IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 24, 2021, accepted February 1, 2021, date of publication February 8, 2021, date of current version February 18, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3057823

BeBoSy: Behavior Examples Meet Bounded Synthesis

DANIIL CHIVILIKHIN[1], ILYA ZAKIRZYANOV[1,2], AND VLADIMIR ULYANTSEV[1]
[1]Computer Technologies Laboratory, ITMO University, St. Petersburg 197101, Russia
[2]JetBrains Research, St. Petersburg 197374, Russia

Corresponding author: Daniil Chivilikhin (chivdan@itmo.ru)

This work was supported in part by the RFBR, under Project 18-07-01285, and in part by the JetBrains Research.

ABSTRACT Bounded synthesis generates a state machine satisfying given temporal specification in linear temporal logic (LTL) while bounding the number of states of the target solution. Bounded synthesis methods currently do not support finite-length behavior examples, which are often helpful when formulating the specification of the desired state machine. In this work we show how to incorporate behavior examples into bounded synthesis methods and exemplify the approach by presenting BeBoSy (*Behavior Examples meet BoSy*), an extension of the bounded synthesis tool BoSy based on reductions to Boolean satisfiability (SAT) and Quantified Boolean formula satisfiability (QSAT) problems. The proposed approach is to augment the encodings used in BoSy with additional constraints that ensure the compliance of the generated state machine with the given behavior examples. Experiments with openly available data from the annual reactive synthesis competition SYNTCOMP augmented with random behavior examples show that the efficiency of the proposed approach is superior to both BoSy (with naïve representation of behavior examples with LTL formulas) and a state-of-the-art counterexample-guided approach EFSM-tools.

INDEX TERMS Bounded synthesis, LTL synthesis, inference algorithms, Boolean satisfiability, quantified Boolean formula.

I. INTRODUCTION

Finite-state machines are widely used for system design [27], testing [3], [21], and verification [5], especially in the domain of control systems and cyber-physical systems. Manual ad-hoc creation of a finite-state machine for a concrete problem is often a non-trivial, laborious, and error-prone process. Thus, a vast amount of research effort has been dedicated to creation of methods for *automatic synthesis* of finite-state machines from specification of various forms [4], [7]–[9], [12], [14], [23], [26], [28].

One type of specification is a set of behavior examples of finite length that the sought state machine must satisfy. Behavior examples as a specification are helpful when the goal of state machine synthesis is to find a model of some existing (legacy) system based on its known behavior. In this case, behavior examples can be recorded automatically from the legacy system or its simulation model. The problem of finding the minimal deterministic finite automaton satisfying given labeled words (a type of behavior examples) is NP-complete [13]. Methods capable of finding minimal state machines satisfying given behavior examples are mostly based on translation to the Boolean satisfiability problem (SAT) [4], [5], [7], [14], [26] and related problems [8], [9]. Methods based on heuristics such as state merging are useful for finding moderate-size models [12], [15], [28], but do not guarantee that the found model will be minimal.

Another widely used type of specification are linear temporal logic (LTL) formulas [20]. These express properties of state machine behavior that are not bounded to finite length. This type of specification is helpful when synthesizing a state machine "from scratch", solely based on design requirements. The problem of synthesizing a state machine satisfying given LTL properties (so-called *LTL synthesis problem*) is 2EXPTIME-complete [24]. However, despite this complexity result, if the size of the LTL specification is small or moderate, LTL synthesis can be solved efficiently for many practical examples. One of the most successful approaches to LTL synthesis is *bounded synthesis* [8], [9], [11], in which the number of states of the synthesized state machine (so-called *transition system*) is bounded, allowing to construct solutions

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

with the minimum number of states, unlike other approaches to LTL synthesis [19], [22].

Described types of specification, behavior examples and temporal properties, may be combined [5], [6], [26], [28]. Indeed, when searching for a finite-state model of a legacy system based on its behavior examples, it is often helpful to restrict its possible behavior by ensuring its compliance with temporal properties which may be derived from documentation or analysis. Also, when synthesizing a system "from scratch", use of behavior examples together with LTL properties may ease the effort of specification preparation, since behavior examples are simpler to comprehend than LTL formulas, especially to engineers not familiar with formal methods.

Known methods allowing synthesis of minimal-sized state machines from both behavior examples and temporal properties [5], [6], [26] are based on the use of SAT solvers and counterexample-guided inductive synthesis (CEGIS) [2], [18], [25]. CEGIS is an iterative procedure, in which the model is synthesized from positive and negative examples, where negative examples are produced by verifying candidate models against temporal properties and collecting counterexamples produced by a model checker. A weak side of this approach is that the number of CEGIS iterations (synthesize, verify, forbid counterexamples) may be large if the target state machine is not sufficiently covered with behavior examples. In particular, use of the CEGIS approach for synthesis only from LTL properties is practically infeasible, since it is effectively reduced to random search loosely guided by model checking results.

Another approach to solve the problem is to encode behavior examples with LTL properties and use existing LTL synthesis tools such as BoSy [8], [9]. However, scalability of such a naïve approach, as shown below in this paper, is limited, since conversion of behavior examples to LTL dramatically increases the overall size of the LTL specification.

Thus, to our best knowledge, currently there is no universal approach to synthesizing finite-state machines (transition systems) from LTL properties and finite-length behavior examples simultaneously: EFSM-tools requires sufficient completeness of the set of behavior examples, and BoSy cannot efficiently account for behavior examples. In this paper we address this challenge by creating a new method for solving the problem of bounded synthesis from both LTL properties and behavior examples. The contributions of this paper are the following.

1) We propose extensions of SAT and Quantified SAT (QSAT, or Quantified Boolean Formula, QBF) based methods implemented in the LTL synthesis tool BoSy [8], [9] that allow the use of behavior examples as additional input data for bounded synthesis. The proposed extensions are additional SAT/QSAT constraints that ensure the compliance of the synthesized transition system with given behavior examples. This result contributes to the state of the art in the area of synthesis by extending the area of applicability of bounded synthesis methods, making them applicable to synthesis from both behavior examples and LTL properties.
2) We perform the first (to the best of our knowledge) rigorous experimental study comparing state-of-the-art state machine synthesis methods from two different research directions: bounded synthesis (BoSy) and counterexample-guided synthesis (EFSM-tools). We compare the proposed approach with these state-of-the-art methods using openly available reactive synthesis competition (SYNTCOMP) benchmark instances [16], [17] augmented with randomly generated behavior examples. We find that for the considered synthesis problem bounded synthesis is superior to counterexample-guided synthesis.

The rest of this paper is structured as follows. In Section II we review SAT and QSAT based bounded synthesis methods proposed in [8], [9] that allow generating transition systems from LTL properties only. Then, Section III describes the proposed approach to augmenting these methods with the new type of input data, behavior examples. Section IV presents the experimental evaluation, comparing the proposed approach with state-of-the-art methods. Finally, Section V concludes the paper with a short discussion of results and directions of future work.

II. BOUNDED SYNTHESIS WITH BoSy

In this section we give a brief overview of the bounded synthesis approaches [8] implemented in the BoSy [9] tool. We describe only the details essential for understanding the way how existing BoSy encodings work, and our modifications. In Section II-A we introduce required definitions. In Section II-B we describe the general approach to bounded synthesis implemented in BoSy. Then, in Section II-C and Section II-D we describe two encodings of bounded synthesis proposed in [8]: the so-called explicit encoding using SAT and the input-symbolic encoding using QSAT.

A. DEFINITIONS

1) LINEAR TEMPORAL LOGIC

Linear temporal logic [20] (LTL) is a formal specification language used to formulate statements about the states of the system at different moments of time. An LTL formula may include propositional variables (here, input and output variables of the state machine or predicates assembled from them), Boolean connectives $\vee$, $\wedge$, $\neg$, $\rightarrow$, and a set of temporal operators: $\mathbf{G}f$ (Always, f holds for all states starting from the current one), $\mathbf{X}f$ (ne**X**t, f holds for the next state), $\mathbf{F}f$ (Eventually, f holds for some consequent state), $g\mathbf{U}f$ (**U**ntil, g holds at least until f becomes true, which must hold for the current or some future state). For example, if x_1, x_2, and z_1 are propositional variables, then $\mathbf{G}((x_1 \wedge \neg x_2) \rightarrow \mathbf{X}(\mathbf{F}z_1))$ is an LTL formula meaning "Always: if x_1 and not x_2 then, starting from the next state, eventually z_1 will be true".

