Ulyantsev, Vladimir; Buzhinsky, Igor; Shalyto, Anatoly

Exact finite-state machine identification from scenarios and temporal properties

Published in:
International Journal on Software Tools for Technology Transfer

DOI:
10.1007/s10009-016-0442-1

Published: 01/02/2018

Document Version
Peer reviewed version

Please cite the original version:
Ulyantsev, V., Buzhinsky, I., & Shalyto, A. (2018). Exact finite-state machine identification from scenarios and temporal properties. *International Journal on Software Tools for Technology Transfer*, *20*(1), 35-55. https://doi.org/10.1007/s10009-016-0442-1

International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Exact Finite-State Machine Identification from Scenarios and Temporal Properties

Vladimir Ulyantsev[1] ·
Igor Buzhinsky[1,2] ·
Anatoly Shalyto[1]

Received: date / Accepted: date

Abstract Finite-state models, such as finite-state machines (FSMs), aid software engineering in many ways. They are often used in formal verification and also can serve as visual software models. The latter application is associated with the problems of software synthesis and automatic derivation of software models from specification. Smaller synthesized models are more general and are easier to comprehend, yet the problem of minimum FSM identification has received little attention in previous research.

This paper presents four exact methods to tackle the problem of minimum FSM identification from a set of test scenarios and a temporal specification represented in linear temporal logic. The methods are implemented as an open-source tool. Three of them are based on translations of the FSM identification problem to SAT or QSAT problem instances. Accounting for temporal properties is done via counterexample prohibition. Counterexamples are either obtained from previously identified FSMs, or based on bounded model checking. The fourth method uses backtracking. The proposed methods are evaluated on several case studies and on a larger number of randomly generated instances of increasing complexity. The results show that the Iterative SAT-based method is the leader among the proposed methods. The methods are also compared with existing inexact approaches, i.e. the ones which do not necessarily identify the minimum FSM, and these comparisons show encouraging results.

Keywords Finite-state machine identification · linear temporal logic · model checking · SAT · QSAT

This work was financially supported by the Government of Russian Federation, Grant 074-U01, and also partially supported by the Russian Foundation for Basic Research (RFBR), research project No. 14-07-31337 mol_a. We also thank Maxim Buzdalov, Daniil Chivilikhin and anonymous reviewers for useful comments.

Vladimir Ulyantsev
ulyantsev@rain.ifmo.ru

Igor Buzhinsky (corresponding author)
igor_buzhinsky@corp.ifmo.ru

Anatoly Shalyto
shalyto@mail.ifmo.ru

[1] Computer Technologies Laboratory, ITMO University, St. Petersburg, Russia
[2] Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland

1 Introduction

Finite-state models, such as finite-state machines, or FSMs, and deterministic finite automata (DFA), are commonly used for solving various problems arising in software engineering, such as software verification and reverse engineering. Recently, there has been growing interest in automated FSM construction based on given specifications, which are often represented as execution traces and logs [23,38,39,31]. Other types of data employed in model construction are temporal properties [38,9,31] and invariants [3]. This research direction is appealing, since inferred finite-state models can help comprehend software, reveal faults in it, facilitate model-driven development, or even serve as software.

Existing techniques, such as state merging [27,38] and metaheuristic approaches [34,9], demonstrate acceptable performance. However, they are almost not concerned about the size of generated models: it is not always possible to obtain the FSM with the minimum number of states, and even when it is, existing methods do not provide the proof that the found automaton is indeed the smallest possible one. Smaller FSMs are preferred since they are easier to comprehend, to maintain, and, according to the Occam's principle, are more general, which is useful in the cases of incomplete specifications. In particular, if FSMs are further used for test case generation [11,6], the smaller number of states leads to more concise test suites.

In order to address this problem of constructing the smallest possible FSM, or the problem of *exact FSM identification*, this paper presents four exact methods of FSM identification from test scenarios and temporal properties represented in linear temporal logic (LTL) [32]. The results of Gold [21] and Rosner [33] on computational complexity of other finite-state model identification problems make us believe that the consid-

ered problem is NP-hard, although no proof is provided in this paper. The proposed methods are hence based on heuristics. Three of them translate the problem to either the Boolean satisfiability problem (SAT) or its quantified version (QSAT). The SAT problem has been previously used in related research: in [22] the authors learn DFA, and in [35] extended finite-state machines (EFSMs) are identified. Conversely, the translation to QSAT has not been applied in solving such problems. In this paper it is used for FSM construction in combination with bounded model checking [4], which is a form of model checking [12], an approach in formal software verification. The remaining method is based on backtracking. All the methods are incorporated into an open-source tool written in Java.

Another issue, which might be important in reactive software model identification, is completeness – the property of having a transition for each event in each state of the identified FSM. For example, complete FSMs are essential in sequential circuit synthesis [10], finite-state protocol synthesis [1], and in the IEC 61499 international industrial standard [37]. The majority of existing methods neglect this requirement, but is not the case for the proposed techniques.

The proposed methods are evaluated on case studies and randomly generated instances. They are further compared with existing inexact approaches. First, two of them are shown to outperform the approach from [9]. Compared to state merging [38], the proposed approaches need more time, but are applicable under fewer premises (such as the absence of actions on transitions). Finally, the comparison with symbolic bounded LTL synthesis [16,17] suggests that the proposed methods generate notably smaller models.

The rest of the paper is organized as follows. In Section 2, we examine related research. In Section 3, we review several key concepts from the fields of model checking and bounded model checking. The considered problem is formally stated in Section 4. Next, in Section 5, we describe the contribution of the paper: the proposed FSM identification techniques. In Section 6, we evaluate them on case study systems and random instances, and then compare them with other techniques. Section 7 concludes the paper.

2 Related work

Many previously proposed finite-state model identification methods are heuristic. The EDSM state merging algorithm [27] for constructing DFA from a number of words labeled with acceptance/rejection information was among the first ones. State merging starts from an *augmented prefix tree acceptor* (APTA), a tree-shaped automaton, and iteratively merges its states until no valid merge exists. This algorithm serves as the basis for the method of FSM identification from execution traces and LTL safety formulae proposed in [38]. The authors perform a number of state merging executions (the practically efficient Blue Fringe approach is chosen) with the increasing number of negative execution traces, which are obtained as contradictions between the current FSM and LTL properties. The validity of each merge is additionally checked against the temporal properties. This reduces the size of the search space and thus makes the state merging procedure more efficient.

While in [38] LTL properties are either known in advance (in the "passive" approach) or messaged to the FSM identification tool by its user (in the "active" approach), in [28] and [3] they are mined from software traces or logs using predefined templates. In [28], the mined temporal properties are employed to guide state merging so that they are not violated. In [3], the initially constructed compact model is iteratively refined to fulfill the temporal properties and then is additionally coerced to cancel the refinements which are redundant due to an imperfect heuristic refinement procedure. The approach [3] is improved in the work [31], which focuses on learning models whose transitions are annotated with numbers indicating resource (i.e. time or memory) consumption. Inferring models richer than simple discrete transition systems has also been attempted in [39], but the idea in this work is different and does not employ temporal properties: instead, finite-state machines are enriched with numeric data classifiers learnt from traces with data values.

Another group of methods is based on metaheuristics, such as genetic algorithms [30] and ant colony optimization [13]. The genetic algorithm has been applied for EFSM construction in [34], but the work [8] shows that the evolutionary algorithm based on ant colony optimization solves this problem faster.

One of the ways of finding an exact solution (i.e. the FSM with the minimum number of states conforming with the specification), apart from the naive brute-force solution enumeration, is the translation of the problem to another NP-hard problem such as SAT or the constraint satisfaction problem (CSP) and feeding the obtained set of constraints to an exact solver (a third-party tool based on heuristics). To the best of our knowledge, all existing translation-based methods currently do not support temporal specifications. A translation-to-SAT DFA learning method, which employs labeled examples as input data, has been proposed in [22]. This method finds a proper coloring of the so-called *consistency graph*, which determines unmergeable pairs of APTA vertices. The paper [36] improves

this approach by adding breadth-first search (BFS) symmetry breaking predicates to narrow the search space. Another work [35], which is based on [22], introduces a SAT-based method of FSM synthesis from user-prepared behavior examples, or test scenarios. Since one of the methods proposed in our paper is based on the method from [35], we will examine it in more detail in Section 5.1.1.

Finally, the problem of identifying an FSM from both scenarios and temporal properties represented in the LTL language is solved in [9]. The solution called CSP+MuACO*sm* combines the use of a CSP solver and a metaheuristic search with an ant colony optimization algorithm. More precisely, the CSP solver finds the initial solution based on scenarios only, and then it is adjusted metaheuristically to account for temporal formulae. Thus, this approach is inexact.

There has also been a large volume of research concerning the LTL synthesis problem, wherein a reactive system compliant with given LTL properties must be derived [5]. This problem in known to be 2EXPTIME-complete [33] in terms of specification length. While the majority of techniques mentioned above aim to construct a finite-state model which *explains* the behavior of a software system, the LTL synthesis problem requires a software system to be *constructed*. In this case LTL properties are often easier to obtain than traces, since there is no software which can generate them. Recent works, which attempted to solve this problem in a practically feasible way, include the approach to bound the parameters of the target system [17,20], paper describing a tool which implements this idea [16], and an approach based on incremental model refinement [19].

3 Model checking and bounded model checking

Concepts related to *model checking* [12] will be extensively used in the rest of the paper. Model checking is a formal verification technique for finite-state models which suggests describing the specification for the software in the form of *temporal properties*. One of the ways to define temporal properties is *linear temporal logic* (LTL): each property to check is expressed as a formula defined over the set of infinite paths in the *Kripke structure*, a special model of software execution. A Kripke structure [12] M is a quadruple (S_K, I, T, L) where S_K is the set of *states*, $I \subset S_K$ is the set of *initial states*, $T \subset S_K \times S_K$ is the *transition relation*, which must be left-total (that is, from each state there is a transition to at least one state), and $L : S_K \to 2^P$ is the *labeling function*, where P is the set of *atomic propositions*, which characterize states. An example of a Kripke structure with an infinite path is shown in Fig. 1.

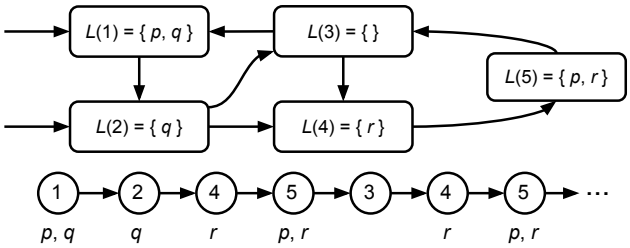

Fig. 1 An example of a Kripke structure (top) and an infinite path in it (bottom). The structure has 5 states, and its labeling function annotates them with three atomic propositions p, q, and r. Two initial states 1 and 2 are marked with incoming arrows from the left. Other arrows describe the transition relation.

LTL formulae consist of Boolean operators ($\wedge$, $\vee$, $\neg$, $\rightarrow$), temporal operators and atomic propositions. If f is simply a Boolean formula, then it asserts that the first state of the path is marked with some atomic propositions and is not marked with some other ones. If f is an LTL formula, then saying that f holds for a state within an infinite path means that it holds for the infinite suffix of the path starting from this state. The following temporal operators can be used:

- The *ne**X**t* operator: $\mathbf{X}\, f$ indicates that formula f holds for the next state of the path.
- The ***G**lobal* operator: $\mathbf{G}\, f$ indicates that f holds for the current state and all future states of the path.
- The ***F**uture* operator: $\mathbf{F}\, f$ indicates that there exists a future state for which f holds, or it already holds for the first state of the path.
- The ***U**ntil* operator: $f\, \mathbf{U}\, g$ indicates that f holds for a finite number of states, and then g holds for the next state.
- The ***R**elease* operator: $f\, \mathbf{R}\, g$ indicates that either g holds until both f and g are true in some state, or g holds forever if f never becomes true.

If f is an LTL formula, then $M \vDash \mathbf{A}\, f$ means that f is satisfied for all infinite paths in M which start in I. Alternatively, $M \vDash \mathbf{E}\, f$ means that there exists a path starting in I for which f is satisfied.

Bounded model checking [4], or BMC, is a technique to approximately verify LTL formulae by reducing the problem to a SAT instance. The idea is to search for a counterexample for the formula being verified among finite execution paths and infinite paths with a simple structure, which enter an infinite loop at some point. Each path, either *finite* or *looping*, is represented as a number of Boolean vectors $s_0, ..., s_k$, where k determines the strength of the verification procedure. This integer can be iteratively increased until a counterex-

ample is found, or a theoretical boundary [4] is reached which proves that BMC of the Kripke structure with the current k is equivalent to its usual model checking, or the employed SAT solver fails to solve the current SAT instance. Each s_j $(0 \le j \le k)$ determines the j-th state of the path.

