IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 21, 2021, accepted January 11, 2022, date of publication January 14, 2022, date of current version January 27, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3143478

Generating and Employing Witness Automata for ACTLW Formulae

ROK VOGRIN[1], (Member, IEEE), ROBERT MEOLIC[2], (Member, IEEE), AND TATJANA KAPUS[1], (Member, IEEE)

[1]Faculty of Electrical Engineering and Computer Science, University of Maribor, 2000 Maribor, Slovenia
[2]Operato, 2000 Maribor, Slovenia

Corresponding author: Rok Vogrin (rok.vogrin@um.si)

The work of Tatjana Kapus was supported in part by the Slovenian Research Agency (ARRS) under Grant P2-0069.

ABSTRACT When verifying the validity of a formula in a system model by a model checker, a common feature is the generation of a linear witness or counterexample, which is a computation path usually showing a single reason why the formula is valid or, respectively, not. For systems represented with Labeled Transition Systems (LTS) and a subset of ACTLW (Action-based Computation Tree Logic with Unless operator) formulae, a procedure exists for the generation of witness automata, which contain all the interesting finite linear witnesses, thus revealing all the reasons of the validity of a formula. Although this procedure uses a symbolic representation of LTSs, transitions of a given LTS are traversed one by one. In this paper, we propose a procedure which exploits the symbolic representation efficiently to traverse several transitions at once. We evaluate the procedure on models of a communication protocol from industry and a biological system. The results show it to be at least several times faster than the former one. Witness automata were first introduced to allow for compositional generation of test sequences. We propose two more possible uses. One is for the detection of multiple errors in a model by exploring the witness automaton for a formula, instead of only one, which is usually the case with a single witness. The other one is for the detection of previously unknown system properties. As witness automata can be rather large, we show how some existing tools could help in examining them through visualization and simulation.

INDEX TERMS Automata, formal verification, logic, model checking.

I. INTRODUCTION

Model checking is an automated technique for verifying whether the behavior of a finite-state system model has a specified property or not [1]. A Labeled Transition System (LTS) can, for example, be used to model the system, and the property can be specified with Action-based Computation Tree Logic with Unless operator (ACTLW) [2]. Like ACTL (Action-based Computation Tree Logic) [3], ACTLW is a propositional action-based branching-time temporal logic interpreted over LTSs [2], thus serving to describe the occurrence of actions rather than the validity of atomic propositions over time as with Computation Tree Logic (CTL) [1]. Despite the name, ACTL is not a straightforward action-based ''version'' of CTL. That is why ACTLW has been introduced. It can express all the properties expressible in ACTL, but, nevertheless, in contrast to ACTL, it has model checking algorithms and formulae patterns analogous to CTL [2].

A common feature of model checking is the generation of a linear witness or counterexample for the formula being verified. In the case of the action-based logics, a linear witness or counterexample is a sequence of actions. A linear witness (respectively, counterexample) usually shows a single reason why the formula is valid (respectively, invalid) in the system. In [4], witness automata are introduced for a subset of ACTL formulae. A witness automaton contains all the interesting finite linear witnesses for a formula, thus revealing all the reasons of its validity. It is shown how to employ witness automata to generate, in a compositional way, a test sequence to cover a branch in a system consisting of a chain of modules. Later on, an algorithm has been developed and implemented for witness automata generation for ACTL formulae [5], and, subsequently, adapted for ACTLW formulae [6]. The algorithm for both kinds of formulae relies on the information on the sets of states of the LTS satisfying the subformulae

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

of the given formula, which can be obtained by preliminary model checking. Guided by the structure of the formula and this information, it visits the states of the LTS and traverses transitions from them one by one, by a depth-first search by recursion.

In order to enable formal verification of finite-state systems with larger state spaces, symbolic methods using Binary Decision Diagrams (BDD) [7] have been introduced [8]. The use of these methods can be advantageous in two ways [9]. One is that the sets of states and transitions are represented implicitly with logical functions and these with BDDs, which, in many practical cases, have much smaller space requirements than explicit set representation by listing all their elements. The other is that the symbolic representation allows for processing a whole set of states or transitions at once, which can be more time-efficient than handling each element separately, i.e., in an enumerative way.