2) UNIVERSAL CO-BÜCHI AUTOMATON

A universal co-Büchi automaton $\mathcal{A}$ is a tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state,

Σ is a finite alphabet, $\delta : Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is a set of rejecting states. For an infinite word $\sigma \in (2^{\Sigma})^{\omega}$, a run of $\mathcal{A}$ on σ is an infinite state sequence $q_0 q_1 q_2 \cdots \in Q^{\omega}$, where $(q_i, \sigma_i, q_{i+1}) \in \delta, i \geq 0$. A run is called accepting, if there are only finitely many rejecting states in the corresponding state sequence. $\mathcal{A}$ accepts a word ω if all runs of ω are accepting. The language of $\mathcal{A}$ is denoted as $\mathcal{L}(\mathcal{A})$ and is defined as $\{\sigma \in (2^{\Sigma})^{\omega} \mid \mathcal{A} \text{ accepts } \sigma\}$.

3) TRANSITION SYSTEM

A transition system $\mathcal{T}$ is a tuple $\langle T, t_0, \Sigma = I \cup O, \tau\rangle$, where T is a finite set of states, $t_0 \in T$ is the initial state, I is a finite set of propositional variables controllable by the environment (*inputs*), O is a finite set of propositional variables controllable by the system (*outputs*), and $\tau : T \times 2^I \to 2^O \times T$ is the transition function. The transition function τ maps a state 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'. Given an infinite ω-word $\boldsymbol{i_0 i_1} \cdots \in (2^I)^{\omega}$ over the inputs, $\mathcal{T}$ produces an infinite trace $(\{t_0\} \cup \boldsymbol{i_0} \cup \boldsymbol{o_0})(\{t_1\} \cup \boldsymbol{i_1} \cup \boldsymbol{o_1}) \cdots \in (2^{T \cup I \cup O})^{\omega}$, where $\tau(t_j, \boldsymbol{i_j}) = (\boldsymbol{o_j}, t_{j+1})$ for every $j \geq 0$.

4) RUN GRAPH

The product of a transition system $\mathcal{T} = \langle T, t_0, \Sigma = I \cup O, \tau\rangle$ and a universal co-Büchi automaton $\mathcal{A} = \langle Q, q_0, \Sigma, \delta, F\rangle$ is called a *run graph* [11]. A run graph is defined as graph $G = \langle V = T \times Q, E\rangle$, where $E \subseteq V \times V$ is the edge relation such that $((t, q), (t', q')) \in E \Leftrightarrow (\exists \boldsymbol{i} \in 2^I)(\exists \boldsymbol{o} \in 2^O)\tau(t, \boldsymbol{i}) = (\boldsymbol{o}, t') \wedge (q, \boldsymbol{i} \cup \boldsymbol{o}, q') \in \delta$.

B. BOUNDED SYNTHESIS

The reactive synthesis problem is to decide whether there exists a state machine (transition system) satisfying the given LTL specification. Bounded synthesis [11] is able to generate the minimal (in terms of number of states) transition system satisfying given LTL properties. In BoSy, a given LTL formula φ is transformed to a universal co-Büchi automaton $\mathcal{A}$, which accepts the language $\mathcal{L}(\varphi)$. The target transition system $\mathcal{T}$ satisfies the specification φ if and only if every trace generated by $\mathcal{T}$ belongs to language $\mathcal{L}(\varphi)$.

Authors of [11] propose to connect a transition system $\mathcal{T}$ with a co-Büchi automaton $\mathcal{A}$ using a *run graph* and an annotation function $\lambda : T \times Q \to \{\perp\} \cup \mathbb{N}$, which assigns run graph vertices either a natural number k or the value $\perp$ in case the vertex is unreachable. The annotation function is deemed correct if two conditions are satisfied.

First, the pair of initial nodes (t_0, q_0) is assigned a natural number. Second, if the pair (t, q) is assigned a natural number k, then for all $\boldsymbol{i} \in 2^I$ and $\boldsymbol{o} \in 2^O$ such that $\tau(t, \boldsymbol{i}) = (\boldsymbol{o}, t')$ and $(q, \boldsymbol{i} \cup \boldsymbol{o}, q') \in \delta$, the pair (t', q') must be assigned a natural number, that is no less than k, or strictly greater than k if $q' \in F$. This is denoted $\lambda(t', q') \rhd_{q'} k$, where $\rhd_{q'} \equiv >$ if $q' \in F$, otherwise $\geq$. Synthesis is done by searching for a transition system with a correct annotation.

Since this part of bounded synthesis is not the focus of the current research, we refer the readers to [8], [11] for more details. In the following sections we briefly describe two encodings of the bounded synthesis problem proposed in [8] that we will augment with additional constraints regarding behavior examples: the explicit SAT encoding, and the input-symbolic QSAT encoding.

C. EXPLICIT SAT ENCODING

In the explicit encoding, inputs are considered in the transition function τ explicitly, and a translation to SAT is used. Thus, the size of the generated SAT formula is exponential in the number of inputs. The transition function τ of the transition system is represented with two types of variables: $o_{t,\boldsymbol{i}}$ denotes whether output proposition $o \in O$ should be `True` for state t and input proposition $\boldsymbol{i} \in 2^I$, and $\tau_{t,\boldsymbol{i},t'}$ denotes whether there is a transition from state t to state t' marked with input $\boldsymbol{i} \in 2^I$.

The transition relation of the co-Büchi automaton is represented with Boolean formulas $\delta_{t,q,\boldsymbol{i},q'}$ over the output variables $o_{t,\boldsymbol{i}}$. The annotation function λ is split into two parts: $\lambda^{\mathbb{B}} : T \times Q \to \mathbb{B}$ for representing reachability and $\lambda^{\#} : T \times Q \to \mathbb{N}$ for representing the bound on the number of visits to rejecting states. For each $t \in T$ and $q \in Q$ a Boolean variable $\lambda^{\mathbb{B}}_{t,q}$ is introduced that is `True` if and only if the vertex (t, q) of the run graph is reachable from the root, and a variable $\lambda^{\#}_{t,q}$, which is a bit vector of length $\mathcal{O}(\log(|T| \cdot |Q|))$ that is assigned the value of the binary encoding of $\lambda(t, q)$.

The final SAT formula asserts that the initial vertex of the run graph (t_0, q_0) is reachable, that the transition system is *complete* (meaning that there is at least one transition from each state for every input), and that the annotation function represented by variables $\lambda^{\mathbb{B}}_{t,q}$ and $\lambda^{\#}_{t,q}$ is well-defined and consistent with the transition relation represented by variables $\tau_{t,\boldsymbol{i},i'}$ and $o_{t,\boldsymbol{i}}$ [8]:

$$
\begin{aligned}
&\exists\{\lambda^{\mathbb{B}}_{t,q}, \lambda^{\#}_{t,q} \mid t \in T, q \in Q\}\\
&\exists\{\tau_{t,\boldsymbol{i},t'} \mid (t, t') \in T \times T, \boldsymbol{i} \in 2^I\}\\
&\exists\{o_{t,\boldsymbol{i}} \mid o \in O, t \in T, \boldsymbol{i} \in 2^I\}\\
&\lambda^{\mathbb{B}}_{t_0,q_0} \wedge \left(\bigwedge_{t \in T} \bigwedge_{\boldsymbol{i} \in 2^I} \bigvee_{t' \in T} \tau_{t,\boldsymbol{i},t'}\right)\\
&\wedge \bigwedge_{q \in Q} \bigwedge_{t \in T} \Bigg(\lambda^{\mathbb{B}}_{t,q} \to \bigwedge_{q' \in Q} \bigwedge_{\boldsymbol{i} \in 2^I} \Bigg(\delta_{t,q,\boldsymbol{i},q'}\\
&\to \bigwedge_{t' \in T} \left(\tau_{t,\boldsymbol{i},t'} \to \left(\lambda^{\mathbb{B}}_{t',q'} \wedge \lambda^{\#}_{t',q'} \rhd_{q'} \lambda^{\#}_{t,q}\right)\right)\Bigg)\Bigg)
\end{aligned}
$$

If the SAT formula above is satisfiable, then the resulting transition system can be constructed using values of variables $\tau_{t,\boldsymbol{i},t'}$ and $o_{t,\boldsymbol{i}}$ found by the SAT solver.