4 Problem statement

In this work, a *finite state machine* (FSM) is a sextuple $(S, s_{\text{init}}, E, Z, \delta, \lambda)$ where S is a finite set of *states*, $s_{\text{init}} \in S$ is the *initial state*, E is a finite set of input *events*, Z is a finite set of output *actions*, $\delta : S \times E \to S$ is the *transition function*, and $\lambda : S \times E \to Z^*$, where Z^* is the set of strings over Z, is the *output function*. If δ and λ are partial functions defined over the same subset of $S \times E$, then the FSM is called *incomplete*: some of its transitions are missing. Otherwise, if δ and λ are total functions, we call such an FSM *complete*. An FSM execution is a sequence of cycles: on each cycle the FSM receives an input event, generates an output sequence according to λ and changes its active state according to δ.

The considered problem involves identifying an FSM with a fixed number of states $|S|$ which satisfies two types of specification: scenarios and LTL properties. If such an FSM does not exist, this also must be eventually spotted. Note that to find an FSM with the smallest number of states, one might try increasing $|S|$ until a solution is found. The first type of specification is a set of *test scenarios*. A test scenario is a sequence of pairs $(e_1, A_1), ..., (e_n, A_n)$, where each $e_i \in E$ and $A_i \in Z^*$ $(1 \le i \le n)$. These pairs are called *scenario elements*.

The second type of specification is a set of LTL formulae. An FSM *complies* with an LTL formula, if the formula holds for each possible execution of the FSM. We assume the following correspondence between FSMs and their Kripke structures: the Kripke structure's states S_K are FSM's transitions (thus, $S_K \subset S \times E \times Z^* \times S$), and a state of the Kripke structure is initial if and only if it corresponds to an FSM transition from its initial state s_{init}:

$$I = \{(s_{\text{init}}, e, \lambda(s_{\text{init}}, e), \delta(s_{\text{init}}, e)) \mid e \in E\}.$$

Consequently, the pair composed of two states (s_1, e, z, s_2) and (s_1', e', z', s_2') belongs to the transition relation T, if and only if $s_2 = s_1'$. An example of the described transformation is shown in Fig. 2. Finally, to define the labeling function L, we consider the following set of atomic propositions P:

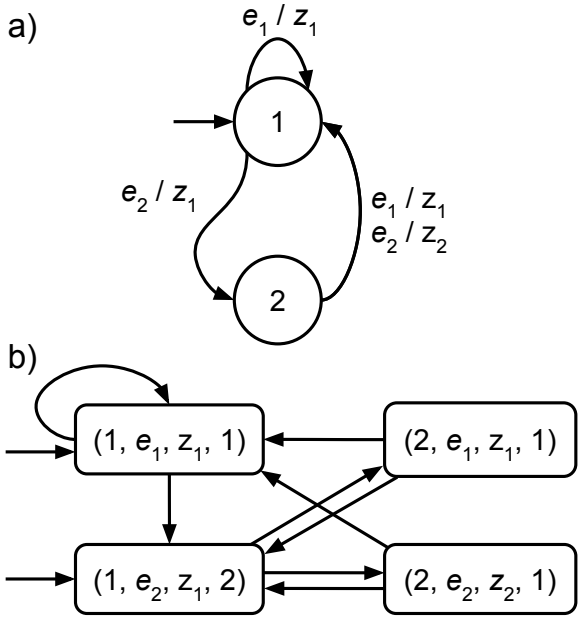

Fig. 2 An example of an FSM (a) and its Kripke structure (b).

- $\texttt{wasEvent}(e)$, $e \in E$: whether the corresponding transition of the FSM is triggered by event e;
- $\texttt{wasAction}(z)$, $z \in Z$: whether the corresponding transition of the FSM includes at least one action z in its output sequence.

The second type of atomic propositions is not sufficient to express constraints which involve the positions of actions in output sequences. Nevertheless, since these propositions were considered in one of the previous works [9] with which we compare our work, we will also use them. Besides, generalizing `wasAction` to handle positions would not essentially influence the proposed methods. Below are some examples of LTL formulae with the defined atomic propositions, which are satisfied for the FSM shown in Fig. 2:

- $\texttt{wasAction}(z_1) \wedge \mathbf{X}(\texttt{wasAction}(z_1) \vee \texttt{wasAction}(z_2))$: the first state of the path is marked with atomic proposition $\texttt{wasAction}(z_1)$ (and possibly with some other atomic propositions), and the second state is marked with either $\texttt{wasAction}(z_1)$ or $\texttt{wasAction}(z_2)$. In terms of the corresponding FSM, this means that z_1 is emitted on the first cycle of FSM execution, and either z_1 or z_2 is emitted on the second cycle.
- $\mathbf{G}(\texttt{wasEvent}(e_1) \to \mathbf{F}(\texttt{wasAction}(z_1)))$: each event e_1 received by the FSM will cause action z_1 in the future.

It is also possible to optionally require the identified FSM to be complete. While describing the FSM identification techniques, we will mention the cases of both presence and absence of the completeness requirement.

The final remark in this section concerns the correspondence of our definitions of the FSM and its identification problem with the ones from previous works. The model of EFSMs considered in [35] and [9] additionally employs guard conditions on transitions. Such conditions depend on Boolean variables – an extra type of input data for an FSM. Nevertheless, any instance of the FSM learning problem with both events and guard conditions can be transformed to an instance with events only. Each event of the transformed instance is a pair of an event from the initial instance and a combination of variable values. Thus, this transformation would increase the number of events in $2^{|V|}$ times, where $|V|$ is the number of variables. For large $|V|$ such a transformation is expensive, but since in this work we deal with $|V| \leq 2$, smarter handling of guard conditions is not considered.

Another FSM definition is the one from [38]. Its main difference with our one is the absence of actions, but the problem stated in [38] assumes the optional presence of negative scenarios – the ones with which the identified FSM must not comply.

5 FSM identification methods

Four exact FSM identification methods are presented in this paper. Our first method, the Iterative SAT-based approach, is largely based on a known method of identifying FSMs from test scenarios only [35] and the idea of iterative counterexample prohibition [38]. The second one, the QSAT-based method, uses the translation of the considered problem to QSAT and involves executing a QSAT solver. Instead, the third approach, named the Exponential SAT-based one, executes a SAT solver on the expanded version of the quantified Boolean formula. Eventually, the fourth and the simplest Backtracking method is based neither on SAT nor on QSAT and performs a heuristic search with backtracking. We implemented the last method to make it the baseline in its comparison with the others. The implementations of all the methods in Java are available online as a cross-platform software tool[1] with a command-line interface.

5.1 Iterative SAT-based solution

The idea of the Iterative SAT-based solution is as follows. We iteratively execute the method of identifying FSMs from test scenarios only, presented in [35], with several adjustments. After each iteration, we verify the obtained FSM with model checking against the LTL specification. We employ the model checker written by the authors of [34] and further modified to make it output minimum counterexamples to falsified formulae. If the FSM's Kripke structure does not comply with the specification, we prohibit the counterexamples found by the model checker using additional Boolean constraints and thus enforce the SAT solver to find a different solution after it is restarted.

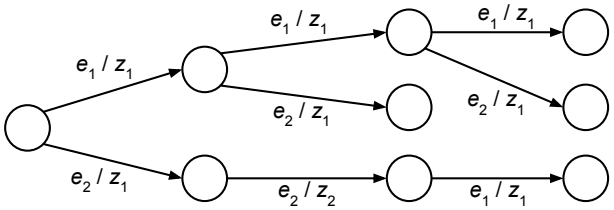

Fig. 3 An example of a scenario tree for four test scenarios: $(e_1, z_1), (e_1, z_1), (e_1, z_1)$; $(e_1, z_1), (e_1, z_1), (e_2, z_1)$; $(e_1, z_1), (e_2, z_1)$; and $(e_2, z_1), (e_2, z_2), (e_1, z_1)$.

An important optimization is to use the capabilities of incremental solvers [15] instead of restarts. On each iteration, only new constraints are fed to the running instance of the solver. This saves computation time, since the number of iterations can be large.

An approach similar to the proposed one, but based on state merging instead of the SAT problem, was introduced in [38]. Another work [19] which applies related ideas is devoted to LTL synthesis.

5.1.1 Method of identifying FSMs from scenarios only

We now shortly describe the method from [35]. In this method, test scenarios are merged into the *scenario tree*. An example of such tree is shown in Fig. 3. Denote the set of tree nodes as V_{sc}. Two variable types are introduced in [35] (we slightly alter the notation from this work):

- $x_{v,i}$: whether node $v \in V_{\text{sc}}$ of the scenario tree corresponds to state i ($1 \leq i \leq |S|$) of the FSM (v is "colored" into color i);
- $y_{i_1,i_2,e}$: whether the transition from state i_1 ($1 \leq i_1 \leq |S|$) triggered by event $e \in E$ leads to state i_2 ($1 \leq i_2 \leq |S|$), i.e. whether $\delta(i_1, e) = i_2$.

A number of constraints enforce the proper coloring of the scenario tree and the compliance of the FSM with this tree. Briefly, these constraints ensure that the first state of the FSM is the initial one (i.e. $x_{1,1} = 1$), that exactly one color (FSM state) is assigned to each node of the tree, that there is no pair of inconsistent nodes [23] with identical colors, that there is at most one transition $y_{i_1,i_2,e}$ in the FSM for each source state i_1 and event e, and that y variables are consistent with

[1] `https://github.com/ulyantsev/EFSM-tools/`

the coloring of the tree. Denote the logical conjunction of all these constraints as $\mathcal{S}$.

5.1.2 Action constraints

In [35], output actions were not included into the SAT model, but were restored based on scenarios. In our case, actions must be considered explicitly to facilitate counterexample prohibition (Section 5.1.3). First we need to introduce an additional variable type for output actions:

– $z_{i,a,e}$: whether the transition from state i triggered by event e produces output action a, i.e. whether $a \in \lambda(i, e)$.

The constraint $\mathcal{Z}$ ensures the compliance of z variables with scenarios by stating that they do not contradict with each edge of the scenario tree. Let $\mathtt{out}(v)$ be the set of outgoing edges from node v, then:

$$\mathcal{Z} = \bigwedge_{v \in V_{\text{sc}}} \bigwedge_{i=1}^{|S|} \left(x_{v,i} \rightarrow \bigwedge_{(e,A,v') \in \mathtt{out}(v)} M_{i,e,A} \right),$$

where $M_{i,e,A} = \bigwedge_{a \in Z} \begin{cases} z_{i,a,e}, & \text{if } a \in A \\ \neg z_{i,a,e}, & \text{if } a \notin A. \end{cases}$

5.1.3 Negative scenario tree

We introduce the concept of the *negative scenario tree*, which is used to represent counterexamples prohibited after each iteration of the method. To do this, we need one more type of variables which will represent the colors of negative scenario tree nodes, the set of which we denote as $\overline{V}_{\text{sc}}$:

– $\overline{x}_{v,i}$: whether node $v \in \overline{V}_{\text{sc}}$ of the negative scenario tree corresponds to state i ($1 \le i \le |S|$) of the FSM.

As in the case of the ordinary, positive scenario tree, the structure of the FSM, encoded in its Boolean model, determines the mapping between tree nodes and FSM states (this will be asserted below with Boolean constraints). However, there are several differences between these types of trees:

– It is possible for negative scenario nodes to not correspond to any of the FSM states. This is intuitive for terminal counterexample nodes, for which the opposite situation would mean that the counterexample belongs to the set of possible FSM behaviors. Some nodes of the tree, nevertheless, correspond to FSM states: when a counterexample is added into the tree, some of its prefixes are still correct.

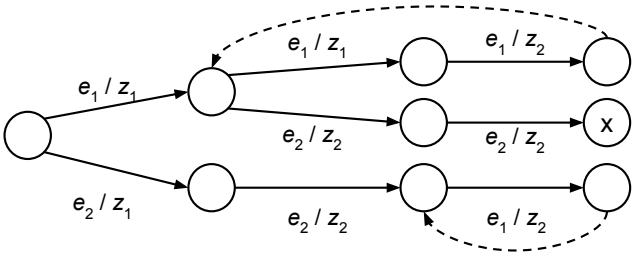

Fig. 4 An example of a negative scenario tree. Two looping counterexamples are $(e_1, z_1), [(e_1, z_1), (e_1, z_2)]$ and $(e_2, z_1), (e_2, z_2), [(e_1, z_2)]$ (cycles are denoted with square brackets), and the single finite counterexample is $(e_1, z_1), (e_2, z_2), (e_2, z_2)$.