The existing algorithm for the generation of witness automata has been implemented as an extension of a symbolic model checker, so that the sets of states satisfying the subformulae can be generated efficiently. The main motivation for the present paper was the fact that, although the implementation of this algorithm uses symbolic representation for LTSs in the form of BDDs and BDD-based functions for navigating the LTS after the symbolic model checking, it cannot take advantage of the symbolic representation to traverse several transitions at once, because the depth-first search is inherently enumerative.

The main contribution of this paper is the generation of witness automata for ACTLW formulae by using symbolic methods based on BDDs which avoids the enumerative traversal. Basically, it consists of two steps. First, all interesting potential witnesses for an ACTLW formula are represented in the form of a symbolically represented formula automaton. Second, a witness automaton is obtained by performing a synchronous product between the formula automaton and the LTS. The advantage of this approach is that the synchronous product can be implemented symbolically in such a way that several transitions of the LTS are traversed at once in a breadth-first manner. The evaluation results show it to have significantly lower witness automata generation times than the old one.

Another contribution of this paper are two more proposals on witness automata employment in addition to their use for the compositional test sequence generation. The first one is to use them to detect multiple errors in a model by exploring a witness automaton, instead of only one, which is usually the case with linear witnesses. Note that a counterexample for a formula is a witness for its negation, and vice versa. There can be multiple counterexamples for a formula, revealing single but possibly different errors in the model. Model checking of a formula returns only one counterexample. To detect multiple errors, model checking of the formula has to be performed several times. Every time, the currently revealed error has to be eliminated because, otherwise, the same counterexample would be generated next time [10]. In our case, a single check of the negation of the formula can give a witness automaton which contains all the counterexamples and, thus, reveals all the errors. The second proposal is to use witness automata to detect previously unknown system properties, which can, for example, be useful for the analyses of biological systems. As witness automata can be rather large, we also show how some existing tools can help in examining them through visualization and simulation.

For the evaluation of the new procedure for the generation of witness automata and illustration of the two kinds of applications, we use a model of the Bounded Retransmission Protocol (BRP), which is a well-known industrial benchmark case study from the verification literature (e.g., [11]–[15]), and a model of the lactose operon regulatory biological system, which is one of the classic examples used for testing formal methods in the area of computational biology (e.g., [16]–[18]).

The rest of this paper is organized as follows. The next section reviews related work. Section III describes the means of system modeling and property specification. Section IV defines finite linear witnesses. In Section V, the definition of witness automata is given and the procedure of their generation described. Section VI briefly describes the implementation of the procedure. In Section VII, we present the two kinds of its application and the evaluation results. Section VIII discusses the work presented in this paper and possible future work. The paper is concluded in Section IX.

II. RELATED WORK

The proposed process of witness automata generation is similar to the verification of safety properties with automata [1]. In both cases we seek the intersection between an automaton that describes a system and an automaton of the property which is a negation of a safety property. The key difference between the two is that, in the case of verification, the main objective is to determine whether the intersection is empty, which indicates that the system satisfies the safety property, and the other way around if it is nonempty. In our work, we employ a nonempty intersection further to obtain a witness automaton containing paths which explain how the property expressed with the automaton is present in the system, or, differently speaking, why the safety property is not satisfied. Another difference is that property automata are usually employed to express a state-based property, typically specified with a state-based logic, such as LTL (Linear-time Temporal Logic), whereas, in this paper, a property automaton represents sequences of actions instead of states. Moreover, our property automata, at least in part, follow the desire not to recognize all the sequences satisfying a property, but only the ''interesting'' ones, thus giving witness automata which include only so-called viable witnesses (e.g., [5]).