D. INPUT-SYMBOLIC QSAT ENCODING

The input-symbolic encoding proposed in [8] differs from the explicit one in the way of handling variables that depend on inputs. The idea is to use a QSAT solver and to add universal quantification over the input variables, thus reducing the size of the formula and avoiding the exponential blow-up of its size produced by the explicit approach. Here, the transition

relation τ and the output function o are treated as Boolean Skolem functions whose domain is the set of assignments of input variables I. Thus, the indices $\boldsymbol{i}$ from $\tau_{t,\boldsymbol{i},t'}$ and $o_{t,\boldsymbol{i}}$ are dropped:

$$\begin{aligned}
&\exists\{\lambda^{\mathbb{B}}_{t,q}, \lambda^{\#}_{t,q} \mid t \in T, q \in Q\}\\
&\forall I\\
&\exists\{\tau_{t,t'} \mid (t,t') \in T \times T\}\\
&\exists\{o_t \mid o \in O, t \in T\}\\
&\lambda^{\mathbb{B}}_{t_0,q_0} \wedge \left(\bigwedge_{t\in T}\bigvee_{t'\in T} \tau_{t,t'}\right)\\
&\wedge \bigwedge_{q\in Q}\bigwedge_{t\in T}\left(\lambda^{\mathbb{B}}_{t,q} \rightarrow \bigwedge_{q'\in Q}\left(\delta_{t,q,q'}\right.\right.\\
&\left.\left.\rightarrow \bigwedge_{t'\in T}\left(\tau_{t,t'} \rightarrow \lambda^{\mathbb{B}}_{t',q'} \wedge \lambda^{\#}_{t',q'} \rhd_{q'} \lambda^{\#}_{t,q}\right)\right)\right)
\end{aligned}$$

E. TRANSITION SYSTEM DETERMINISM CONSTRAINTS

In this work we are targeting controller synthesis, therefore we require that the generated transition system is deterministic. Thus, we augmented both explicit and input-symbolic encodings with additional constraints forcing the synthesized transition system to be deterministic: for each state t and each $\boldsymbol{i} \in 2^I$, only one guard condition of outgoing transitions from t may evaluate to `True`. For the input-symbolic case the following constraint is added, where we additionally assume, without loss of generality, that the set of states T is ordered:

$$\bigwedge_{t\in T}\bigwedge_{t'\in T}\bigwedge_{t''\in T,\, t'<t''} \tau_{t,t'} \rightarrow \neg\tau_{t,t''}.$$

III. BEHAVIOR EXAMPLES MEET BOUNDED SYNTHESIS

In this section we describe several approaches of handling behavior examples (scenarios) in bounded synthesis. The general and overall idea is to augment the SAT and QSAT encodings developed in BoSy [8] with new constraints that would ensure the compliance of the resulting synthesized transition system with given behavior examples (scenarios). The behavior examples are represented using a data structure called a scenario tree, and the new constraints aim to find a correct mapping of nodes and edges of the scenario tree to states and transitions of the synthesized transition system.

Consider the set of constraints defined in BoSy (for, example, the SAT version). They require that the resulting transition system satisfies the given LTL formulas. Essentially, these constraints (denote them F_{LTL}) allow for multiple different resulting transition systems. In our work we are interested only in transition systems that additionally satisfy the given behavior examples. So our new constraints restrict the set of possible resulting transition systems allowed by F_{LTL} to only those that satisfy the behavior examples.

First, in Section III-A behavior examples are formally introduced. Second, in Section III-B, a naïve approach is described in which behavior examples are represented as LTL formulas, and existing off-the-shelf bounded synthesis tools are used. Third, in Section III-C we describe the data structure used for representing scenarios in a more compact way: scenario tree. Then, in Section III-D and in Section III-E, using the scenario tree for representing behavior examples, we describe propositional constraints for both SAT and QSAT encodings that restrict the transition system synthesized by BoSy to comply with the scenario tree. In other words, proposed constraints encode the relation between scenario tree vertices and states of the synthesized transition system, demanding that the synthesized transition system correctly processes behavior examples stored in the scenario tree.

A. BEHAVIOR EXAMPLES

A *behavior example* or a *scenario* $\{\langle \boldsymbol{i}_k, \boldsymbol{o}_k\rangle \mid \boldsymbol{i}_k \in 2^I,\ \boldsymbol{o}_k \in 2^O,\ k \in [1..n]\}$ is a finite sequence of n *scenario elements*, where each k-th element contains a valuation of input variables $\boldsymbol{i}_k \in 2^I$ (input event) and a valuation of outputs variables $\boldsymbol{o}_k \in 2^O$ (output event).

A transition system satisfies a scenario element $\langle \boldsymbol{i}_k, \boldsymbol{o}_k\rangle$ in state t if after processing the input $\boldsymbol{i}_k$ in state t the output becomes equal to $\boldsymbol{o}_k$: $\tau(t, \boldsymbol{i}_k) = (\boldsymbol{o}_k, t')$, where $t' \in T$ is an arbitrary state. And a transition system satisfies a scenario $\{\langle \boldsymbol{i}_k, \boldsymbol{o}_k\rangle \mid k \in [1..n]\}$ if it satisfies all its elements in corresponding states, starting from the initial state t_0: $\tau(t_0, \boldsymbol{i}_1) = (\boldsymbol{o}_1, t'_1)$, $\tau(t'_1, \boldsymbol{i}_2) = (\boldsymbol{o}_2, t'_2)$, etc.

B. NAÏVE APPROACH

The naïve approach to incorporating scenarios into LTL synthesis is to convert each scenario to an LTL formula and use existing LTL synthesis tools, e.g. BoSy. For a scenario $s = \{\langle \boldsymbol{i}_k, \boldsymbol{o}_k\rangle \mid k \in [1..n]\}$ we can construct an LTL formula φ_s equivalent to s:

$$\varphi_s = \boldsymbol{i}_k \rightarrow \boldsymbol{o}_k \wedge \boldsymbol{X}(\boldsymbol{i}_{k+1} \rightarrow \boldsymbol{o}_{k+1} \wedge \boldsymbol{X}(\ldots)).$$

Note that by saying that the scenario s is equivalent to the LTL formula φ_s we mean that any transition system satisfies the scenario s if and only if it satisfies the LTL formula φ_s.

For example, let $I = \{i_1, i_2, i_3\}$, $O = \{o_1, o_2, o_3\}$, and consider the set of input events $\{e_1 = i_1\overline{i_2}\,\overline{i_3}, e_2 = i_2\overline{i_1}\,\overline{i_3}, e_3 = i_3\overline{i_1}\,\overline{i_2}\} \subset 2^I$ and set of output events $\{z_1 = \overline{o_1\ o_2\ o_3}$, $z_2 = o_1\overline{o_2\ o_3}, z_3 = o_2\overline{o_1\ o_3}\} \subset 2^O$. Then, the scenario $s = \langle\langle e_1, z_2\rangle; \langle e_2, z_1\rangle; \langle e_3, z_3\rangle; \langle e_2, z_1\rangle\rangle$ is represented with the following LTL formula φ_s:

$$\begin{aligned}
\varphi_s = (i_1 \wedge \neg i_2 \wedge \neg i_3) &\rightarrow (o_1 \wedge \neg o_2 \wedge \neg o_3\\
\wedge \mathbf{X}(i_2 \wedge \neg i_1 \wedge \neg i_3) &\rightarrow (\neg o_1 \wedge \neg o_2 \wedge \neg o_3\\
\wedge \mathbf{X}(i_3 \wedge \neg i_1 \wedge \neg i_2) &\rightarrow (o_2 \wedge \neg o_1 \wedge \neg o_3\\
\wedge \mathbf{X}(i_2 \wedge \neg i_1 \wedge \neg i_3) &\rightarrow (\neg o_1 \wedge \neg o_2 \wedge \neg o_3)))).
\end{aligned}$$

Such representation increases the size of the LTL specification by $(|I|+|O|)\cdot$(number of elements in scenarios). This leads to an increase in the size of the co-Büchi automaton used in BoSy and also an increased time required for its construction.