– For each node of the positive scenario tree and each event, there cannot be more than one outgoing edge. Otherwise, the tree would require the FSM to be nondeterministic. This restriction does not apply to the negative tree: such a situation just means that more than one combination of actions is prohibited in a particular node for a particular event.
– Generally, each counterexample to an LTL formula is an infinite path, and without loss of generality we may assume that it is composed of a finite prefix followed by a cycle [12]. Moreover, for some formulae there are finite prefixes such that all possible infinite continuations of them are counterexamples, so we will regard such prefixes as counterexamples themselves. A finite counterexample simply corresponds to a path from the root of the tree to the end node of the counterexample. To represent a looping counterexample, after adding the finite prefix and a single occurrence of the cycle, a *back edge* is inserted to link the end of the cycle with its beginning.

An example of a negative scenario tree is shown in Fig. 4. It consists of three counterexamples: two looping ones (back edges are shown in dashed lines) and a finite one (indicated with a cross inside its end node).

Boolean constraints which specify the negative scenario tree are totally different from the ones of the positive tree. First, proper coloring of the negative scenario tree must be ensured. Its root (node 1) corresponds to the initial state of the FSM:

$$\overline{\mathcal{S}}_1 = \overline{x}_{1,1}.$$

Then, negative node colors are propagated along the edges of the tree (excluding back edges) according to the Boolean model of the FSM:

$$\overline{\mathcal{S}}_2 = \bigwedge_{\substack{v \in \overline{V}_{\text{sc}} \\ (e,A,v') \in \mathtt{out}(v) \\ 1 \le i_1, i_2 \le |S|}} (\overline{x}_{v,i_1} \wedge y_{i_1,i_2,e} \wedge M_{i_1,e,A} \rightarrow \overline{x}_{v',i_2}).$$

Similarly to the positive scenario tree, it is possible to ensure that exactly one color is assigned to each negative node, but these constraints can be shown to be redundant.

Next, each added counterexample is associated with its own constraint. The simplest case is adding finite counterexamples: their end nodes (denoted as "terminal" ones) are asserted to not correspond to any states:

$$\overline{\mathcal{S}}_3 = \bigwedge_{v \in \overline{V}_{\text{sc}}\colon \texttt{terminal}(v)} \bigwedge_{i=1}^{|S|} \neg \overline{x}_{v,i}.$$

Note that in general such terminal nodes may still have outgoing edges: this corresponds to situations when a shorter, more restrictive counterexample is added after a longer one.

Next, recall that a looping counterexample is added to the tree as a path consisting of the before-the-cycle prefix, a single occurrence of the cycle, and a back edge which links the end of the cycle to its beginning (see Fig. 4). In general, such an end node may still have outgoing edges due to previously added counterexamples. The respective constraints state that cycles are forbidden: their start and end nodes (the ones linked by back edges) cannot have identical colors, i.e. they do not correspond to the same state of the FSM:

$$\overline{\mathcal{S}}_4 = \bigwedge_{v \in \overline{V}_{\text{sc}}} \bigwedge_{u\colon \texttt{backEdge}(v,u)} \bigwedge_{i=1}^{|S|} \neg(\overline{x}_{v,i} \wedge \overline{x}_{u,i}).$$

Finally, the overall constraint on the negative scenario tree is denoted as $\overline{\mathcal{S}}$:

$$\overline{\mathcal{S}} = \overline{\mathcal{S}}_1 \wedge \overline{\mathcal{S}}_2 \wedge \overline{\mathcal{S}}_3 \wedge \overline{\mathcal{S}}_4.$$

5.1.4 FSM completeness

To simply resolve the FSM completeness issue, we add the completeness constraint which ensures that for every state i_1 and every event e there exists a transition to some state i_2:

$$\mathcal{C} = \mathcal{C}_\forall = \bigwedge_{i_1=1}^{|S|} \bigwedge_{e \in E} \bigvee_{i_2=1}^{|S|} y_{i_1,i_2,e}.$$

However, even when completeness is not required, we must ensure that there is at least one transition from each state of the FSM to prevent vague interpretations of LTL formulae, which are defined over infinite paths. This can be done with a weaker constraint:

$$\mathcal{C} = \mathcal{C}_\exists = \bigwedge_{i_1=1}^{|S|} \bigvee_{e \in E} \bigvee_{i_2=1}^{|S|} y_{i_1,i_2,e}.$$

5.1.5 Symmetry breaking

In addition, symmetry-breaking constraints [36] are used to speed up solver execution on unsatisfiable problem instances. They ensure that the states of the FSM are traversed in the BFS order. We denote them as $\mathcal{B}$. The use of $\mathcal{B}$ requires additional variables described in [36], but we omit them for simplicity.

5.1.6 Assembled formula

Finally, we assemble all the mentioned constraints into the formula which is fed to the SAT solver:

$$\exists\{x_{v,i},\ y_{i_1,i_2,e}, z_{i,a,e}, \overline{x}_{v,i}\} :\ \mathcal{S} \wedge \mathcal{Z} \wedge \overline{\mathcal{S}} \wedge \mathcal{C} \wedge \mathcal{B}. \qquad (1)$$

The Iterative SAT-based solution is summarized in Algorithm 1. The function `ModelCheck` runs the model checker on the FSM and the LTL specification and returns minimum counterexamples to falsified formulae, and `SatSolve` runs a SAT solver (in the Iterative SAT-based solution, SAT solving is implemented incrementally, thus on each step only changes in the Boolean formula are fed to the solver). `SatSolve` might fail and return *null*.

```
Data: set of scenarios SC, temporal specification LTL
f ← generate formula (1)
run a SAT solver in the incremental mode
while true do
    FSM ← SatSolve(f)
    if FSM = null then return 'UNSATISFIABLE'
    counterexamples ← ModelCheck(FSM, LTL)
    if counterexamples ≠ ∅ then update S̄ within f
    else return FSM
end
```

Algorithm 1: Iterative SAT-based solution.

5.2 QSAT-based solution

The QSAT-based solution employs BMC. Assume k is the BMC boundary, that is, paths with the length of $k + 1$ are checked for counterexamples. If we find a way of identifying an FSM which satisfies scenarios and LTL properties with the boundary k, then we can iteratively increase k until the FSM satisfies the properties in the unbounded sense (this can be checked with model checking). Such k always exists according to Theorem 1 from [4], and the reasons why this theorem is applicable here will become evident soon.

5.2.1 Idea of the method

In usual BMC, the Kripke structure to be model-checked is assumed to be known in advance. BMC checks whether there are no paths with length bounded with k in this structure for which the negation of the LTL specification hold – such a path would be a counterexample for the specification, which must hold for every path in the Kripke structure. But instead of querying the SAT solver whether there exists such a path, with the help of a QSAT solver we can solve a quantified Boolean formula which states that each path in the model is not a counterexample. Furthermore, we can now assume that the Kripke structure is not known in advance and add the existential part of the formula, which defines the structure to be identified, before the universal one, which specifies the absence of counterexamples. The proof of the correctness of the outlined idea and its formal description are provided below.

Recall the notations $M \vDash \mathbf{A}\, f$ and $M \vDash \mathbf{E}\, f$, which state that f is satisfied either for all paths or for some path in M (see Section 3). Assume M is the Kripke structure which models an FSM complying with scenarios (denote the set of such models as M_{sc}), and for which $M \vDash \mathbf{A}\, g$, where g is the LTL property we require from the FSM. If there are several such properties, assume that g is their logical conjunction. $M \vDash \mathbf{A}\, g$ is equivalent to $\neg(M \vDash \mathbf{E} \neg g)$. Next, we need to utilize two theorems from [4]:

- Theorem 1. $M \vDash \mathbf{E}\, f \Leftrightarrow \exists k \geq 0 : M \vDash_k \mathbf{E}\, f$, where the symbol "$\vDash_k$" denotes property satisfiability in the k-bounded sense.
- Theorem 2. There is a Boolean formula $[\![M, f]\!]_k$ (defined in [4]), which is satisfiable if and only if $M \vDash_k \mathbf{E}\, f$.

Theorem 1 implies: $M \vDash \mathbf{E} \neg g \Leftrightarrow \exists k = k_0 \geq 0 : M \vDash_k \mathbf{E} \neg g$. Thus, if we try $k = k_0$, we need to find M such that $\neg(M \vDash_k \mathbf{E} \neg g)$. Then, according to Theorem 2, $M \vDash_k \mathbf{E} \neg g$ can be expressed as a Boolean formula $\exists p\ [\![M, f]\!]_k$, where p is a variable assignment which determines a path in M (possibly an invalid one, see clarifications below), and $f = \neg g$. Hence, we search for M such that $\exists p\ [\![M, f]\!]_k$ is false. To find M, we start from $k = 0$ and iteratively increase it by one. On each iteration, we solve the following quantified Boolean formula:

$$\exists M \in M_{\text{sc}}\ \forall p\ \neg [\![M, f]\!]_k. \tag{2}$$

If the formula is unsatisfiable, then it is also unsatisfiable for greater k (this can be inferred from the definition of $[\![M, f]\!]_k$ [4]) and thus the desired Kripke structure M does not exist. Otherwise, we verify $M \vDash \mathbf{A}\, f$ with model checking. If the desired Kripke structure M exists, it will be found together with the corresponding FSM when k reaches k_0.

5.2.2 Kripke structure representation and correctness

We have not yet discussed the way M can be represented with Boolean variables and how the constraint (2) can be expressed as a QSAT instance. The translation of the stated problem to QSAT is again based on the method from [35]. If we assume that state 1 is the initial state of the FSM, then y and z variables are sufficient to define the Kripke structure. Thus, we will search both the scenario coloring determined by x variables and the information sufficient to construct the Kripke structure of the FSM. We constrain all three types of variables with $\mathcal{S}$, $\mathcal{Z}$, $\mathcal{B}$ and $\mathcal{C}$ (see Section 5.1).

5.2.3 Path representation and correctness

We have just identified how to express $M \in M_{\text{sc}}$ as a Boolean formula. We now move to the way of defining a path in M (recall that its length is $k+1$). We introduce the following variables for each position j ($0 \leq j \leq k$) of the path:

- $\sigma_{i,j}$: the j-th position of the path is a transition from state i of the FSM;
- $\varepsilon_{e,j}$: the j-th position of the path is a transition triggered by event e;
- $\zeta_{a,j}$: the j-th position of the path is a transition with action a.

Thus, each Boolean vector s_j, introduced in the end of Section 3, is composed of $\sigma_{i,j}$ ($1 \leq i \leq |S|$), $\varepsilon_{e,j}$ ($e \in E$), and $\zeta_{a,j}$ ($a \in Z$). In fact, σ and ε variables are sufficient to determine a path in the Kripke structure, since the action sequence of the transition in the FSM can be uniquely determined from the source state of the transition and its triggering event. Thus, ζ variables are supplemental, but later they will become helpful to express atomic propositions. In Fig. 5, we show an example of a path in a Kripke structure with the corresponding variable assignment.