Since the invention of symbolic model checking [8], there has been much work that used and implemented symbolic methods in model checking and formal verification [19]–[21]. Most of the work in this field combines symbolic model checking with BDDs [22]–[24]. The terms counterexample

and witness were first used in combination with symbolic model checking in [25]. Whereas quite a lot of work has been done regarding counterexamples [26], [27], witnesses, and, even more so, witness automata, are still a rather unexplored topic.

In the context of system correctness assurance, besides the already mentioned work on witness automata as defined in this paper, we know only of the work on witness automata related to software verification (e.g., [28], [29]). The authors introduced witness automaton as a structure for storing, for example, sequences of program line numbers and values of variables found by a verification tool to lead to an error in a program in order to apply them to the program by using another tool, for example, to check if the former worked properly.

In [10], an algorithm is proposed which returns, using our terminology, several counterexamples for a Simulink model and a property specified in the same language. Those counterexamples are, in fact, sequences of operations (''paths'') in the model which cause the violation of the property, whereas a counterexample in that paper is defined as a sequence of inputs which lead to a wrong output. The algorithm searches for these counterexamples one by one by calling a model checker. Each time it finds a new counterexample, it finds the affected paths and calls the model checker again by directing it away from them.

III. PRELIMINARIES

A. ACTION-BASED COMPUTATION TREE LOGIC WITH UNLESS OPERATOR

ACTLW is a propositional action-based branching-time logic interpreted over LTSs [2]. An LTS $\mathcal{M}$ is a 4-tuple $\mathcal{M} = (S, Act_\tau, D, s_{init})$, where S is a nonempty set of states, Act_τ is a set of actions including the internal action τ, $D \subseteq S \times Act_\tau \times S$ is a transition relation, and $s_{init} \in S$ is an initial state. With $(r, a, s) \in D$ we denote that the system can transit from state r to state s by executing action a. State s is called a successor of state r. For $A \subseteq Act_\tau$ let $D_A(r)$ denote the set of successors of state r which are reachable from r by a transition labeled with an action from A. A transition of the form (r, τ, s) is useful to represent the internal operation of the system. Set Act_τ can also contain external actions. The latter are divided into input actions, labeled with ''?'', and output ones, labeled with ''!''. For example, $a!$ is an output action, whereas $a?$ is an input one. An LTS $\mathcal{M}$ is finite if and only if S and Act_τ are finite.

A path in an LTS is a finite or infinite sequence of states and actions $s_0, a_1, s_1, a_2, s_2, \ldots$ such that it starts in state s_0, and for each pair of consecutive states s_i and s_{i+1}, $(s_i, a_i, s_{i+1}) \in D$. We denote the path as π and the sequence of actions on the path as $act(\pi)$. A finite path ends in a state. We denote a state on the path as $\pi(i)$, where $\pi(0)$ is the first state and $\pi(i+1)$ is the successor of $\pi(i)$. The length of a path π, denoted $len(\pi)$, is the number of states along it. The length of an infinite path is ω. A fullpath is a path that ends in a state with no successors or an infinite path [17]. A state or an action is reachable in an LTS if and only if it is on a path starting in its initial state.

ACTLW syntax for a set of actions Act_τ consists of path quantifiers **EE** (*a path exists*) and **AA** (*for all paths*), temporal operators **U** (*until*) and **W** (*unless*), and actions $\alpha \in Act_\tau$, including internal action τ, which can be a part of the action formulae. The syntax includes the Boolean constant *true*, as well as disjunction ($\vee$) and negation ($\neg$) operators. Labels χ, φ, γ in the following definition represent the so-called action, state and path formulae, respectively:

$$\begin{aligned} \chi &::= \alpha \mid \neg\chi \mid \chi \vee \chi \\ \varphi &::= true \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{EE}\ \gamma \mid \mathbf{AA}\ \gamma \\ \gamma &::= [\{\chi\}\varphi\ \mathbf{U}\ \{\chi\}\varphi] \mid [\{\chi\}\varphi\ \mathbf{W}\ \{\chi\}\varphi] \end{aligned} \tag{1}$$