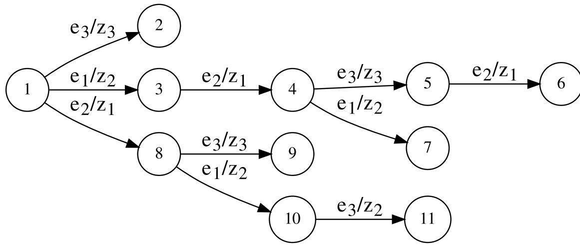


FIGURE 1. Scenario tree constructed from example scenarios (1), each path in the tree starting from its root corresponds to some prefix of some scenario from (1): e.g. scenario $\langle\langle e_1, z_2\rangle; \langle e_2, z_1\rangle; \langle e_3, z_3\rangle; \langle e_2, z_1\rangle\rangle$ corresponds to the path (1, 3, 4, 5, 6).

C. SCENARIO TREE

A scenario tree [14], [26] is a prefix tree containing all prefixes of all scenarios it represents. Each edge is labeled by an input event and an output event. The tree is constructed iteratively. Initially, the scenario tree only contains one vertex, the root. Then, scenarios are added one by one, element by element. Before adding a scenario to the tree, the current vertex is the root of the tree. Then, if the current vertex u has an outgoing edge (u, v) labeled with the same label (input/output pair) as the scenario element currently being added to the tree, then the current vertex is changed to vertex v. If, on the contrary, the current vertex has no such outgoing edge, a new vertex and a new edge with the corresponding label are created in the tree, and the new vertex becomes the current one. Consider for example the following scenarios:

$$\begin{aligned}
&\langle\langle e_3, z_3\rangle\rangle\\
&\langle\langle e_1, z_2\rangle; \quad \langle e_2, z_1\rangle;\ \langle e_3, z_3\rangle;\ \langle e_2, z_1\rangle\rangle\\
&\langle\langle e_1, z_2\rangle; \quad \langle e_2, z_1\rangle;\ \langle e_1, z_2\rangle\rangle\\
&\langle\langle e_2, z_1\rangle; \quad \langle e_3, z_3\rangle\rangle\\
&\langle\langle e_2, z_1\rangle; \quad \langle e_1, z_2\rangle;\ \langle e_3, z_2\rangle\rangle,
\end{aligned} \tag{1}$$

where events $\{e_1, e_2, e_3\}$ and actions $\{z_1, z_2, z_3\}$ have the same definition as above.

The scenario tree corresponding to scenarios (1) is shown in Fig. 1. Note that each path in the tree starting the root node corresponds to some prefix of some scenario from (1). Use of the scenario tree reduces the number of variables required to represent the scenarios in the encoding, since separate variables are needed for each vertex.

D. EXTENDING EXPLICIT SAT-BASED BoSy WITH BEHAVIOR EXAMPLES

The general idea is to organize a mapping between nodes of the scenario tree S and states of the transition system $\mathcal{T}$. As in synthesis from behavior examples (see e.g., [14], [26]), the goal is to find a specific coloring of the scenario tree with N colors, where N is the number of states in $\mathcal{T}$. The coloring must have the following property: when all nodes of the same color are joined together, the resulting transition system must be deterministic and must satisfy the initial scenarios.

To represent node colors, we introduce Boolean variables $c_{j,t}$, where j is a scenario tree vertex and t is a transition system state. For some j and t, $c_{j,t}$ is `True` if and only if vertex j is colored with color t (mapped to state t), i.e. after processing vertex j the transition system is in state t. To simplify notation, for each node $j \in S$ we define a set of tuples $out(j) = \{(j', \boldsymbol{i}, \boldsymbol{o}) \mid \text{there is an edge } j \xrightarrow{i/o} j' \text{ in } S\}$, representing outgoing edges of node j.

The constraint representing the mapping from S to $\mathcal{T}$ can be defined as follows:

$$\bigwedge_{j\in S}\bigwedge_{t\in T} c_{j,t} \rightarrow \bigwedge_{(j',\boldsymbol{i},\boldsymbol{o})\in out(j)}\ \bigvee_{t'\in T} (\tau_{t,\boldsymbol{i},t'} \wedge \boldsymbol{o}_{t,\boldsymbol{i}} \wedge c_{j',t'}). \tag{2}$$

It means that if the scenario tree vertex j is mapped to the transition system state t, then for each edge $j \xrightarrow{i/o} j'$ in S there must be a state t' such that:

- there is a transition from state t to state t' labeled with input $\boldsymbol{i}$ and output $\boldsymbol{o}$;
- vertex j' is mapped to state t'.

Besides this constraint, we must also require the correspondence between the root of the scenario tree and the initial state of the transition system by forcing the variable s_{j_0,t_0} to be `True`. Now, if we add the described constraints to the original SAT encoding from [8], the SAT solver will search for a transition system which not only satisfies given LTL formulas, but is also consistent with given scenarios (behavior examples).

The size of the behavior examples constraint is $\mathcal{O}(n^2 \cdot |S|^2 \cdot |O|)$ clauses and the number of variables is $\mathcal{O}(n \cdot (2^{|I|} \cdot |O| + |S|))$, where $n = |T|$, and $|S|$ is the scenario tree size.

E. EXTENDING INPUT-SYMBOLIC QSAT-BASED BoSy WITH BEHAVIOR EXAMPLES

1) BASIC ENCODING

To start with, let us modify the quantifier-free formula (2) taking into account the fact that variables $\tau_{t,t'}$ and $\boldsymbol{o}_t$ are functions over the inputs now:

$$\bigwedge_{j\in S}\bigwedge_{t\in T} c_{j,t} \rightarrow \bigwedge_{(j',\boldsymbol{i},\boldsymbol{o})\in out(j)}\ \bigvee_{t'\in T} (\boldsymbol{i} \rightarrow (\tau_{t,t'} \wedge \boldsymbol{o}_t \wedge c_{j',t'})). \tag{3}$$

The difference from (2), besides that $\tau_{t,t'}$ and $\boldsymbol{o}_t$ are functions now, is that we check if the current input is the same one as on the edge $j \xrightarrow{i/o} j'$.

Note that variables $\boldsymbol{i}$ and $\boldsymbol{o}_t$ do not depend on t', hence we can take them outside the scope of the operator $\bigvee_{t'\in T}$:

$$\bigwedge_{j\in S}\bigwedge_{t\in T} c_{j,t} \rightarrow \left[\bigwedge_{(j',\boldsymbol{i},\boldsymbol{o})\in out(j)} \boldsymbol{i} \rightarrow \left(\boldsymbol{o}_t \wedge \bigvee_{t'\in T} \left(\tau_{t,t'} \wedge c_{j',t'}\right)\right)\right].$$

The size of the formula is $\mathcal{O}(n\cdot|S|^2\cdot(|O|+|I|+n))$ clauses, and the number of variables is $\mathcal{O}(n \cdot |S|)$.

2) ENCODING WITH A CLUSTERED SCENARIO TREE

The *clustered scenario tree* is a modification of the scenario tree which allows representing behavior examples in a convenient way. As the first step, we build clusters according to

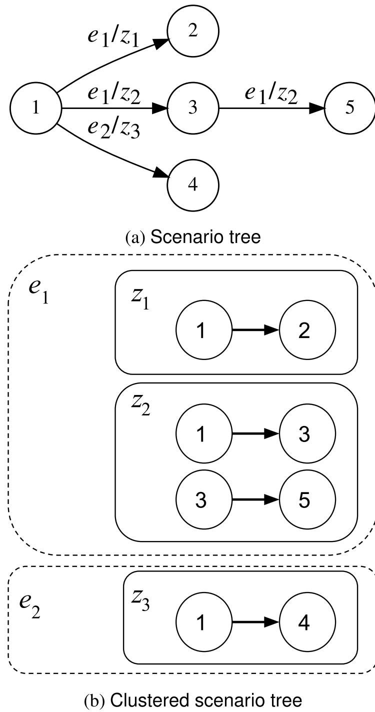


FIGURE 2. An example of a clustered scenario tree: the scenario tree (top figure) is represented in a clustered form (bottom figure) where a top-level cluster corresponds to each unique input (here, e_1 and e_2), and each bottom-level cluster corresponds to a single unique input/output pair: for example, the edges of the scenario tree (1, 3) and (3, 5) which are labeled with the same input/output pair e_1/z_2 are attributed to one bottom-level cluster of the clustered tree.

the inputs on edges. Let $\mathcal{I}_S \subset 2^I$ to be the set of all inputs occurring in the scenario tree. Then, clusters are defined as $Cl(\boldsymbol{i}) = \{(j, j') \mid \boldsymbol{i} \in \mathcal{I}_S \text{ and there is a transition } j \xrightarrow{\boldsymbol{i}/*} j' \text{ in } S\}$. As the second step, we separate the edges inside the cluster according to their outputs, i.e. we build subclusters. A modified version of function $out(j)$ is used to define the subclusters: $out'(j, \boldsymbol{i}) = \{(j', \boldsymbol{o}) \mid \boldsymbol{i} \in \mathcal{I}_S \; \exists \text{ transition } j \xrightarrow{\boldsymbol{i}/\boldsymbol{o}} j' \text{ in } S\}$. An example of a scenario tree and its clustered version is shown in Fig. 2.