Which constraints would ensure that an assignment of the introduced variables produces a correct path? We denote this constraint as $[\![M]\!]_k$ (from now on, some notations from [4] are employed):

$$[\![M]\!]_k = \sigma_{1,0} \wedge \mathcal{P}_\sigma \wedge \mathcal{P}_\varepsilon \wedge \mathcal{P}_y \wedge \mathcal{P}_y^k \wedge \mathcal{P}_z, \text{ where}$$

$$\mathcal{P}_\sigma = \bigwedge_{j=0}^{k} \left(\left(\bigvee_{i=1}^{|S|} \sigma_{i,j} \right) \wedge \bigwedge_{i_1=1}^{|S|} \bigwedge_{i_2=i_1+1}^{|S|} \neg \left(\sigma_{i_1,j} \wedge \sigma_{i_2,j} \right) \right),$$

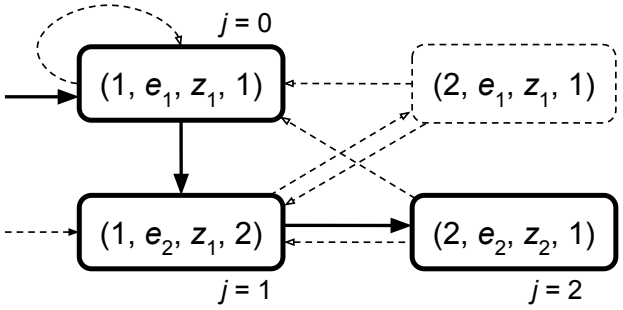

Fig. 5 An example of a path in a Kripke structure shown in Fig. 2 for $k = 2$. The following path variables are true for this path: $\sigma_{1,0}, \varepsilon_{e_1,0}, \zeta_{z_1,0}, \sigma_{1,1}, \varepsilon_{e_2,1}, \zeta_{z_1,1}, \sigma_{2,2}, \varepsilon_{e_2,2}, \zeta_{z_2,2}$, and the other ones are false.

$$\mathcal{P}_\varepsilon = \bigwedge_{j=0}^{k} \left(\left(\bigvee_{e \in E} \varepsilon_{e,j} \right) \wedge \bigwedge_{\{e_1 \neq e_2\}} \neg \left(\varepsilon_{e_1,j} \wedge \varepsilon_{e_2,j} \right) \right),$$

$$\mathcal{P}_y = \bigwedge_{j=0}^{k-1} \bigwedge_{(i_1,i_2,e)} \left(\sigma_{i_1,j} \wedge \varepsilon_{e,j} \wedge \sigma_{i_2,j+1} \rightarrow y_{i_1,i_2,e} \right),$$

$$\mathcal{P}_y^k = \bigwedge_{(i_1,e)} \left(\sigma_{i_1,k} \wedge \varepsilon_{e,k} \rightarrow \bigvee_{i_2} y_{i_1,i_2,e} \right),$$

$$\mathcal{P}_z = \bigwedge_{j=0}^{k} \bigwedge_{(i,a,e)} \left(\sigma_{i,j} \wedge \varepsilon_{e,j} \rightarrow (\zeta_{a,j} \leftrightarrow z_{i,a,e}) \right).$$

The path must start in the initial state of the FSM, therefore we need $\sigma_{1,0}$. The constraints $\mathcal{P}_\sigma$ and $\mathcal{P}_\varepsilon$ check that each transition in the path starts in exactly one state and is triggered by exactly one event, respectively. Among several existing encodings of the at-most-one constraint [24] in $\mathcal{P}_\sigma$ and $\mathcal{P}_\varepsilon$, we have chosen the simplest binomial encoding, since these constraints do not form a significant portion of the final formula. The constraints $\mathcal{P}_y$ and $\mathcal{P}_y^k$ (the special case of $\mathcal{P}_y$ for $j = k$) assert that the transitions in the path correspond to y variables. Note that $\mathcal{P}_y^k$ is not required if the completeness constraint $\mathcal{C}$ is included. Finally, $\mathcal{P}_z$ defines ζ variables, enforcing correct (corresponding to z variables) actions in each state of the path.

5.2.4 Absence of a witness of the formula's negation

By now, the only remaining thing is to express $[\![M, f]\!]_k$ as a Boolean formula. The idea of $[\![M, f]\!]_k$ is to check whether there exists a finite or looping path in M for which f holds – its *witness*. Such path is also a counterexample for the original formula g. According to [4],

$$[\![M, f]\!]_k = [\![M]\!]_k \wedge \mathcal{W},$$

where the path correctness condition $[\![M]\!]_k$ has been discussed previously, and $\mathcal{W}$ expresses the existence condition of a witness of f, the negation of g. More precisely, $\mathcal{W}$ states that there exists a path in the Kripke structure on which the negation f of the required LTL formula g holds. While defining $\mathcal{W}$, we will largely use the derivations from [4].

By ${}_\ell L_k$ $(0 \leq \ell \leq k)$ we denote a Boolean formula which requires a path to be a (k, ℓ)-loop. In such a path, there exists a transition in the Kripke structure from the last position k of the path to some position ℓ. ${}_\ell L_k$ has the following form:

$${}_\ell L_k = \bigvee_{(i_1,i_2,e)} \sigma_{i_1,k} \wedge \varepsilon_{e,k} \wedge \sigma_{i_2,\ell} \wedge y_{i_1,i_2,e}.$$

Note that the looping edge is not included in the Boolean description of the path, and thus $y_{i_1,i_2,e}$ is obligatory in the definition of ${}_\ell L_k$. An example of a looping path with $(2, 0)$ and $(2, 1)$-loops is the one shown in Fig. 5: its last state has transitions to the first two ones. Next, L_k will denote the existence of a (k, ℓ)-loop for at least one ℓ:

$$L_k = \bigvee_{\ell=0}^{k} {}_\ell L_k.$$

Finally, the witness condition is expressed in the following way:

$$\mathcal{W} = \left(\neg L_k \wedge [\![f]\!]_k^0 \right) \vee \bigvee_{\ell=0}^{k} \left({}_\ell L_k \wedge {}_\ell [\![f]\!]_k^0 \right),$$

where $[\![f]\!]_k^0$ and ${}_\ell [\![f]\!]_k^0$ are formula "translations" – constraints produced from the structure of f. The translations can be performed according to the rules defined in [25], but before this f must be transformed to the negation-normal form [25]: all negations must be propagated towards atomic propositions. Atomic propositions encountered at position j of the path are translated in the following simple way:

- $\texttt{wasEvent}(e) = \varepsilon_{e,j}$;
- $\texttt{wasAction}(a) = \zeta_{a,j}$.

5.2.5 Assembled formula

We are now ready to assemble the complete quantified formula (2), which is further fed to a QSAT solver:

$$\begin{aligned} \exists \{x_{v,i},\, y_{i_1,i_2,e},\, z_{i,a,e}\} :\ \forall \{\sigma_{i,j},\, \varepsilon_{e,j},\, \zeta_{a,j}\} : \\ \mathcal{S} \wedge \mathcal{Z} \wedge \mathcal{B} \wedge \mathcal{C} \wedge (\neg [\![M]\!]_k \vee \neg \mathcal{W}). \end{aligned} \tag{3}$$

Variables $x_{v,i}$, $y_{i_1,i_2,e}$, and $z_{i,a,e}$ define the FSM being identified and its Kripke structure. Constraints $\mathcal{S}$, $\mathcal{Z}$ and $\mathcal{B}$ ensure that the assignment of these variables is valid and defines an FSM with BFS symmetry breaking predicates, and $\mathcal{C}$ guarantees the completeness of the

synthesized FSM (or, if completeness is not required, it just forbids states with no outgoing transitions). Next, each assignment of path variables either does not define a correct path ($\neg[\![M]\!]_k$) or is not a witness for f ($\neg\mathcal{W}$).

The pseudocode of the QSAT-based solution is shown in Algorithm 2. The function `QSatSolve` runs a QSAT solver to find a proper FSM, and it returns *null* in case of the unsatisfiability of the formula. The differences between the QSAT-based and the Iterative SAT-based solutions are also stressed in Fig. 6. In addition, a partial example of a QSAT translation is given in Table 1.

```
Data: set of scenarios SC, temporal specification LTL
k ← 0
while true do
    f ← generate formula (3), FSM ← QSatSolve(f)
    if FSM = null then return 'UNSATISFIABLE'
    else if ModelCheck(FSM, LTL)= ∅ then
        return FSM
    end
    else k ← k + 1
end
```

Algorithm 2: QSAT-based solution.

5.3 Exponential SAT-based solution

Any QSAT instance can be transformed to a SAT instance by eliminating every universal quantifier: each formula $\forall x\, f$ is converted to $f|_{x:=0} \wedge f|_{x:=1}$, where the subscript expressions after vertical lines denote variable assignments. If the formula contains q universally quantified variables, then this procedure can bloat its size in up to 2^q times. We take this approach in an optimized form and feed the following constraint to the SAT solver:

$$\exists\{x_{v,i},\ y_{i_1,i_2,e},\ z_{i,a,e}\}:$$
$$\mathcal{S} \wedge \mathcal{Z} \wedge \mathcal{B} \wedge \mathcal{C} \wedge \bigwedge_{t\in X} \left(\neg\mathcal{P}_y \vee \neg\mathcal{P}_y^k \vee \neg\mathcal{W}\right)\Big|_t,$$

where $T = \{\{\sigma_{i,j}, \varepsilon_{e,j}, \zeta_{a,j}\} \mid \sigma_{1,0} \wedge \mathcal{P}_\sigma \wedge \mathcal{P}_\varepsilon \wedge \mathcal{P}_z\}$.

First, constraints $\mathcal{S}$, $\mathcal{Z}$, $\mathcal{B}$ and $\mathcal{C}$ do not depend on path variables and thus are included into the formula only once. Then we iterate over the set T of all valid path variable assignments – the ones for which $\sigma_{1,0}$, $\mathcal{P}_\sigma$, $\mathcal{P}_\varepsilon$, and $\mathcal{P}_z$ hold. It is important to mention that while σ and ε variables are assigned to constants (improper assignments are filtered out by $\sigma_{1,0} \wedge \mathcal{P}_\sigma \wedge \mathcal{P}_\varepsilon$), each $\zeta_{a,j}$ is assigned to the corresponding $z_{i,a,e}$ variable, which is uniquely determined from the $\mathcal{P}_z$ constraint based on σ and ε values. For each path variable assignment, we include the remaining part of the constraint with substituted concrete values of $\sigma_{i,j}, \varepsilon_{e,j}$, and $\zeta_{a,j}$ into the final Boolean formula.

Table 1 Several subformulae of the QSAT translation for $|S| = |E| = |Z| = 2$, the scenario tree from Fig. 3 and the LTL formula $\mathbf{G}(\texttt{wasAction}(z_2) \rightarrow \mathbf{X}\,\texttt{wasAction}(z_1))$. The FSM from Fig. 2 satisfies these data. For this particular example $k = 0$ is sufficient, but $k = 1$ is used instead to make the example nontrivial.

Name	Subformula
Variables	$\exists x_{1..9,1..2}, y_{1..2,1..2,1..2}, z_{1..2,1..2,1..2}$
	$\forall \varepsilon_{1..2,0..1}, \sigma_{1..2,0..1}, \zeta_{1..2,0..1}$
$\mathcal{P}_\sigma$	$(\sigma_{1,0} \vee \sigma_{2,0}) \wedge \neg(\sigma_{1,0} \wedge \sigma_{2,0}) \wedge$
	$(\sigma_{1,1} \vee \sigma_{2,1}) \wedge \neg(\sigma_{1,1} \wedge \sigma_{2,1})$
$\mathcal{P}_\varepsilon$	$(\varepsilon_{1,0} \vee \varepsilon_{2,0}) \wedge \neg(\varepsilon_{1,0} \wedge \varepsilon_{2,0}) \wedge$
	$(\varepsilon_{1,1} \vee \varepsilon_{2,1}) \wedge \neg(\varepsilon_{1,1} \wedge \varepsilon_{2,1})$
$\mathcal{P}_y$	$(\sigma_{1,0} \wedge \varepsilon_{1,0} \wedge \sigma_{1,1} \rightarrow y_{1,1,1}) \wedge$
	$(\sigma_{1,0} \wedge \varepsilon_{2,0} \wedge \sigma_{1,1} \rightarrow y_{1,1,2}) \wedge$
	$(\sigma_{1,0} \wedge \varepsilon_{1,0} \wedge \sigma_{2,1} \rightarrow y_{1,2,1}) \wedge$
	$(\sigma_{1,0} \wedge \varepsilon_{2,0} \wedge \sigma_{2,1} \rightarrow y_{1,2,2}) \wedge$
	$(\sigma_{2,0} \wedge \varepsilon_{1,0} \wedge \sigma_{1,1} \rightarrow y_{2,1,1}) \wedge$
	$(\sigma_{2,0} \wedge \varepsilon_{2,0} \wedge \sigma_{1,1} \rightarrow y_{2,1,2}) \wedge$
	$(\sigma_{2,0} \wedge \varepsilon_{1,0} \wedge \sigma_{2,1} \rightarrow y_{2,2,1}) \wedge \ldots$
$\mathcal{P}_z$	$(\sigma_{1,0} \wedge \varepsilon_{1,0} \rightarrow (\zeta_{1,0} \leftrightarrow z_{1,1,1})) \wedge$
	$(\sigma_{1,0} \wedge \varepsilon_{2,0} \rightarrow (\zeta_{1,0} \leftrightarrow z_{1,1,2})) \wedge$
	$(\sigma_{1,0} \wedge \varepsilon_{1,0} \rightarrow (\zeta_{2,0} \leftrightarrow z_{1,2,1})) \wedge$
	$(\sigma_{1,0} \wedge \varepsilon_{2,0} \rightarrow (\zeta_{2,0} \leftrightarrow z_{1,2,2})) \wedge$
	$(\sigma_{2,0} \wedge \varepsilon_{1,0} \rightarrow (\zeta_{1,0} \leftrightarrow z_{2,1,1})) \wedge$
	$(\sigma_{2,0} \wedge \varepsilon_{2,0} \rightarrow (\zeta_{1,0} \leftrightarrow z_{2,1,2})) \wedge \ldots$
${}_0L_1$	$(\sigma_{1,1} \wedge \varepsilon_{1,1} \wedge \sigma_{1,0} \wedge y_{1,1,1}) \vee$
	$(\sigma_{1,1} \wedge \varepsilon_{2,1} \wedge \sigma_{1,0} \wedge y_{1,1,2}) \vee$
	$(\sigma_{1,1} \wedge \varepsilon_{1,1} \wedge \sigma_{2,0} \wedge y_{1,2,1}) \vee$
	$(\sigma_{1,1} \wedge \varepsilon_{2,1} \wedge \sigma_{2,0} \wedge y_{1,2,2}) \vee$
	$(\sigma_{2,1} \wedge \varepsilon_{1,1} \wedge \sigma_{1,0} \wedge y_{2,1,1}) \vee$
	$(\sigma_{2,1} \wedge \varepsilon_{2,1} \wedge \sigma_{1,0} \wedge y_{2,1,2}) \vee$
	$(\sigma_{2,1} \wedge \varepsilon_{1,1} \wedge \sigma_{2,0} \wedge y_{2,2,1}) \vee$
	$(\sigma_{2,1} \wedge \varepsilon_{2,1} \wedge \sigma_{2,0} \wedge y_{2,2,2})$
$[\![f]\!]_1^0$	$(\zeta_{2,0} \wedge \neg\zeta_{1,1}) \vee (\zeta_{2,1} \wedge \text{false})$
${}_0[\![f]\!]_1^0$	$(\zeta_{2,0} \wedge \neg\zeta_{1,1}) \vee (\zeta_{2,1} \wedge \neg\zeta_{1,0})$
${}_1[\![f]\!]_k^0$	$(\zeta_{2,0} \wedge \neg\zeta_{1,1}) \vee (\zeta_{2,1} \wedge \neg\zeta_{1,1})$