The semantics of the state formulae, hereafter also called ACTLW formulae, is defined inductively with the rules given in Fig. 1, where $a \in Act_\tau$ and $/\chi/ = \{\alpha \mid \alpha \models \chi\}$. With the expression $a \models \chi$ we say that action a satisfies action formula χ, whereas with $s \models_{\mathcal{M}} \varphi$ we denote that ACTLW formula φ is valid in state s of LTS $\mathcal{M}$. To express that path formula γ is valid on fullpath π of LTS $\mathcal{M}$ we write $\pi \models_{\mathcal{M}} \gamma$. With the ACTLW syntax we can derive additional operators and abbreviations. Instead of action formula $\alpha \vee \neg\alpha$ for any action $\alpha \in Act_\tau$, we write *true*. For action or state formula $\neg true$ we write shortly *false*. For action and state formulae, all the Boolean operators can be defined in the usual way. The state s, such that $s \models_{\mathcal{M}} \varphi$ holds, is called a φ-state. Similarly, we call a transition (r, a, s) for which $a \models \chi$ holds a χ-transition. If (r, a, s) is a χ-transition and s is a φ-state, we denote this transition as (χ, φ)-transition. Finally, ACTLW formula φ is valid in LTS $\mathcal{M}$ ($\mathcal{M} \models \varphi$) if and only if $s_{init} \models_{\mathcal{M}} \varphi$ holds [2].

$$\begin{aligned}
a &\models \alpha, && \text{iff } a = \alpha; \\
a &\models \neg\chi, && \text{iff } a \not\models \chi; \\
a &\models \chi \vee \chi', && \text{iff } a \models \chi \text{ or } a \models \chi'; \\
s &\models_{\mathcal{M}} true && \text{always}; \\
s &\models_{\mathcal{M}} \neg\varphi, && \text{iff } s \not\models_{\mathcal{M}} \varphi; \\
s &\models_{\mathcal{M}} \varphi \vee \varphi', && \text{iff } s \models_{\mathcal{M}} \varphi \text{ or } s \models_{\mathcal{M}} \varphi'; \\
s &\models_{\mathcal{M}} \mathbf{EE}\ \gamma, && \text{iff there is a fullpath } \pi \text{ in } \mathcal{M}, \\
& && \text{such that } \pi(0) = s \text{ and } \pi \models_{\mathcal{M}} \gamma; \\
s &\models_{\mathcal{M}} \mathbf{AA}\ \gamma, && \text{if for all fullpaths } \pi \text{ in } \mathcal{M}, \\
& && \pi(0) = s \Longrightarrow \pi \models_{\mathcal{M}} \gamma \text{ holds}; \\
\pi &\models_{\mathcal{M}} [\{\chi\}\varphi\ \mathbf{U}\ \{\chi'\}\varphi'], && \text{iff } len(\pi) > 1 \text{ and there exists } 1 \le i < len(\pi) \\
& && \text{such that } \pi(i) \models_{\mathcal{M}} \varphi' \text{ and } \pi(i) \in D_{/\chi'/}(\pi(i-1)) \\
& && \text{and for each } 1 \le j \le i-1, \\
& && \pi(j) \models_{\mathcal{M}} \varphi \wedge \pi(j) \in D_{/\chi/}(\pi(j-1)) \text{ holds}; \\
\pi &\models_{\mathcal{M}} [\{\chi\}\varphi\ \mathbf{W}\ \{\chi'\}\varphi'], && \text{iff } \pi \models_{\mathcal{M}} [\{\chi\}\varphi\ \mathbf{U}\ \{\chi'\}\varphi'] \\
& && \text{or for each } 1 \le i < len(\pi), \\
& && \pi(i) \models_{\mathcal{M}} \varphi \text{ and } \pi(i) \in D_{/\chi/}(\pi(i-1)) \text{ holds}.
\end{aligned}$$

FIGURE 1. Definition of the meaning of ACTLW operators.