Using the clustered version of the scenario tree, we can take variables $\boldsymbol{i}$ out of the scope of all other operators. For each input occurring in behavior examples, we encode the correspondence between the $Cl(\boldsymbol{i})$ cluster vertices and edges and the transition system states and transitions:

$$\bigwedge_{\boldsymbol{i} \in \mathcal{I}_S} \boldsymbol{i} \rightarrow \Bigg[\bigwedge_{j \in Cl(\boldsymbol{i})} \bigwedge_{t \in T} c_{j,t} \rightarrow \Bigg(\bigwedge_{(j', \boldsymbol{o}) \in out'(j, \boldsymbol{i})} \boldsymbol{o}_t \wedge \bigvee_{t' \in T} \left(\tau_{t,t'} \wedge c_{j',t'}\right) \Bigg) \Bigg].$$

The outer expression $\bigwedge_{\boldsymbol{i} \in \mathcal{I}_S} \boldsymbol{i}$ fixes the current value of input $\boldsymbol{i} \in 2^I$, which might ease the inference of Boolean functions $\tau_{t,t'}$ and $\boldsymbol{o}_t$ by the QSAT solver. The size of this formula is $\mathcal{O}(|\mathcal{I}_S| \cdot (|I| + n \cdot |S|^2 \cdot (|O| + n)))$.

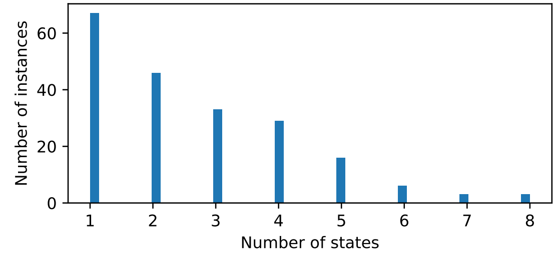


FIGURE 3. Sizes of generated transition systems for SYNTCOMP instances.

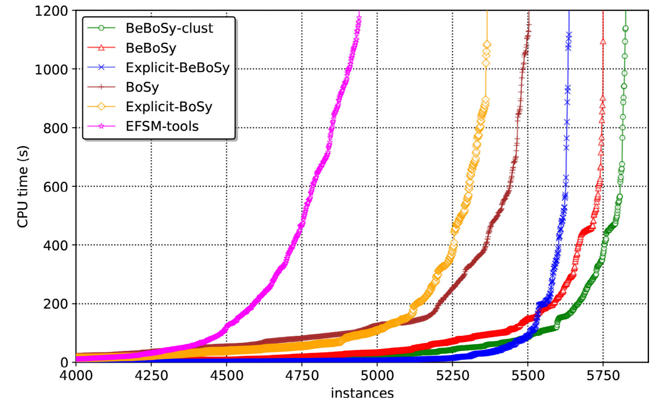


FIGURE 4. Cactus plots for all methods and all instances, plots that reach larger values of the number of instances indicate better performing algorithms: BeBoSy-`clust` solves more instances than all other algorithms, while EFSM-tools solves the smallest amount of instances.

IV. EXPERIMENTAL EVALUATION

The purpose of the experiments was to compare the proposed BeBoSy approach with the original BoSy [8], [9] and EFSM-tools [26] in terms of efficiency. First, in Section IV-A we describe in more detail the EFSM-tools approach, which is one of the state-of-the-art ones in state machine synthesis. Then, in Section IV-B the input data preparation process for considered methods is described. Experimental setup is described in Section IV-C. Results of experiments are reported and discussed in Section IV-D.

A. EFSM-TOOLS: COUNTEREXAMPLE-GUIDED SYNTHESIS OF STATE MACHINES

In this section we describe in more detail the EFSM-tools [26] approach to synthesizing finite-state machines from behavior examples and LTL formulas. Among several methods proposed in [26], we consider the *iterative SAT-based approach* that demonstrated the best performance during experiments reported in the paper. The idea of the approach is to build a minimum-sized state machine from positive finite-length behavior examples and then use a CEGIS [2], [18], [25] loop in order to ensure the compliance of the state machine with the LTL formulas. On each iteration of the CEGIS loop, a SAT encoding is used to generate a state machine that complies with given positive and derived negative behavior examples: the generated state machine should be able to demonstrate all behaviors described by positive behavior examples and