5.4 Backtracking solution

The solution based on backtracking is the baseline one and does not involve SAT or QSAT solver execution. A recursive procedure iterates over various (possibly incomplete) FSMs, starting from the FSM with no transitions. It maintains the current set of edges of the scenario tree which can not yet be passed by the FSM due to the absence of transitions – the *frontier* (see Fig. 7 for an example).

If the frontier is not empty, then the procedure tries augmenting the FSM with one of its edges. Each new FSM A is checked for compliance with the scenario tree, and if it complies with it, then the new frontier is found. Moreover, A is verified and thus again can be rejected. The rationale behind verifying intermediate FSMs is as follows. If A is incomplete, then the set of paths in

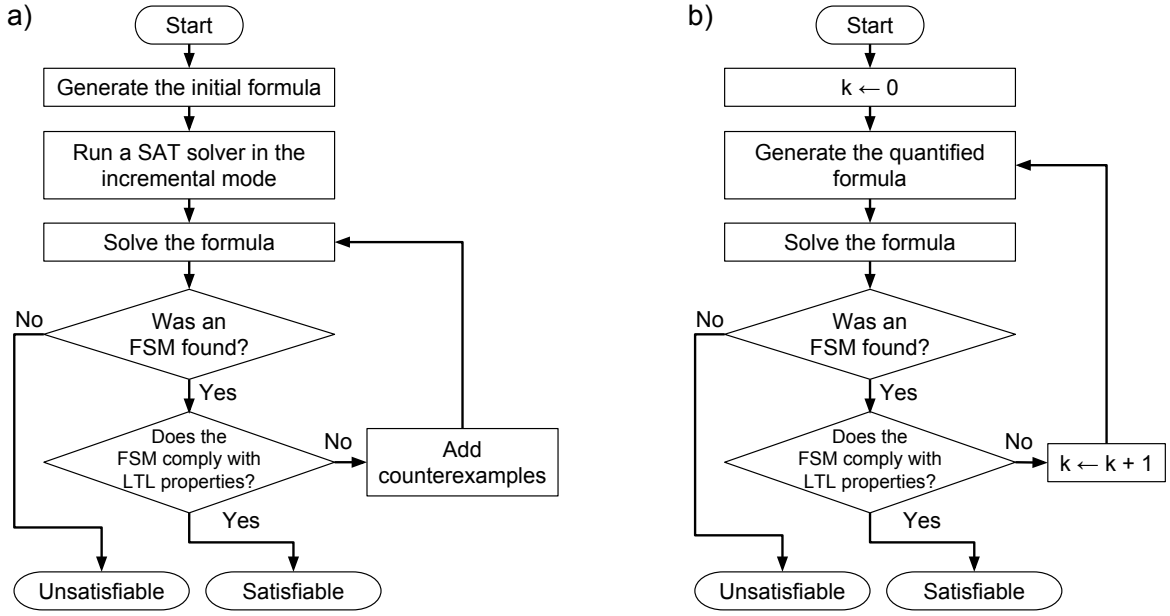

Fig. 6 Flowcharts of the proposed Iterative SAT-based (a) and the QSAT-based approaches (b).

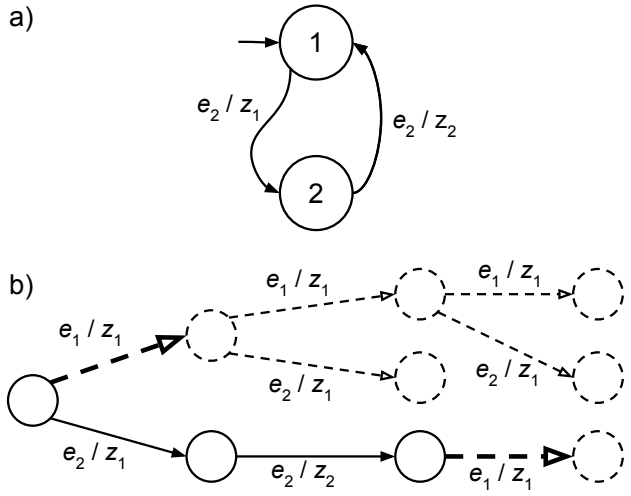

Fig. 7 An example of an FSM under construction (a) and a scenario tree (b) with a frontier (shown by bold dashed arrows). The frontier consists of two occurrences of the same scenario element (e_1, z_1).

its Kripke structure is included into the path sets of augmentations of A. This allows to limit the search: if verification fails, then it will fail for all augmentations of the current FSM. If A complies with scenarios and is verified, the procedure recursively executes itself for A.

If the frontier is empty, we ensure FSM completeness with the same procedure as the one applied in the Iterative SAT solution, or we just return the found answer in the case of incomplete FSM identification.

Algorithm 3 illustrates the solution. An additional function `FindNewFrontier` is used, which finds the frontier for the FSM augmented with a new transition, or returns *null*, if the FSM is inconsistent with scenarios. `Backtracking` is the recursive invocation of the algorithm being defined.

```
Data: set of scenarios SC, temporal specification LTL,
      current FSM (initially empty), frontier (initially
      contains first test elements of scenarios in SC)
edge ← any edge in frontier
for destination ∈ 1..|S| do
    if ∃ unvisited FSM's state s < destination then
    break
    source ← the state of FSM from which edge should
      be added
    FSM′ ← FSM ∪ transition(source, edge,
      destination)
    frontier′ ← FindNewFrontier(SC, FSM′, frontier)
    if frontier′ ≠ null ∧ ModelCheck(FSM′, LTL)= ∅
    then
        if frontier′ = ∅ then
            if completeness is required then
                FSM′ ← Complete(FSM′, LTL)
                if FSM′ ≠ null then return FSM′
            else return FSM′
        else
            FSM′ ← Backtracking(SC, LTL, FSM′,
              frontier′)
            if FSM′ ≠ 'UNSATISFIABLE' then
                return FSM′
            end
        end
    end
end
return 'UNSATISFIABLE'
```

Algorithm 3: Backtracking solution.

5.5 Preliminary comparison

In this subsection we present a brief preliminary qualitative comparison of the proposed methods and two other previously mentioned approaches which solve similar problems: the ones presented in [38] and [9]. This comparison is summarized in Table 2.

The row of the table which lists formula sizes deserves some comments. To begin, the part of the Boolean formula expressing the compliance of the FSM with scenarios is $O(l^2|S|)$ (assuming the total length of scenarios $l \geq |S|$). This scenario-related part is present in formulae generated by all three solver-based methods. Next, LTL formula "translations" mentioned in Section 5.2.4 are in the worst case exponential of the formula size. Overcoming this issue with the subterm extraction technique [4] is impractical, since in our case this technique would introduce new universally quantified variables. As we found, this slows down the QSAT solver and further increases the length of the formula produced by the Exponential SAT-based approach. However, in some cases this estimate is polynomial: for example, for the case of a number of LTL formulae of a preassigned size, the Boolean formula length grows linearly with the number of LTL formulae. Finally, the Exponential SAT-based solution produces the formula whose length is exponential not only of the LTL formula size, but also of k, since the number of considered combinations of universal variables is $O(k|S|)$. In the estimations in this paragraph we assumed $|E|$ and $|Z|$ to be constant.

6 Experimental evaluation

This section reports on experimental evaluation of the proposed approaches. It consists of three main parts: the evaluation on case studies from the literature (Section 6.1), the evaluation on randomly generated instances (Section 6.2) and the comparison of the proposed approaches with the known ones (Section 6.3). The experimental evaluation seeks to answer the following research questions:

RQ1: Are the proposed methods practically applicable?
- (a) Are they applicable on FSM synthesis tasks from the literature?
- (b) Is there a potential to use them in industry?

RQ2: To which extent are the proposed methods scalable with respect to the size of the problem?

RQ3: What are the benefits and shortcomings of the proposed methods in comparison with known approaches?

To answer research questions RQ1 (a), RQ2 and RQ3, three groups of experiments are conducted, which are described in Sections 6.1, 6.2 and 6.3, respectively. To answer RQ1 (b), all three experiments will be used. The analysis of the results with respect to the research questions is provided in Section 6.5.

6.1 Experimental evaluation on case studies

This subsection reports on the evaluation of the proposed FSM identification techniques on case study instances. Its purpose is to examine whether the proposed methods are applicable in practice and how one might benefit from identifying minimum FSMs consistent with the given specification.

6.1.1 Case study systems

Several case study instances connected with software model inference were collected from previous works. In the alarm clock example [35], the FSM controlling the alarm clock must be identified. This example is an easy one since the original set of scenarios for the clock example was very comprehensive. Next, in the elevator example [34] the elevator door controlling logic is to be induced. A more complex instance obtained from the repository[2] of the same authors is the ATM example. The remaining examples, adopted from [38], are connected not with industrial automation but with desktop software. They include the problems of model inference for a simple text editor, the JHotDraw framework and the Jakarta Commons CVS client. The properties of the mentioned problem instances and the number of states of reference FSMs from previous studies are summarized in Table 3.

6.1.2 Experiment setup

Since the proposed methods are exact, we were interested in finding a minimum FSM for each instance. We did it by increasing the number of states $|S|$ until the solver found a satisfying assignment. As the starting point for $|S|$ we chose the size of the clique in the consistency graph of the scenario tree found by the greedy *max-clique* algorithm [22], which is a lower bound on $|S|_{\min}$, the optimal number of states. Total execution times (i.e. sums of execution times for each attempted $|S|$) were recorded, and the obtained FSMs were further analyzed. The computation was performed on the *Intel Core i*7-4510U 2.0 GHz CPU on a single core. Each execution series was given a time limit of 48 hours. As

[2] `https://code.google.com/p/gabp/`

Table 2 Qualitative comparison of the proposed and known FSM identification methods. Denotations: ITER – the Iterative SAT-based solution, QSAT – the QSAT-based solution, EXP – the Exponential SAT-based solution, BTR – the Backtracking solution, CMA – CSP+MuACO*sm* [9], SM – the passive state merging approach [38].