From Fig. 1 it can be seen that there are, basically, four kinds of ACTLW formulae. They are obtained by combining operators **EE** or **AA** and **U** or **W**, respectively. With the path quantifiers, we express the validity of a path formula for either one or multiple paths, whereas the temporal operators express the validity of action and state formulae on an individual path. ACTLW formula $\mathbf{EE}[\{\chi\}\varphi\ \mathbf{U}\ \{\chi'\}\varphi']$ is said to define an

ACTLW operator **EEU**. The same applies to the other three kinds of ACTLW formulae, thus, altogether, giving operators **EEU**, **EEW**, **AAU** and **AAW**. For the purpose of this work we will, in fact, only use operator **EEU**. From the latter, we can derive operators **EEX** and **EEF** in the following way [2]:

$$\begin{aligned} \mathbf{EEX}\{\chi\}\varphi &\triangleq \mathbf{EE}[\{\mathit{false}\}\mathit{false}\ \mathbf{U}\ \{\chi\}\varphi] \\ \mathbf{EEF}\{\chi\}\varphi &\triangleq \mathbf{EE}[\{\mathit{true}\}\mathit{true}\ \mathbf{U}\ \{\chi\}\varphi] \end{aligned} \tag{2}$$

The meaning of the three ACTLW operators in an LTS is illustrated in Fig. 2. The formula with operator **EEU** expresses that a path exists in the LTS such that a possibly empty set of (χ, φ)-transitions is followed by a (χ', φ')-transition. Formula $\mathbf{EEX}\{\chi\}\varphi$ states that the first transition on the path is a (χ, φ)-transition, whereas $\mathbf{EEF}\{\chi\}\varphi$ states that a (χ, φ)-transition occurs eventually on a path.

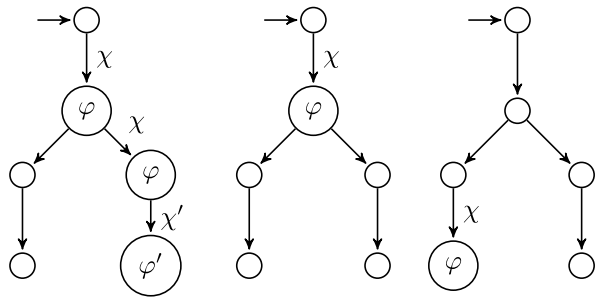


FIGURE 2. Meaning of the ACTLW operators EEU, EEX and EEF.

In order to achieve an even more effective ACTLW syntax, we can abbreviate the formulae further. Instead of $\{\chi\}\mathit{true}$ we write $\{\chi\}$ and define the following abbreviations:

$$\begin{aligned} \mathbf{EEX}\{\chi\} &\triangleq \mathbf{EEX}\{\chi\}\mathit{true} \\ \mathbf{EEF}\{\chi\} &\triangleq \mathbf{EEF}\{\chi\}\mathit{true} \\ \mathbf{EE}[\{\chi\}\ \mathbf{U}\ \{\chi'\}] &\triangleq \mathbf{EE}[\{\chi\}\mathit{true}\ \mathbf{U}\ \{\chi'\}\mathit{true}] \end{aligned} \tag{3}$$

Similarly, we can shorten the expression $\{\mathit{true}\}\varphi$ to φ.

B. NONDETERMINISTIC FINITE AUTOMATA

To define and generate witness automata we need the notion of Nondeterministic Finite Automaton (NFA) with a single initial state. An NFA $\mathcal{A}$ is a 5-tuple $\mathcal{A} = (S, \Sigma, \delta, s_{init}, F)$ [30], where S is a finite set of states, Σ is a finite alphabet, $\delta \subseteq S \times \Sigma \times S$ is a transition relation, s_{init} is an initial state and $F \subseteq S$ is a set of final states. An NFA is, in fact, very similar to an LTS, with the exception of the final states.

The alphabet represents a set of input symbols, which are, in fact, like actions. A path in an NFA is defined analogously to the one in an LTS. A finite sequence (or a string) of symbols $w_1 \ldots, w_n$ from Σ is said to be accepted by $\mathcal{A}$ if and only if there is a finite path π starting in the initial state of the NFA, such that $act(\pi) = w_1, \ldots, w_n$ and the final state of the path is in F. The language of $\mathcal{A}$ is defined as the set of finite sequences of input symbols accepted by $\mathcal{A}$. Two NFAs are equivalent if and only if they have the same language.