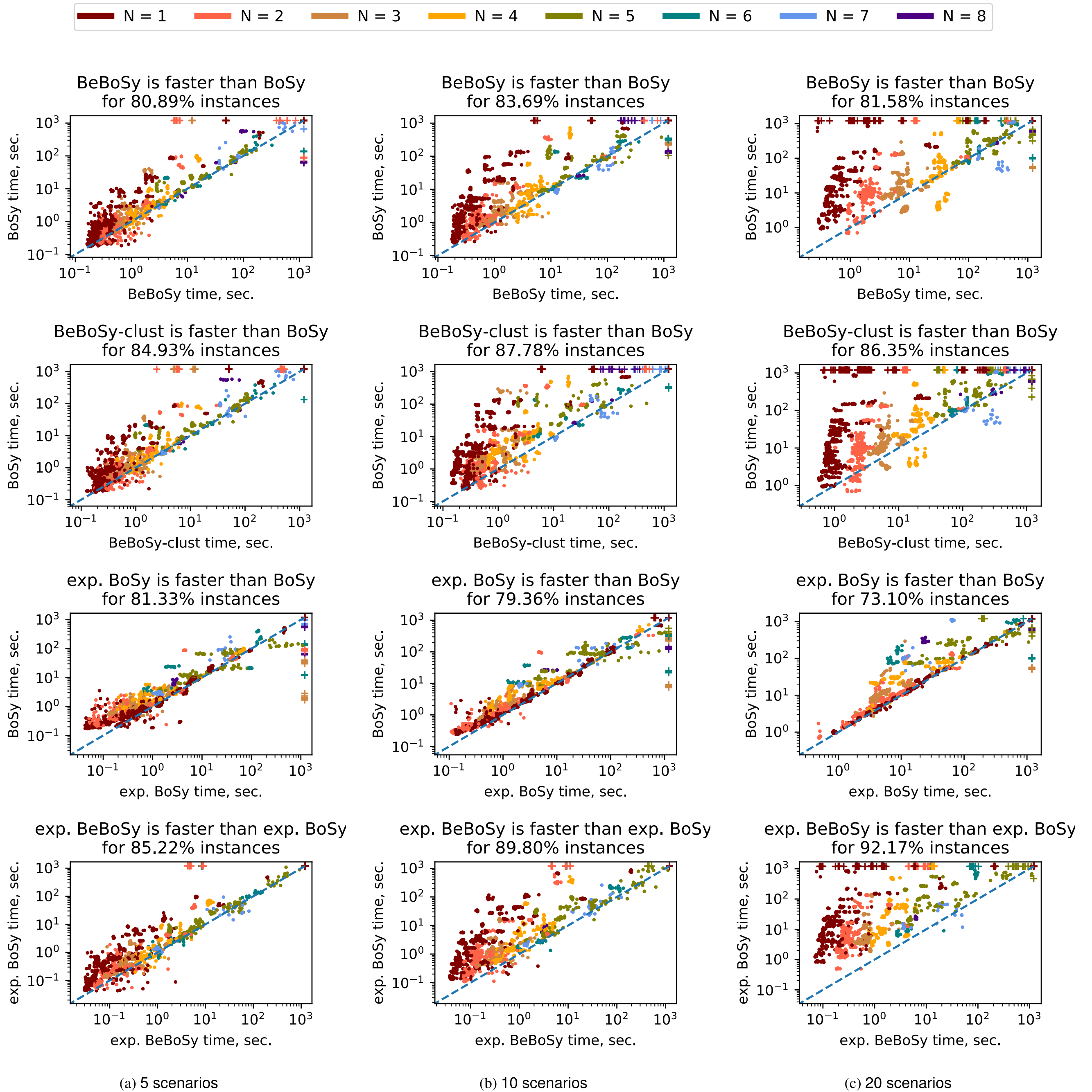


FIGURE 5. Comparison of variants of BeBoSy with variants of BoSy. Each point corresponds to one instance and the running time of the two respective algorithms. Round points denote instances for which both algorithms found a solution within the time limit, crosses denote timeouts. Point color corresponds to the number of states N of the corresponding transition system. BeBoSy is faster than BoSy for 81% to 83% of instances, BeBoSy-`clust` is faster than BoSy for 84% to 88% of instances, Explicit BoSy is faster than input-symbolic BoSy for 73% to 81% of instances, and Explicit BeBoSy is faster than explicit BoSy for 85% to 92% of all instances.

should not exhibit any behaviors described by negative examples. Then the synthesized state machine is checked against LTL formulas by means of a model checker. If counterexamples to LTL formulas are discovered, they are added to the set of negative behavior examples.

Based on the nature of this approach, we can anticipate the following performance. Consider the boundary case when no behavior examples are supplied and only LTL formulas are used as input data. In this case, EFSM-tools can, in principle, be applied, but search will only be done based on negative behavior examples which, informally, describe what the state machine should *not* do. Since the set of undesired behaviors is naturally much larger than the set of desired ones, we can expect that the algorithm will require an extensively

large number of CEGIS iterations to find the correct solution.

Then, as the size of behavior examples is gradually increased, we can expect that the number of iterations needed to find the correct solution should decrease, since the set of known desired behaviors increases. In the boundary case, if the set of positive behavior examples is sufficiently large, only one CEGIS iteration might be sufficient to find a correct state machine.

B. INPUT DATA PREPARATION

For experiments we used instances from the SYNTCOMP-2018 [17] and SYNTCOMP-2019 [1] competitions, specifically, the dataset for the sequential synthesis track. Since SYNTCOMP instances do not feature finite-length behavior examples, the behavior examples were randomly generated based on the transition system generated for SYNTCOMP instances. Therefore, only instances with realizable specifications were considered. Each SYNTCOMP instance has been augmented with randomly generated execution scenarios in the following way:

1) generate the minimal transition system satisfying the original LTL specification using QSAT-based BoSy (with added determinism constraints) with a time limit of one hour;
2) generate random execution scenarios using NuSMV: each scenario is generated by performing a random walk on the NuSMV model of the transition system starting from the initial state;
3) store generated scenarios in three different forms: plain scenarios for BeBoSy, scenarios rewritten in terms of input events for EFSM-tools, and scenarios rewritten using LTL formulas for BoSy.

Within the time limit the QSAT-based BoSy solved 203 instances, and the maximum execution time was 11 minutes. The distribution of the number of states for the generated transition systems is shown in Fig. 3.

For each SYNTCOMP instance we generated sets of $k \in \{5, 10, 20\}$ scenarios, the length of each scenario was also equal to k. Since the scenario generation process is randomized, for each SYNTCOMP instance and each size of scenarios, ten different scenario sets were generated, giving a total of $203 \times 10 = 2030$ instances for each value of k, or a total of 6090 instances.

C. EXPERIMENTAL SETUP

Experiments were run on a computer with an AMD Opteron 6380 CPU @ 2.5 GHz with a memory limit of 10 GB per run of each algorithm on each instance. We compared all variations of the proposed BeBoSy approach (simply BeBoSy for the basic approach and BeBoSy-`clust` for the approach with the clustered scenario tree) with the original BoSy and EFSM-tools. Though in the original paper on BoSy [8] the input-symbolic encoding has been shown to be superior to the explicit one in terms of running time, here we compare both input-symbolic and explicit encodings since the set of

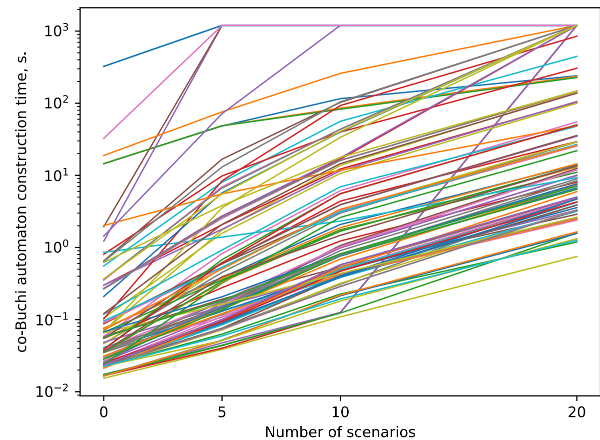


FIGURE 6. Time used in BoSy to construct co-Büchi automaton for different instances from the entire LTL specification augmented with LTL representation of scenarios; each line corresponds to a single instance (target transition system) with varied size of scenarios: the co-Büchi automaton construction time substantially increases as the size of scenarios grows.

instances is different to the one considered in [8] due to added behavior examples. For explicit and input-symbolic BoSy, behavior examples are converted to LTL formulas as described in Section III-B.

For each instance, all tools were run with a time limit of 20 minutes. For all tools, the number of states was not minimized. Instead, it was fixed to the known minimal value found by BoSy during instance preparation. BoSy and BeBoSy were run with the following configuration: `-player system` (search of environment counter-strategies is disabled), `-min-bound` and `-max-bound` set to the known minimum number of states, other parameters were set by default: `spot` for automata conversion, `RAReQS` and `bloqqer` for QBF solving, `cryptominisat5` for SAT solving.

EFSM-tools was run with the configuration: *qbf-automaton-generator.jar* tool was used, `-strategy COUNTEREXAMPLE` (counterexample-guided synthesis method is used), `-size` set to the known minimum number of states, default incremental SAT solver `lingeling`.

D. EXPERIMENTAL RESULTS AND ANALYSIS

First, consider the cactus plots shown in Fig. 4. For each algorithm the plot shows the number of instances that were solved within the time limit. As suggested by these plots, the proposed algorithms BeBoSy-`clust`, Explicit BeBoSy, and BeBoSy clearly dominate the other algorithms. BeBoSy-`clust` appears to be the most efficient algorithm, allowing solving more instances than all other algorithms. Hence, if a competition similar to SYNTCOMP was held with the instances used in this paper, BeBoSy-`clust` would be the victorious algorithm among all other algorithms considered in this paper.

However, the cactus plots only show an overall efficiency comparison. In order to gain a deeper, more detailed understanding, we consider scatter plots: each plot compares

