridge University, U.K., and had permanent appointments with the University of Auckland, New Zealand; and Martin Luther University, Germany, as well as in Japan and Russia. He is currently on joint appointment as the Chair of Dependable Computations and Communications, Luleå University of Technology, Sweden, and a Professor of information technology in automation with Aalto University, Finland. He is also the Co-Director of the International Research Laboratory "Computer Technologies," ITMO University, Saint Petersburg, Russia. His research interests include dependable distributed automation and industrial informatics; software engineering for industrial automation systems; artificial intelligence; distributed architectures; and multi-agent systems in various industries: smart grid, material handling, building management systems, datacenteres, and reconfigurable manufacturing.

Dr. Vyatkin received the Andrew P. Sage Award for the Best IEEE TRANSACTIONS paper in 2012. He is the Chair of IEEE IES Technical Committee on Industrial Informatics.