IV. WITNESSES FOR ACTLW

For the definition of witness automata, we only need the definition of finite linear witnesses for ACTLW formulae.

Consider LTS $\mathcal{M}$ and an ACTLW formula φ. A finite sequence of actions $act(\pi)$ is a finite linear witness for $s \models_{\mathcal{M}} \varphi$ if and only if there exists a finite path π in $\mathcal{M}$ that starts in state s and shows completely one of the reasons why $s \models_{\mathcal{M}} \varphi$ holds [5]. If s is the initial state of $\mathcal{M}$, then $act(\pi)$ is a finite linear witness for $\mathcal{M} \models \varphi$. Let $\mathcal{M}$ be the LTS in Fig. 3 and $\varphi = \mathbf{EEF}\{b?\}\,\mathbf{EEX}\{c?\}$.

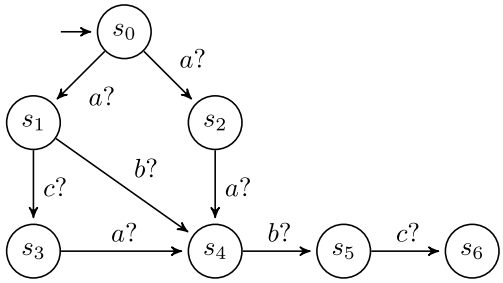


FIGURE 3. LTS $\mathcal{M}$.

Starting in the initial state of $\mathcal{M}$ three paths exist which show that the ACTLW formula is valid in $\mathcal{M}$. From these we can derive finite linear witnesses $\eta_0 = a?, a?, b?, c?$, $\eta_1 = a?, b?, b?, c?$ and $\eta_2 = a?, c?, a?, b?, c?$.

Finite linear witnesses cannot explain the reasons for the validity of all kinds of ACTLW formulae in any LTS. Therefore, we restrict the syntax of ACTLW formulae such that, if valid in an LTS, they certainly have a finite linear witness [6]:

$$\begin{aligned} \chi &::= \mathit{true} \mid \mathit{false} \mid \alpha \mid \neg\chi \mid \chi \vee \chi \mid \\ \varphi &::= \mathit{true} \mid \varphi \vee \varphi \mid \mathbf{EE}[\{\chi\}\mathit{true}\ \mathbf{U}\ \{\chi\}\varphi] \mid \\ &\quad \mathbf{EE}[\{\chi\}\mathit{false}\ \mathbf{U}\ \{\chi\}\varphi] \mid \mathbf{EE}[\{\mathit{false}\}\varphi\ \mathbf{U}\ \{\chi\}\varphi] \mid \\ &\quad \mathbf{EEX}\{\chi\}\varphi \mid \mathbf{EEF}\{\chi\}\varphi \end{aligned} \tag{4}$$

Despite being abbreviations, we deal with the formulae with operators **EEX** and **EEF** separately. Therefore, such formulae are included in (4).

A finite linear witness for a formula φ can be extended arbitrarily by appending actions that are either related or unrelated to the validity of the formula in a given LTS. Regardless of the newly added actions, the sequence remains a finite linear witness for the formula. If a finite linear witness has no such suffix, it is called a viable finite linear witness and denoted as a $\mathcal{V}$-witness [5]. For a state formula φ with the syntax given in (4) and LTS $\mathcal{M}$ with a finite path $\pi = s_0, a_1, s_1, \ldots, a_n, s_n$, where $n \geq 0$, the sequence $\eta = act(\pi)$ is declared to be a $\mathcal{V}$-witness for $s_0 \models_{\mathcal{M}} \varphi$ if and only if the following applies:

- $\varphi = \mathit{true}$ and η is an empty sequence of actions,
- $\varphi = \varphi_1 \vee \varphi_2$ and η is a $\mathcal{V}$-witness for $s_0 \models_{\mathcal{M}} \varphi_1$ and no proper prefix of η is a $\mathcal{V}$-witness for $s_0 \models_{\mathcal{M}} \varphi_2$ or η is $\mathcal{V}$-witness for $s_0 \models_{\mathcal{M}} \varphi_2$ and no proper prefix of η is a $\mathcal{V}$-witness for $s_0 \models_{\mathcal{M}} \varphi_1$,

- $\varphi = \mathbf{EE}[\{\chi_1\}true\ \mathbf{U}\ \{\chi_2\}\varphi_2]$ and there exists $1 \leq i \leq n$, such that $a_i \models \chi_2$ and η without first i actions is a $\mathcal{V}$-witness for $s_i \models_{\mathcal{M}} \varphi_2$, for each $1 \leq j \leq i-1$, $a_j \models \chi_1$ and either $a_j \not\models \chi_2$ or $s_j \not\models_{\mathcal{M}} \varphi_2$ holds,
- $\varphi = \mathbf{EE}[\{\chi_1\}false\ \mathbf{U}\ \{\chi_2\}\varphi_2]$, $\varphi = \mathbf{EE}[\{false\}\varphi_1\mathbf{U}\ \{\chi_2\}\varphi_2]$ and $a_1 \models \chi_2$ holds, whereas η without the first action is a $\mathcal{V}$-witness for $s_1 \models_{\mathcal{M}} \varphi_2$,
- $\varphi = \mathbf{EEX}\{\chi_2\}\varphi_2$ and $a_1 \models \chi_2$ and η without the first action is a $\mathcal{V}$-witness for $s_1 \models_{\mathcal{M}} \varphi_2$,
- $\varphi = \mathbf{EEF}\{\chi_2\}\varphi_2$ and there exists $1 \leq i \leq n$, such that $a_i \models \chi_2$ holds, whereas η without the first i actions is a $\mathcal{V}$-witness for $s_i \models_{\mathcal{M}} \varphi_2$, and for each $1 \leq j \leq i-1$ either $a_j \not\models \chi_2$ or $s_j \not\models_{\mathcal{M}} \varphi_2$.

Hereafter we write shortly witness in place of viable finite linear witness.

V. WITNESS AUTOMATA

Assume a finite LTS $\mathcal{M}$ and an ACTLW formula φ with the syntax from (4). We define the witness automaton $\mathcal{WA}$ for $\mathcal{M} \models \varphi$ as an NFA with a language which is the set of all witnesses for $\mathcal{M} \models \varphi$. To generate a witness automaton $\mathcal{WA}$ for ACTLW formula in an LTS, we first generate an NFA representing the formula and afterwards calculate a synchronous product between that automaton and the LTS. The automaton representing a formula φ for a (finite) set of actions A is an NFA $\mathcal{A} = (S, \Sigma, \delta, s_{init}, F)$, where $\Sigma \subseteq A$ and the language of which consists of all the possible witnesses of φ for A. A finite sequence $\eta = a_1, \ldots, a_n$ of actions from A is a possible witness of φ for A if and only if there exists such an LTS $\mathcal{M} = (S_{\mathcal{M}}, Act_\tau, D_{\mathcal{M}}, s_{init_{\mathcal{M}}})$ that A is (a subset of) the set of reachable actions of $\mathcal{M}$ and η is a witness for $\mathcal{M} \models \varphi$.