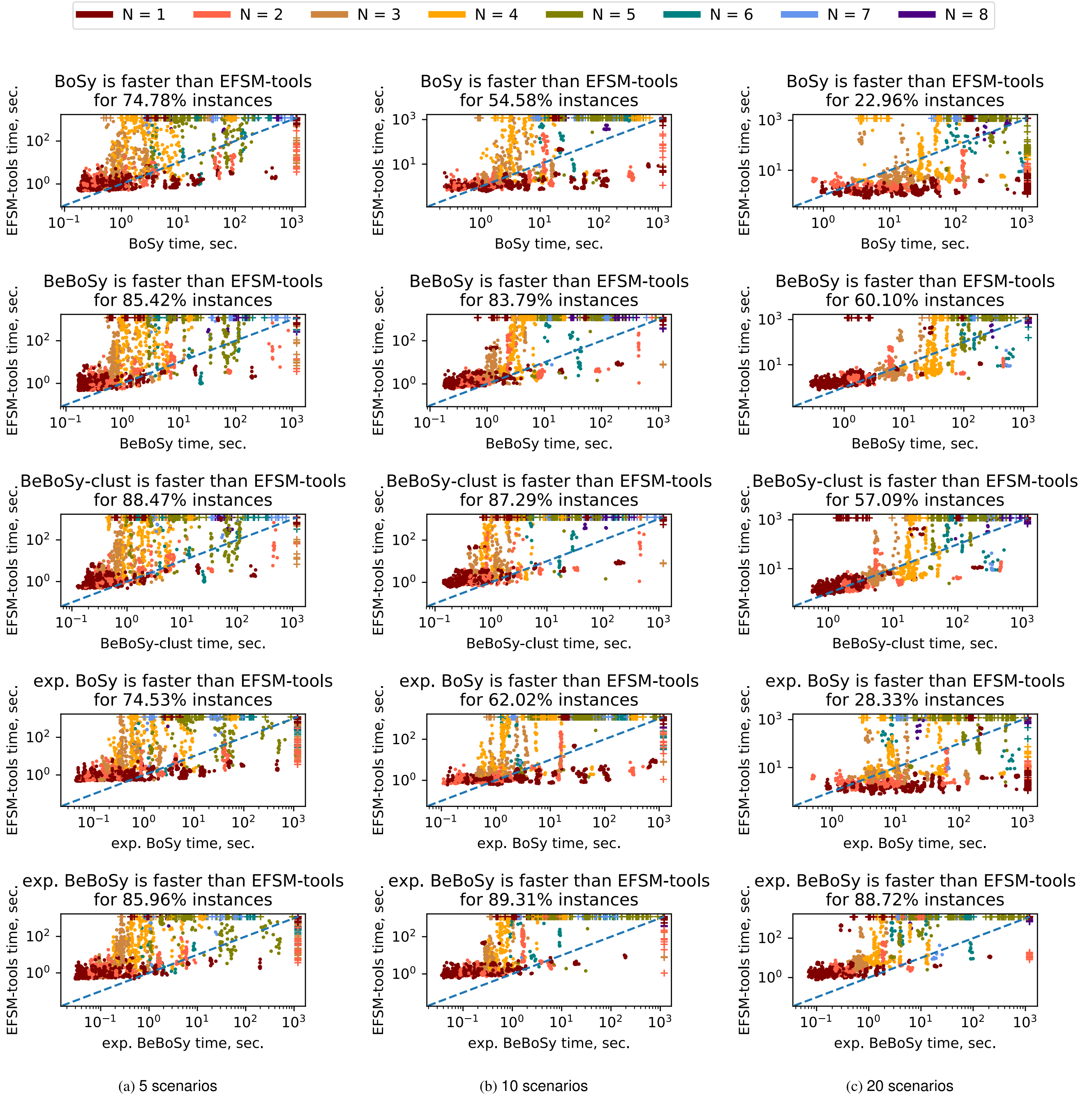
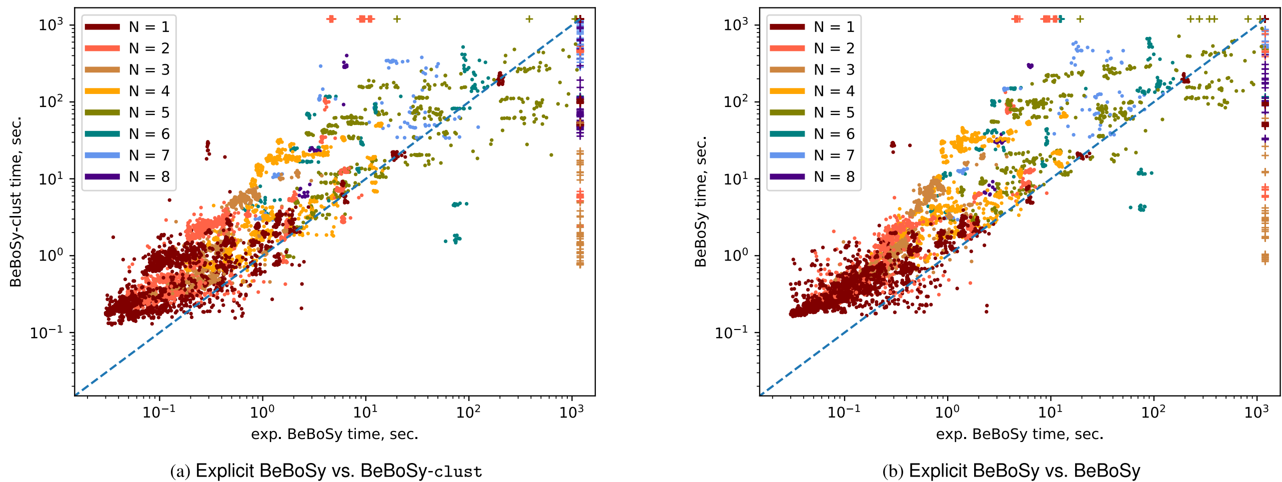


FIGURE 7. Comparison of BoSy and BeBoSy with EFSM-tools: BoSy and BeBoSy dominate EFSM-tools for small sets of scenarios, but loose their advantage as the size of scenarios grows. The exception is the proposed Explicit BeBoSy approach, which is faster than all other algorithms for the majority of instances.

the running time of two algorithms on instances with the specified number of scenarios. Each point corresponds to one instance and the running time of the two respective algorithms. Round points denote instances for which both compared algorithms found a solution within the time limit. Cross-marks denote instances for which at least one of the algorithms timed out. Both axes of the plots are in logarithmic scale. The color of a point corresponds to the number of states N of the corresponding transition system.

Lets us first analyze the effect of our modifications of the original BoSy encodings: compare BeBoSy with the original BoSy. Experimental results in which variations of BoSy are compared to variations of BeBoSy are depicted in Fig. 5.



(a) Explicit BeBoSy vs. BeBoSy-`clust`

(b) Explicit BeBoSy vs. BeBoSy

FIGURE 8. Comparison of algorithms with best performance: BeBoSy, BeBoSy-`clust`, and Explicit BeBoSy: Explicit BeBoSy is faster than BeBoSy-`clust` and BeBoSy for 84% and 88% of instances correspondingly, but has larger amounts of timeouts.

1) BeBoSy AND BeBoSy-`clust` VS. BoSy

BeBoSy is faster than BoSy for 81% to 83% of instances, depending on the number of scenarios. As the size of scenarios increases, BeBoSy becomes more preferential than BoSy, as indicated by the distribution of points moving upwards: this indicates that the difference in running times of the two algorithms increases. An increased number of cross symbols for larger sets of scenarios show that in these cases BoSy exceeds the time limit, while BeBoSy finds a solution within the time limit. BeBoSy-`clust` slightly improves the result of BeBoSy: it is faster than BoSy for 84% to 88% of instances.

2) EXPLICIT BoSy VS. BoSy

Explicit BoSy is faster than input-symbolic BoSy for instances with added behavior examples for 73% to 81% of instances. However, the advantage in the running time is not very significant as seen from the plots: points are situated along the reference line.

3) EXPLICIT BeBoSy VS. EXPLICIT BoSy

For the explicit encodings, the difference in BeBoSy and BoSy performance is more significant than for input-symbolic encodings. It is especially well seen for runs with 20 scenarios, since the points in the plot are significantly shifted upwards from the reference line. Explicit BeBoSy is faster than explicit BoSy for 85% to 92% of all instances.

4) ANALYSIS OF BoSy PERFORMANCE

As we claimed above, BoSy's low efficiency is due to increased size of LTL formulas: larger formulas require more time for constructing the co-Büchi automaton. The plot in Fig. 6 is evidence supporting this claim, showing the time used to construct co-Büchi automata for a representative sample of all instances. Each line corresponds to a single instance with different sizes of scenarios: the time used to construct the co-Büchi automaton increases exponentially with the size of scenarios (the time axis is in logarithmic scale).

5) BoSy AND BeBoSy VS. EFSM-TOOLS

Let us now compare BoSy and BeBoSy with EFSM-tools. Results of experimental runs are depicted in Fig. 7.

In general, all BoSy/BeBoSy algorithms are better than EFSM-tools for the case of five scenarios (Fig. 7a). Many points are shifted upwards along the vertical axis, indicating that BoSy/BeBoSy algorithms are significantly faster. Moreover, for many instances EFSM-tools timed out (shown with crosses), while bounded synthesis methods found a solution. This was to be expected due to low coverage of the target transition system with only five scenarios: the difference is especially well seen for instances with more than two states.