Method	Exact	LTL class	Solver type	Formula size
ITER	Yes	Arbitrary	SAT	$O\left(l^2\lvert S\rvert\right)$
QSAT	Yes	Arbitrary	QSAT	$O\left(l^2\lvert S\rvert + 2^{\lvert\mathrm{LTL}\rvert}\right)$
EXP	Yes	Arbitrary	SAT	$O\left(l^2\lvert S\rvert + 2^{\lvert\mathrm{LTL}\rvert+k\lvert S\rvert}\right)$
BTR	Yes	Arbitrary	–	–
CMA	No	Arbitrary	CSP	$O\left(l^2 + l\lvert S\rvert\right)$
SM	No	Safety	–	–

Table 3 Case study systems and properties which measure their complexity.

Instance	Events	Actions	Scenarios	LTL properties	Original $\lvert S\rvert$
Alarm clock	16	7	38	11	3
Elevator	5	3	9	13	5
ATM	14	13	37	30	unknown
Text editor	5	0	13	5	4
JHotDraw	6	0	27	10	7
CVS client	16	0	12	29	18

for the memory limit, it was roughly equal to 8GB, the amount of memory installed on the used computer.

Boolean formulae in the context-free representation obtained as the result of translations described in Sections 5.2 and 5.3 were transformed to the DIMACS format by *limboole*[3]. As for Boolean formulae in Section 5.1, they were sufficiently simple to generate them straightly in the DIMACS format.

To solve SAT instances, we used two solvers, both of which were highly ranked in the SAT Competition 2016[4]. In the Iterative SAT-based approach, *cryptominisat*[5], the winner of the incremental track, was applied. The Exponential SAT-based approach, in which incremental SAT solving was not used, employed *lingeling*[6]. This solver won the third prize in the main track of the competition, but showed excellent performance on unsatisfiable instances, which is important for finding the minimum FSM.

To solve the quantified formula (3) in the QSAT-based solution, we applied the *DepQBF* QSAT solver [29], which has shown the best performance among several QSAT solvers tried. Its input format is QDIMACS, an extension of DIMACS, but *limboole* can still be used to produce such inputs: it mostly remains to append quantifiers to its output.

6.1.3 Results

The results of the evaluation are summarized in Table 4, which outlines whether the FSM identification methods succeeded, how much time they consumed and how large the FSMs they produced were.

Unfortunately, the QSAT-based method, which has more complex theory behind it compared to other methods, failed to solve almost all instances, being successful only on the alarm clock and the text editor examples. Its substitute, the Exponential SAT-based approach, performed better, but still violated memory limits on three instances. The Backtracking and the Iterative SAT-based methods were much more successful, with the performance of the former being biased towards smaller instances. Clearly, the leader in this comparison is the Iterative SAT-based method, but even it was unable to solve the CVS client instance. More precisely, it was able to find the solution fast (in only 35 seconds) for $|S| = |S|_{\min} = 18$, but the proofs of unsatisfiability of instances with smaller $|S|$ did not terminate within the time limit.

In general, the major fraction of time is spent by the Iterative SAT-based and the Backtracking methods to prove that solutions with $|S| < |S|_{\min}$ (in particular, with $|S| = |S|_{\min} - 1$) do not exist. For the Iterative SAT-based method, long delays were caused exclusively by the last iteration (after adding all counterexamples), when the SAT solver was faced with unsatisfiable problem instances. This is the cost of solving the problem precisely. If the proper $|S|$ is known in advance, their run times will be much smaller.

For the presented case study instances, the target FSMs were known in advance (except the ATM example) and were proved by our methods to be minimum (except the CVS client example) by showing that there is no solution with fewer states. The minimality of a solution implies that all its states are obligatory and

[3] `http://fmv.jku.at/limboole/`
[4] `http://baldur.iti.kit.edu/sat-competition-2016/`
[5] `http://www.msoos.org/cryptominisat4/`
[6] `http://fmv.jku.at/lingeling/`

Table 4 Execution times (in seconds) of the proposed methods on case study systems and the state numbers of identified FSMs. ML and TL stand for a failure due to the memory limit (8GB) and the time limit (48 hours) respectively.

Instance	ITER	QSAT	EXP	BTR	Minimum $\|S\|$
Alarm clock	0.7	77.0	1.6	0.4	3
Elevator	2.2	TL	5.7	1.3	5
ATM	19.0	TL	ML	481.5	9
Text editor	1.3	14433.3	8.9	1.0	4
JHotDraw	3.6	TL	ML	27.9	7
CVS client	TL	TL	ML	TL	18

meaningful, which can aid the comprehension of the system subject to model inference. We provide the previously unknown minimum FSM for the ATM instance in Fig. 8.

6.2 Experimental evaluation on random instances

This subsection reports on the evaluation of the proposed FSM construction techniques on a larger number of random problem instances of increasing complexity. While in the previous subsection almost all instances (except the alarm clock) specified incomplete FSM identification, in this part of the evaluation both types of instances are equally present.

6.2.1 Instance preparation

To evaluate the proposed FSM identification methods on a larger sample and on a more diverse set of LTL properties, we prepared problem instances based on random FSMs. For each $3 \le |S| \le 12$ and both subtypes of the problem (either complete or incomplete FSM identification), 50 FSMs were randomly generated. Each generated FSM had four events and four actions, and the lengths of output sequences on each transition were sampled uniformly at random from the set $\{0, ..., 4\}$. Incomplete FSMs possessed 50% of possible transitions. Following that, four random LTL formulae were generated for each FSM with the *randltl* tool [14]. To ensure that the formulae were sufficiently difficult, we discarded the ones satisfied for more than 5 of 10 other random FSMs generated with the same parameters.

Then we randomly generated test scenarios by executing the generated FSMs choosing on each cycle a random outgoing transition from the current state. For complete FSMs, each transition was prohibited to be executed with the probability of 50% while generating scenarios. The rationale behind this was to test the methods on instances for which the use of scenarios only is not sufficient to generate a proper FSM. For each FSM we created 10 scenarios with a total length of $50|S|$.

While generating instances for the problem of incomplete FSM identification, we accepted only *hard* instances: the ones for which the method from [35] is insufficient, i.e. it produces an FSM not compliant with the formulae. If the instance was not hard, it was replaced with a new one until the new instance appeared to be hard. A similar approach was applied in [9]. Thus, we prepared satisfiable problem instances, for which a solution is guaranteed to exist. The entire instance generation procedure is outlined in Algorithm 4.

for $|S| \in 3..12$, complete $\in \{true, false\}$, $i \in 1..50$ **do**
  repeat
    $\mathsf{FSM}_i \leftarrow$ `RandomFSM`$(|S|, |E| = 4, |Z| = 4,$ complete$)$
    $\mathsf{SC}_i \leftarrow$ `RandomScenarios`$(\mathsf{FSM}_i,$ number $= 10,$ $l = 50|S|,$ complete$)$
    for $j \in 1..4$ **do**
      repeat
        $\mathsf{LTL}_{i,j} \leftarrow$ `RandomLTL`(FSM_i)
        for $k \in 1..10$ **do**
          $\mathsf{CheckFSM}_k \leftarrow$ `RandomFSM`$(|S|,$ $|E| = 4, |Z| = 4,$ complete$)$
        end
      until $\mathsf{LTL}_{i,j}$ is true for less than 6 FSMs out of $\mathsf{CheckFSM}_{1..10}$
    end
  until the method from [35] run on FSM_i and SC_i produces an FSM incompliant with $\mathsf{LTL}_{i,1..4}$
end
return $(\mathsf{FSM}_{1..50}, \mathsf{SC}_{1..50}, \mathsf{LTL}_{1..50,1..4})$

Algorithm 4: Instance generation procedure.

6.2.2 Experiment setup

Similarly to the experiments with the case study systems, we again aimed to find a minimum FSM for each instance and thus applied the same procedure of iterating over $|S|$ as before. Note that the number of states $|S|_{\min}$ of a minimum FSM can not exceed $|S|_{\max}$ of the FSM from which the scenarios were generated.

While evaluating the methods, we wished to determine how many instances could be solved by the methods within a reasonable time limit. The time span given

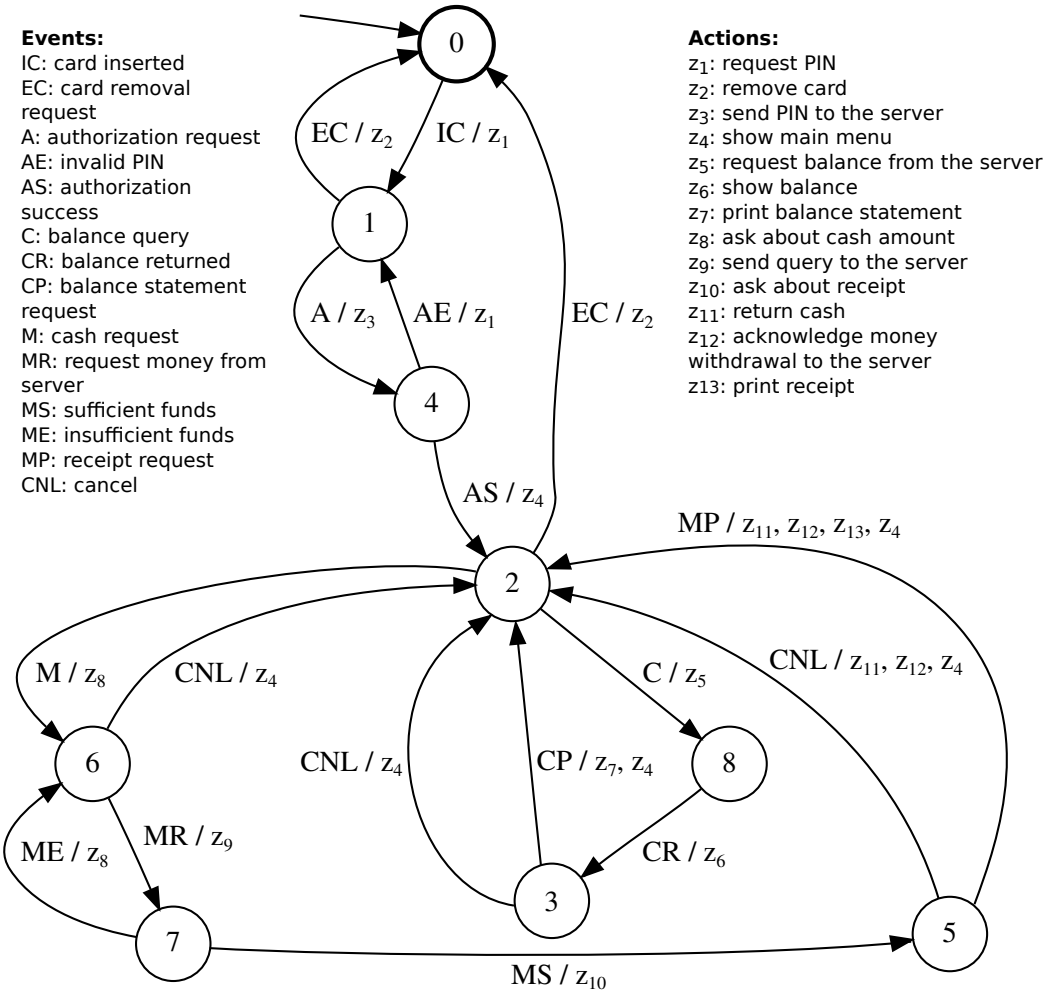

Fig. 8 Generated FSM for the ATM problem.

for each method to solve each $|S|$-iteration of each instance was chosen to be 5 minutes. If either of the iterations failed, then the whole run for the instance was regarded as failed. Except violating the time limit, the Exponential SAT-based method could fail due to the lack of memory for its operation.

For each problem subtype (complete and incomplete FSM identification), $3 \leq |S|_{\max} \leq 12$ and each FSM construction algorithm, 50 executions were performed for target FSMs with these properties. The number of solved instances was recorded for each set of executions.

6.2.3 Results

Table 5 shows the results of FSM identification method executions in terms of numbers of solved instances. It is clearly visible from the table that the QSAT-based approach again performed almost inadequate. Following that, the Exponential SAT-based solution solved the majority of instances being unsuccessful only on incomplete instances with large $|S|_{\max}$. Finally, the leaders among the strategies were the Backtracking and the Iterative approaches with the latter being better on large $|S|_{\max}$. The obtained results are consistent with the ones obtained in Section 6.1.

We note that the complete instance set was generally easier for all methods to solve. This was due to the fact that the minimum numbers $|S|_{\min}$ for this instance set were often less than the ones of the incomplete set. For example, for $|S|_{\max} = 10$, FSMs had average $|S|_{\min}$ of 5.8 for complete instances and 8.5 for incomplete ones.