As the number of scenarios increases to 10 (Fig. 7b) and then to 20 (Fig. 7c), the efficiency of most bounded synthesis algorithms decreases. The distribution of points moves to the bottom and to the right, indicating that EFSM-tools starts to perform faster for many instances. This is natural, since larger number of scenarios means higher coverage of target transition systems with scenarios, which is favorable for EFSM-tools: in fact, for most instances for which EFSM-tools is faster it only required one CEGIS iteration.

Then, all BeBoSy algorithms are, in general, faster than BoSy algorithms. For the largest set of scenarios (20), EFSM-tools clearly dominates all other algorithms, except the explicit version of BeBoSy. In fact, the explicit BeBoSy is the best-performing algorithm in terms of running time, being faster than EFSM-tools for 85% to 89% of instances.

Finally, consider the plots in Fig. 8 comparing the best-performing algorithms for all 6090 instances with all sizes of scenarios. According to the plot in Fig. 8a, Explicit BeBoSy is faster than BeBoSy-`clust` for most instances (84%). This, however, comes at the cost of more timeouts for the Explicit

BeBoSy: 453 instances were not solved within the time limit, as opposed to only 265 for BeBoSy-`clust`. The situation is similar for the plot in Fig. 8b: though Explicit BeBoSy is faster than BeBoSy for 88% of instances, it timed out 453 times, and BeBoSy did not find a solution for only 340 instances. According to these results, we can recommend using either the Explicit BeBoSy, or the BeBoSy-`clust` algorithms.

6) BeBoSy-`clust` VS. EXPLICIT BeBoSy WHEN ITERATING NUMBER OF STATES

As a final set of experiments, we ran BeBoSy-`clust` vs. Explicit BeBoSy on the same instances, but with iterating the number of states starting from one. In each such run, each method solves $N_{\min} - 1$ unsatisfiable formulas and one satisfiable one, where $N_{\min}$ is the minimal number of states of a transition system that satisfies the specification. We used a larger time limit of one hour per each experiment. Results are as follows: BeBoSy-`clust` solved 5862 instances, Explicit BeBoSy solved 5712 instances, while Explicit BeBoSy was faster than BeBoSy-`clust` for 90% of instances. These results indicate that the analysis above applies to runs with minimization in a similar way to runs without minimization of the number of states.

V. DISCUSSION & CONCLUSION

In this work we have proposed an approach BeBoSy for incorporating finite-length behavior examples, or scenarios, into the LTL synthesis tool BoSy. Instead of naïvely converting behavior examples to LTL properties, we propose to encode SAT/QSAT constraints that ensure the correspondence of the synthesized transition system with given behavior examples. Through an experimental study performed using SYNTCOMP instances and randomly generated behavior examples we discovered that BeBoSy works faster than BoSy in most cases. Thus, our research contributed to increasing the area of applicability of bounded synthesis, extending it to specifications featuring finite-length behavior examples. The second experimental finding is that the iterative counterexample-guided approach EFSM-tools is faster than considered bounded synthesis algorithms (both BoSy and BeBoSy) when the number of behavior examples is large, and, hence, the coverage of the transition system with behavior examples is high. The exception is the proposed explicit BeBoSy, which is faster than EFSM-tools even for large sizes of behavior examples. Furthermore, EFSM-tools is the worst algorithm in terms of the number of solved instances. The proposed encoding BeBoSy-`clust`, though not being the fastest method, solves more instances than all other considered approaches.

Furthermore, to the best of our knowledge, in this paper we presented the first rigorous experimental comparison of bounded synthesis (BoSy) with counterexample-guided synthesis (EFSM-tools), and tried to establish cases in which each approach should be applied. As a result, we draw the following conclusion. If the specification features only LTL properties, BoSy should be applied. If a small to moderate number of behavior examples are also included, use of our BeBoSy approach is recommended: BeBoSy-`clust` or explicit BeBoSy. If the number of behavior examples is large, then explicit BeBoSy or BeBoSy-`clust` will probably be the best option.

One direction of future work may be to devise a means of bounding other parameters of synthesized transition systems apart from the number of simple cycles as done in [10]: for example, one may bound the number of transitions or even the size of Boolean formulas that represent the transition conditions in the transition system.

ACKNOWLEDGMENT

The authors would like to thank Ruslan Davletshin for implementing the initial version of BeBoSy, and are grateful to the anonymous reviewers for helping improve this article.

REFERENCES

[1] (2019). *Results of SYNTCOMP*. [Online]. Available: http://www.syntcomp.org/syntcomp-2019-results/

[2] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Counterexample guided inductive synthesis modulo theories," in *Computer Aided Verification*. Cham, Switzerland: Springer, 2018, pp. 270–288.

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

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

[5] 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.

[6] 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)*, vol. 1, Sep. 2018, pp. 91–98.

[7] 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.

[8] P. Faymonville, B. Finkbeiner, M. N. Rabe, and L. Tentrup, "Encodings of bounded synthesis," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst. (TACAS)*, 2017, pp. 354–370.

[9] 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.

[10] B. Finkbeiner and F. Klein, "Bounded cycle synthesis," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham, Switzerland: Springer, 2016, pp. 118–135.

[11] B. Finkbeiner and S. Schewe, "Bounded synthesis," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, nos. 5–6, pp. 519–539, 2013.

[12] G. Giantamidis and S. Tripakis, "Learning Moore machines from input-output traces," in *Formal Methods*. Cham, Switzerland: Springer, 2016, pp. 291–309.

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

[14] M. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Proc. Int. Colloq. Conf. Grammatical Inference*, 2010, pp. 66–79.

[15] M. J. H. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Softw. Eng.*, vol. 18, no. 4, pp. 825–856, 2013.

[16] S. Jacobs, R. Bloem, R. Brenguier, A. Khalimov, F. Klein, R. Könighofer, J. Kreber, A. Legg, N. Narodytska, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker, "The 3rd reactive synthesis competition (SYNTCOMP 2016): Benchmarks, participants & results," in *Proc. 5th Workshop Synth.*, Toronto, ON, Canada, 2016, pp. 149–177.

[17] 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," *CoRR*, vol. abs/1904.07736, pp. 1–21, Apr. 2019. [Online]. Available: http://arxiv.org/abs/1904.07736

[18] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, ''Oracle-guided component-based program synthesis,'' in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, vol. 1, New York, NY, USA, May 2010, pp. 215–224.
[19] B. Jobstmann and R. Bloem, ''Optimizations for LTL synthesis,'' in *Proc. FMCAD*, Nov. 2006, pp. 117–124.
[20] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems*. New York, NY, USA: Springer, 1995.
[21] 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.
[22] P. J. Meyer, S. Sickert, and M. Luttenberger, ''Strix: Explicit reactive synthesis strikes back!'' in *Computer Aided Verification*. Cham, Switzerland: Springer, 2018, pp. 578–586.
[23] 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.
[24] R. Rosner, ''Modular synthesis of reactive systems,'' Ph.D. dissertation, Weizmann Inst. Sci., Rehovot, Israel, 1992.
[25] A. Solar-Lezama, ''Program synthesis by sketching,'' Ph.D. dissertation, Univ. California, Berkeley, Berkeley, CA, USA, 2008.
[26] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, ''Exact finite-state machine identification from scenarios and temporal properties,'' *Int. J. Softw. Tools Technol.*, vol. 20, no. 1, pp. 35–55, 2018.
[27] 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.
[28] N. Walkinshaw, R. Taylor, and J. Derrick, ''Inferring extended finite state machine models from software executions,'' *Empirical Softw. Eng.*, vol. 21, no. 3, pp. 811–853, 2016.

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.

ILYA ZAKIRZYANOV received the bachelor's and master's degrees in applied mathematics and informatics and the Ph.D. degree in technical sciences (theoretical foundations of computer science) from ITMO University, Saint Petersburg, Russia, in 2015, 2017, and 2020, respectively.

He is currently a Junior Research Associate with the Computer Technologies Laboratory, ITMO University. His research interests include automata synthesis methods, SAT solvers, and machine learning.

VLADIMIR ULYANTSEV 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 and the Head of the Computer Technologies Laboratory, ITMO University. His research interests include SAT solvers, formal verification and synthesis, bioinformatics, evolutionary algorithms, combinatorial optimization, and machine learning.