In addition to performance measurements, we examined in more detail the execution of the Iterative SAT-based approach, which is the most appealing one according on the results. In Fig. 9, its execution times are compared with iteration numbers (the data for all instances is combined). From this plot it is clearly visible that a considerable fraction of instances (more precisely, 96.2%) was solved within only 10 seconds. The vertical

Table 5 Numbers of problem instances (out of 50) solved by FSM identification method executions arranged by the type of problem (complete and incomplete identification) and the number of states $|S|_{\max}$.

$\|S\|_{\max}$	Complete FSM identification				Incomplete FSM identification			
	ITER	QSAT	EXP	BTR	ITER	QSAT	EXP	BTR
3	50	47	50	50	50	42	50	50
4	50	34	50	50	50	19	50	50
5	50	33	50	50	50	12	50	50
6	50	23	49	50	50	6	45	50
7	50	20	47	50	50	5	37	50
8	50	20	46	47	50	4	28	48
9	50	11	48	44	50	1	17	42
10	50	10	46	41	46	0	19	28
11	50	9	41	38	44	0	21	20
12	50	15	44	30	43	0	18	7

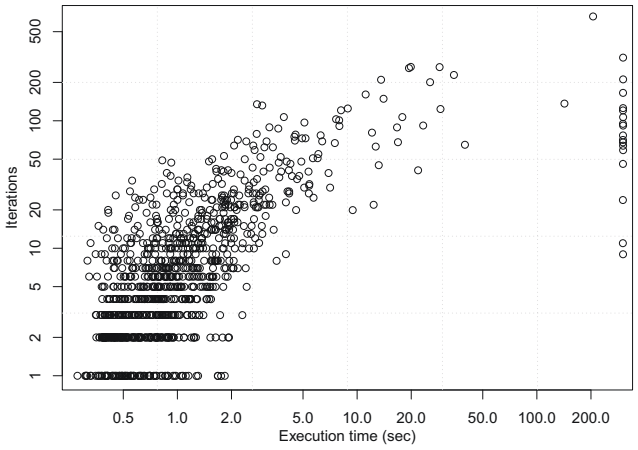

Fig. 9 Execution time vs. iteration number for the Iterative SAT-based method.

row of point on the right corresponds to unsolved instances (1.7%). This plot additionally provides a glance at instance complexity: for example, 59.7% of instances required at least five iterations to be solved. Since the instances were filtered based on complexity during their generation, the number of instances solved in one iteration is small (8.6%), which implies that a method not supporting LTL properties, like the one from [35], would not perform adequately on our dataset.

6.3 Comparison with inexact methods

This subsection reports on the comparison of the proposed FSM identification techniques with three inexact methods: the CSP+MuACO*sm* algorithm [9], the state merging approach [38], and the *Unbeast* tool [16] based on bounded LTL synthesis. Since the performance of the QSAT-based method has been shown to be inappropriate, this method was excluded from experiments in this subsection.

6.3.1 Comparison with the metaheuristic FSM identification method

We compared the Iterative SAT-based, the Backtracking and the Exponential SAT solutions with the CSP +MuACO*sm* method introduced in [9] to infer FSMs with no completeness requirement. In [9], the evaluation of the method for three total scenario lengths ($l \in \{50|S|, 100|S|, 200|S|\}$) and six different state numbers ($5 \le |S| \le 10$) is reported. For each combination of these parameters, 50 instances were prepared, and only hard ones were retained. This test data was obtained from [9] with precise values of median method execution times (these values were shown in [9] on charts) and with annotations of which instances are hard.

Since the FSM model of [9] uses input variables, we transformed them into events as described in Section 4. Problem instances considered in [9] have two variables and two events. Hence, we end up with $|E| = 8$ new events. Such a transformation also duplicates some edges in the scenario tree, since there might be several satisfying assignments for a Boolean guard condition of a scenario element. Thus, we had to modify the Backtracking approach to handle multiple edges: each edge group originating from a common scenario element is always added or removed simultaneously.

Table 6 presents the comparison of median execution times of the proposed methods and the CSP+ MuACO*sm* algorithm on this data. This time, since CSP+MuACO*sm* is not an exact method, we did not try to minimize $|S|$ and executed the methods for the maximum possible numbers of states. Note that [9] used the *AMD Phenom II x*4 955 3.2 GHz CPU for experiments. As for the proposed methods, which were still executed on an *Intel Core i*7-4510U 2.0 GHz CPU, they were now given 15 minutes for $l = 50|S|$, 30 minutes for $l = 100|S|$ and 60 minutes for $l = 200|S|$, since solver execution time was affected by the length of scenarios

Table 6 Median execution times (in seconds) of the proposed methods and CSP+MuACO*sm* (designated as CMA) on the instance sets from [9]. Asterisks (*) show lower bounds on medians in the cases when less than 50% of runs were finished within their time limits.

$\|S\|$	$l = 50\|S\|$				$l = 100\|S\|$				$l = 200\|S\|$			
	ITER	EXP	BTR	CMA	ITER	EXP	BTR	CMA	ITER	EXP	BTR	CMA
5	0.7	3.5	0.8	25.5	1.1	8.1	1.1	27.5	2.1	20.1	0.9	76.5
6	0.9	6.1	3.8	32.0	1.5	15.6	10.2	19.0	2.8	40.2	4.1	41.5
7	1.0	9.4	37.4	68.0	1.8	28.5	26.6	103.0	4.0	91.1	16.1	169.5
8	1.2	10.8	900*	410.0	2.5	27.9	465.1	191.0	5.3	61.2	3205.6	462.0
9	1.4	39.1	900*	213.0	3.0	41.9	1800*	406.5	8.2	169.6	3600*	299.5
10	2.4	15.9	900*	958.0	3.9	40.9	1800*	2443.0	9.9	408.8	3600*	4025.5

and we wished the methods to solve the majority of instances.

The Iterative approach solved all the instances, unlike the other methods, and, according to the table, its performance was also the best. Next, the Exponential SAT-based approach failed to solve around 16% of instances (92 of 580), mostly due to reaching the memory limit of 8GB (this can be explained by the fact that the length of the formula to be solved grows exponentially with k). Still, this did not influence the execution time medians, and this approach outperformed CSP+MuACO*sm* as well. The superiority of the Iterative and the Exponential SAT-based approaches in comparison with CSP+MuACO*sm* is most evident for $|S| = 10$. Finally, the Backtracking approach was less successful: starting from $|S| = 8$, it generally lacked time to solve the instances. Nonetheless, this approach has the benefit of being quite indifferent to scenario length and consumes little memory.

6.3.2 Comparison with counterexample-based state merging

A more narrow problem than the one considered in this paper is the problem of finite-state model identification stated in [38]. Their task was to construct an FSM whose transitions are marked only with events using a number of software execution traces (event sequences) and temporal properties, represented as LTL safety formulae. An LTL formula is a safety formula, if every counterexample to it has a finite prefix such that every its continuation is a counterexample – such finite counterexamples were previously mentioned in Section 5.1.3. Informally speaking, such formulae ensure that some undesired conditions never become true.

The passive approach from [38] (this paper also suggested the active approach, in which the user is inquired during FSM synthesis) was implemented independently, and instead of the Spin[7] verifier we used the one mentioned previously. Following [38], we chose Blue Fringe [27] as the state merging method.

[7] `http://spinroot.com/`

Table 7 Execution times (in seconds) of the proposed methods and the passive state merging (SM) approach [38] on three instances adopted from the same work. ML and TL stand for a failure due to the memory limit (8GB) and the time limit (48 hours) respectively.

Instance	$\|S\|_{\min}$	ITER	EXP	BTR	SM
Text editor	4	1.3	8.9	1.0	0.4
JHotDraw	7	3.6	ML	27.9	0.8
CVS client	18	TL	ML	TL	36.2

As the data for comparison, we adopted the examples from [38] with few changes. In particular, we ensured that the data allowed the compared methods to produce the desired FSMs, if they were able to produce any FSMs at all. The data used in this comparison has already been applied in Section 6.1 for the case study evaluation. It includes the instances in which the finite-state models of a simple text editor, the JHotDraw drawing framework and the Jakarta Commons CVS client are to be learnt.

The results of the method comparison are presented in Table 7, which partially duplicates the data from Table 4 in Section 6.1. The performance of the proposed methods on the considered instances has been discussed previously. As for state merging, it appeared to be much faster than the exact methods. Nonetheless, although it identified target minimum FSMs, in general it does not guarantee the minimality of its solutions: it has no means of proving that its answers are optimal with respect to the number of states. Moreover, as far as we know, there are no ways of applying it in the cases of FSMs with actions or non-safety (i.e. general) LTL properties.

To demonstrate that state merging may not find a minimum FSM, we randomly generated 100 FSMs without output actions, each with 5 states, 10 events and 25% of possible transitions. To ensure that LTL formulae were safety ones, instead of using *randltl* we generated them according to several simple templates. One of such templates was $\mathbf{G}(e \rightarrow \mathbf{X}(e_1 \vee ... \vee e_k))$, where e is an event and $e_1, ..., e_k$ are the events which can occur on a transition which follows a transition marked with

e. For 12 of 100 FSMs state merging failed to find the minimum answer.

Similarly to the approach from [38], our Iterative, Backtracking and Exponential SAT-based approaches can be modified to be active: while at least two FSMs can be identified from the input data, it may ask the user to confirm or reject an execution trace of one of such FSMs to obtain more data. These approaches can also be applied to find all possible solutions of the problem. To do this for solver-based methods, after a solution is found, a constraint prohibiting the found FSM is appended to the formula, and the solver is restarted. Instead of this, the Backtracking approach can simply continue its search.

6.3.3 Comparison with LTL synthesis

Recent advances in LTL synthesis resulted in software tools called *Lily* [26], *Acacia* [18] and *Unbeast* [16]. Among them, we used *Unbeast* for comparison since it is more recent and is claimed to be superior over others [17]. Another tool *G4LTL-ST* [7] is also known, but it is focused solely on program synthesis for PLCs and more rich forms of LTL specification.

Since in the problem of LTL synthesis the specification is given only as LTL properties, we had to encode scenarios in LTL. While doing so we unfortunately lost the ability to distinguish the order of actions produced on the same cycle, which could potentially allow smaller FSMs to satisfy scenarios.

Next, the problem of LTL synthesis requires the construction of complete reactive systems. We tried to express incompleteness using a dummy action for transitions which are to be removed after the synthesis and exempting paths which included such dummy transitions (i.e. actually impossible paths) from the need to match LTL properties. Unfortunately, this led to the problem of states with no proper outgoing transitions; paths leading to such states were also exempted from compliance with temporal specifications. Thus, we report on the results of running *Unbeast* only on instances for complete FSM identification.

Unbeast produces the target system as a game which resembles a Mealy FSM. Using the provided game simulator, it is possible to reconstruct this FSM. After the reconstruction, we also greedily minimized the FSM maintaining the compliance with the specification.

We executed *Unbeast* on complete instances from our instance set described in Section 6.2.1 for $3 \leq |S|_{\max} \leq 6$ (for larger $|S|_{\max}$ the execution time of *Unbeast* was often impractically large). Executions for various $|S|_{\max}$resulted in median run times of 8, 38, 130 and 510 seconds, respectively. As described in Section 6.2.3, the majority of the methods proposed in this paper performed better. Furthermore, the number of states in FSMs produced by *Unbeast* was immense, reaching the median of 103 already for $|S|_{\max} = 3$.

The reason for the weak performance of *Unbeast* seems to be the large size of LTL specifications, which mainly results from scenarios. Shortened scenarios were noticed to cause better performance. We conclude that while *Unbeast* might perform well on natural LTL specifications, the inclusion of scenarios, which are typical for the problem of software model reverse engineering, causes it to perform much worse.

6.4 Threats to validity

All three groups of experiments, which are described above, are potentially prone to validity threats. In Section 6.1, a number of instances from the literature are used to evaluate the proposed methods. This set of instances is small, and hence might be not representative with respect to other software models. This threat has been partially mitigated by selecting case instances from different studies.

Next, the random FSM generation procedure (Algorithm 4) might have produced nonsensical or oversimplified reference models. Unfortunately, the extent to which randomly generated models are similar to software models encountered in practice is problematic to determine, so threat mitigation was limited to instance complexity evaluation. First, the data generation algorithm (Algorithm 4) was equipped with complexity filtering. The numbers of iteration required to solve the remaining instances were provided in Fig. 9 and appear to be satisfactory. Next, a subset of generated models was examined visually, which confirmed that state transition graphs of the generated FSMs were sufficiently complex. Following that, minimum solution sizes were determined during the evaluation. According to Section 6.2.3, the complete instance set was generally easier in terms of this measure; nevertheless, instances with $|S|_{\max} = 12$ were able to distinguish the performance of different methods.

A possible error in the implementations of methods might have made the corresponding execution metrics meaningless. To prevent this, for each solved instances its correspondence with the required specification was ensured. Finally, in Section 6.3.2, we use our own implementation of the state merging method from [38], which might perform differently than in [38]. Nevertheless, the performance of this method is clearly superior to the one of the proposed ones, and the shortcomings of this approach are mainly qualitative and do not depend on the implementation.

6.5 Discussion

It remains to discuss the results from all subsections of the evaluation and finally answer the research questions formulated in the beginning of Section 6. As revealed in Section 6.1.3, the Iterative SAT-based and the Backtracking methods were able to cope with the majority of considered case instances, which answers RQ1 (a). However, it was also found that for large numbers of states the performance of these methods (especially the one of the Iterative SAT-based method) could be much better if the optimal number of states was known in advance. This raises the question whether the precision of solving the problem can be partially traded for performance. This question is discussed in more detail in Section 7.

With respect to scalability, which is questioned in RQ2, the methods are ranked in the following order: the Iterative SAT-based, the Exponential SAT-based, the Backtracking, and the QSAT-based methods. These are the results of both Section 6.2.3 are Section 6.3.1. The first of these methods was able to solve almost all of the hardest instances considered.

As for RQ3, the comparison of the proposed methods with the known ones can be viewed in terms of capabilities and performance. In terms of capabilities, the considered alternative methods lacked some of the features supported by the proposed ones. However, we did not compare our methods with the ones which are able to synthesize richer finite-state models, like in [31, 39], which is obviously an advantage of such methods. The results in terms of performance are quite diverse. Even the most scalable Iterative SAT-based method did not excel state merging, but one should remember that the latter is quite limited in the set of problem instances it can handle (this is also the reason why state merging was not included in the majority of experiments considered in this paper). On the other hand, two of the proposed methods were able to surpass CSP +MuACO*sm* (see Section 6.3.1).

Finally, we must answer RQ1 (b) by determining whether the proposed methods are potentially applicable in industrial software engineering. All empirical evaluations performed in this study assume that the number of FSM states is quite small: it did not exceed 18. Clearly, this number of states is insufficient to represent real-world systems in full detail. Nevertheless, identified models should not be considered as full-scale substitutes of the target systems. Their purpose is to provide a picture of the system's logic for a software engineer, which becomes more clear and concise once the minimality requirement is fulfilled. This picture can serve as a basis either for understanding or reverse-engineering the system. It is also worth noting that FSMs which include 20 or more states are difficult to comprehend. If this number of states is insufficient for the given specification, one might try simplifying it by focusing only on particular aspects of the system. These thoughts, as well as the performed case study evaluation, make us hope that the Iterative SAT-based approach (the proposed approach with the best performance) is potentially applicable in practice, although more research is needed to improve it.

Aside from research questions, we need to explain why the methods obtained the results observed in the evaluation. Despite complex theory behind, the QSAT-based approach is clearly an outsider. This might be due to the low performance of the state-of-the-art QSAT solvers, which will hopefully be improved in the future. Another possible reason is the consideration of only one translation [4] of the BMC problem into SAT. Some other translations can be found in [2].

The SAT-based implementation of the QSAT-based approach, the Exponential SAT-based approach, performed better, but at least two drawbacks still hindered its performance. First, the length of the BMC part of the Boolean formula is exponential of the LTL property length in the worst case. Second, the length of the formula is exponential of k, therefore in our experiments the tractable value of k did not exceed four. This means that only counterexamples with length up to five were taken into account, which is clearly insufficient for large FSMs. When a larger value of k was required, this method consumed too much memory.

Next, the good results of the Backtracking approach were quite surprising given its simplicity. Because of it, it easily solves small instances, but its performance drops fast when the number of states becomes sufficiently high (around 11 in Table 5 and around 8 in Table 6). A possible reason why this approach performs worse in Table 6 compared to Table 5 is the more complex structure of the scenario tree (see Section 6.3.1). The Backtracking approach is also not prone to memory problems, unlike the Exponential SAT-based approach.

Finally, possible reasons for the success of the Iterative SAT-based approach include matching the idea of the method with the efficient incremental solver *cryptominisat*, and a relatively small overhead of encoding counterexamples in the Boolean formula compared to the Exponential SAT-based approach. We must note that the performance of this approach can be much worse on large unsatisfiable problem instances (see Section 6.1.3).

7 Conclusions and future work

We have presented a number of approaches for the exact FSM identification problem from scenarios and temporal properties. This problem arises when one wants to infer or reverse engineer the model of a software system, such as a desktop application or an industrial reactive system (e.g. elevator controller). The approaches were evaluated and compared with existing inexact (i.e. not producing minimum FSMs) solutions both on case study instances from previous research and on randomly generated instances. The ranking of the methods in terms of their performance, from the best to the worst, was determined to be as follows: the Iterative SAT-based, the Exponential SAT-based, the Backtracking, and the QSAT-based approaches.

Unlike the earlier metaheuristic approach [9], the proposed ones support the FSM completeness requirement and are able to prove the unsatisfiability of instances, which can be applied to construct minimum FSMs compliant with the given specification. However, on a more narrow problem of identifying FSMs without actions from scenarios and LTL safety properties, the proposed approaches are outperformed by the one from [38]. As for the comparison with the symbolic bounded LTL synthesis tool [16], it reveals that scenarios, which usually do not cause problems for the presented approaches, make this tool produce large solutions and work slowly.

The performance of the Iterative SAT-based method is encouraging, but supporting larger FSMs would bring it closer to industrial application. An alternative way of applying FSM synthesis in software engineering is sacrificing features, such as precision and full support of LTL, in order to improve performance. If only positive examples are used for learning, then prohibiting invalid model behaviors is only possible by providing an exhaustive set of scenarios. Also, as shown in [38, 28, 3], temporal properties are important in guiding FSM synthesis, thus the entire refusal of LTL specifications would be discouraging. Among known methods, a good option is the method from [38]. However, it supports only the safety subset of LTL, which, for example, does not include the common unbounded response property $\mathbf{G}(x \rightarrow \mathbf{F}\, y)$. As for the methods proposed in this paper, Section 6.1.3 has shown that proving the optimality of the solution can be much more time consuming than finding the solution. Thus, finding the minimum solution approximately may be a proper trade-off between precision and performance.

Another possible future work direction might involve considering other translations from BMC into SAT [2], which may improve the QSAT-based solution. Then, the methods can be brought closer to practice by supporting wider classes of FSMs, like in [31] and [39]. One more interesting idea to try is automatic mining of temporal properties from execution traces [28, 3], which can solve the difficulty of obtaining temporal specifications.

References

1. Alur, R., Martin, M., Raghothaman, M., Stergiou, C., Tripakis, S., Udupa, A.: Synthesizing finite-state protocols from scenarios and requirements. In: Hardware and Software: Verification and Testing, pp. 75–91. Springer (2014)
2. Amla, N., Du, X., Kuehlmann, A., Kurshan, R.P., McMillan, K.L.: An analysis of SAT-based model checking techniques in an industrial environment. In: Correct Hardware Design and Verification Methods, pp. 254–268. Springer (2005)
3. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D.: Leveraging existing instrumentation to automatically infer invariant-constrained models. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 267–277. ACM (2011)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. ADV COMPUT, Elsevier **58**, 117–148 (2003)
5. Bodik, R., Jobstmann, B.: Algorithmic program synthesis: introduction. International Journal on Software Tools for Technology Transfer, Springer **15**(5-6), 397–411 (2013)
6. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. (eds.): Model-based testing of reactive systems: advanced lectures. Lecture Notes in Computer Science, vol. 3472. Springer (2005)
7. Cheng, C.H., Huang, C.H., Ruess, H., Stattelmann, S.: G4LTL-ST: Automatic generation of PLC programs. In: Computer Aided Verification, pp. 541–549. Springer (2014)
8. Chivilikhin, D., Ulyantsev, V.: MuACOsm: a new mutation-based ant colony optimization algorithm for learning finite-state machines. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation (GECCO), pp. 511–518. ACM (2013)
9. Chivilikhin, D., Ulyantsev, V., Shalyto, A.: Combining exact and metaheuristic techniques for learning extended finite-state machines from test scenarios and temporal properties. In: Proceedings of the 13th International Conference on Machine Learning and Applications (ICMLA), pp. 350–355. IEEE (2014)
10. Chongstitvatana, P., Aporntewan, C.: Improving correctness of finite-state machine synthesis from multiple partial input/output sequences. In: Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware, pp. 262–266. IEEE (1999)
11. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering **4**(3), 178–187 (1978)
12. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT press (1999)
13. Dorigo, M., Stützle, T.: Ant colony optimization. MIT Press (2004)
14. Duret-Lutz, A.: Manipulating LTL formulas using Spot 1.0. In: Automated Technology for Verification and Analysis, pp. 442–445. Springer (2013)

15. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science, Elsevier **89**(4), 543–560 (2003)
16. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: P. Abdulla, K. Leino (eds.) Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 6605, pp. 272–275. Springer Berlin Heidelberg (2011)
17. Ehlers, R.: Symbolic bounded synthesis. Formal Methods in System Design, Springer **40**(2), 232–262 (2012)
18. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: Computer Aided Verification, pp. 263–277. Springer (2009)
19. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, pp. 219–234. Springer (2012)
20. Finkbeiner, B., Schewe, S.: Bounded synthesis. International Journal on Software Tools for Technology Transfer, Springer **15**(5-6), 519–539 (2013)
21. Gold, E.M.: Complexity of automaton identification from given data. Information and Control **37**(3), 302–320 (1978)
22. Heule, M.J., Verwer, S.: Exact DFA identification using SAT solvers. In: Grammatical Inference: Theoretical Results and Applications, pp. 66–79. Springer (2010)
23. Heule, M.J., Verwer, S.: Software model synthesis using satisfiability solvers. Empirical Software Engineering, Springer **18**(4), 825–856 (2013)
24. Hölldobler, S., Nguyen, V.H.: On SAT-encodings of the at-most-one constraint. In: Proceedings of the 12th International Workshop on Constraint Modelling and Reformulation, Uppsala, Sweden, pp. 16–20 (2013)
25. Jackson, P.B., Sheridan, D.: A compact linear translation for bounded model checking. Electronic Notes in Theoretical Computer Science **174**(3), 17–30 (2007)
26. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Formal Methods in Computer Aided Design (FMCAD), pp. 117–124. IEEE (2006)
27. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In: Grammatical Inference, pp. 1–12. Springer (1998)
28. Lo, D., Mariani, L., Pezzè, M.: Automatic steering of behavioral model inference. In: 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE), pp. 345–354. ACM (2009)
29. Lonsing, F., Bacchus, F., Biere, A., Egly, U., Seidl, M.: Enhancing search-based QBF solving by dynamic blocked clause elimination. In: Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), Lecture Notes in Computer Science (to appear). Springer (2015)
30. Mitchell, M.: An introduction to genetic algorithms. MIT press (1998)
31. Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., Brun, Y.: Behavioral resource-aware model inference. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE), pp. 19–30. ACM (2014)
32. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, 1977, pp. 46–57. IEEE (1977)
33. Rosner, R.: Modular synthesis of reactive systems. Ph.D. thesis, Weizmann Institute of Science (1992)
34. Tsarev, F., Egorov, K.: Finite state machine induction using genetic algorithm based on testing and model checking. In: 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO), pp. 759–762. ACM (2011)
35. Ulyantsev, V., Tsarev, F.: Extended finite-state machine induction using SAT-solver. In: Proceedings of the 14th IFAC Symposium "Information Control Problems in Manufacturing (INCOM)", pp. 512–517. IFAC (2012)
36. Ulyantsev, V., Zakirzyanov, I., Shalyto, A.: BFS-based symmetry breaking predicates for DFA identification. In: Language and Automata Theory and Applications, pp. 611–622. Springer (2015)
37. Vyatkin, V.: IEC 61499 function blocks for embedded and distributed control systems design, Second ed. Instrumentation Society of America (2012)
38. Walkinshaw, N., Bogdanov, K.: Inferring finite-state models with temporal constraints. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 248–257. IEEE Computer Society (2008)
39. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empirical Software Engineering, Springer **21**(3), 811–853 (2016)