We limit the generation of the automaton to a set of actions A for the following reasons. One is that some formulae are valid in an LTS with an arbitrary labeled transition between a certain pair of states. Consequently, there would be infinitely many possible witnesses for such a formula, implying that the automaton should contain infinitely many transitions between a certain pair of states, each labeled with a different action, but we would like the automaton to be finite. Another reason is that, in order to calculate the synchronous product between the automaton and the LTS for which the witness automaton is to be generated symbolically, the actions common to the formula automaton and the LTS should be encoded with logical variables in the same way in both. It should be noticed that if an action of the LTS is not reachable, it does not affect the validity of a formula in it. We, therefore, generate the automaton representing the formula for a finite set of actions A and, generally, interpret the latter as the set of reachable actions of any LTS for which this automaton would potentially be used to generate the witness automaton. In the implementation, however, A is currently calculated as the set of reachable actions of the LTS for which the witness automaton is being sought. To represent the ACTLW formulae from (4), we developed an algorithm (Algorithm 1) which generates a suitable NFA recursively, based on the structure of the given formula φ and the set of actions A. The generation of the NFA is called with the loop parameter equal to true.

In general, the NFA for an arbitrary ACTLW formula allowed by (4) is constructed by combining the basic forms of NFAs shown in Fig. 4. To represent the (sub)formula *true*, we introduced the single-state NFA (*a*), the language of which contains only the empty string. This is in accordance with the definition of witnesses in Section IV. NFAs (*b*), (*c*) (as well as (*d*)), and (*e*) (as well as (*f*)) contain all the possible witnesses for the (sub)formulae **EEX**, **EEF**, and **EEU**, respectively, as defined in Section IV, assuming $\varphi_2 = true$, except that A is also taken into account. NFAs (*d*) and (*f*) are equivalent to (*c*) and (*e*), respectively. By constructing (*d*) and (*f*), we eliminated the loop in the initial state of (*c*) and (*e*). To achieve that, we introduced an additional state, resulting in slightly larger NFAs. Such NFAs are used to simplify the implementation for certain kinds of ACTLW formulae, as commented on later on.

We abuse the notation in the automata representing ACTLW formulae a little. Even though the transitions in an NFA are, by definition, labeled with a single action, we label edges with formulae of the form $\chi \wedge A$, which represent multiple actions (analogous to multiple labeled edges). An action between two states is meant to satisfy formula $\chi \wedge A$ if it is in the set $/\chi/ \cap A$.

We distinguish between two types of ACTLW formulae based on (4). There are formulae for which there is no LTS $\mathcal{M}$ with the set of reachable actions $A \subseteq Act_\tau$, such that $\mathcal{M} \models \varphi$ and, otherwise, formulae for which such an LTS does exist. We call the latter satisfiable formulae. The NFA is not generated for the unsatisfiable ones. Even though (4) does not allow for formulae with $\varphi = false$, a formula φ can still be unsatisfiable. This is the case if φ is equivalent to *false* for the given A. For example, formulae $\mathbf{EEX}\{\chi_2\}\varphi_2$, $\mathbf{EEF}\{\chi_2\}\varphi_2$ and $\mathbf{EE}[\{\chi_1\}\ \mathbf{U}\ \{\chi_2\}\varphi_2]$ are unsatisfiable when either φ_2 is unsatisfiable or $\chi_2 \wedge A \equiv false$. The latter holds if either χ_2 is equivalent to *false* or the actions that satisfy χ_2 are not in A. In both cases, the set of actions that satisfy $\chi_2 \wedge A$ is empty. Consequently, supposing that $\varphi_2 = true$, it can be seen from Fig. 4 that an NFA for such ACTLW formulae does not exist and cannot be generated, because the label $\chi_2 \wedge A$, in fact, does not define any transition.

For all previously mentioned basic formulae, if $\varphi_2 = true$ and if they are satisfiable, Algorithm 1 returns exactly one of the NFAs with the loops in the initial state from Fig. 4. For example, for formula $\mathbf{EEF}\{\chi_2\}\varphi_2$, if $\chi_2 \wedge A \neq false$ and $\overline{\chi_2} \wedge A \neq false$, automaton (*c*) is returned, the language of which contains all the strings beginning with a finite, possibly empty, string of actions in A which do not satisfy χ_2 and ending with an action in A which satisfies χ_2. With the formula $\overline{\chi_2}$ on the transitions to the non-final states we express the requirements for actions from the definitions of viable witnesses. Please notice that Algorithm 1 contains conditional clauses which ensure the satisfiability of the

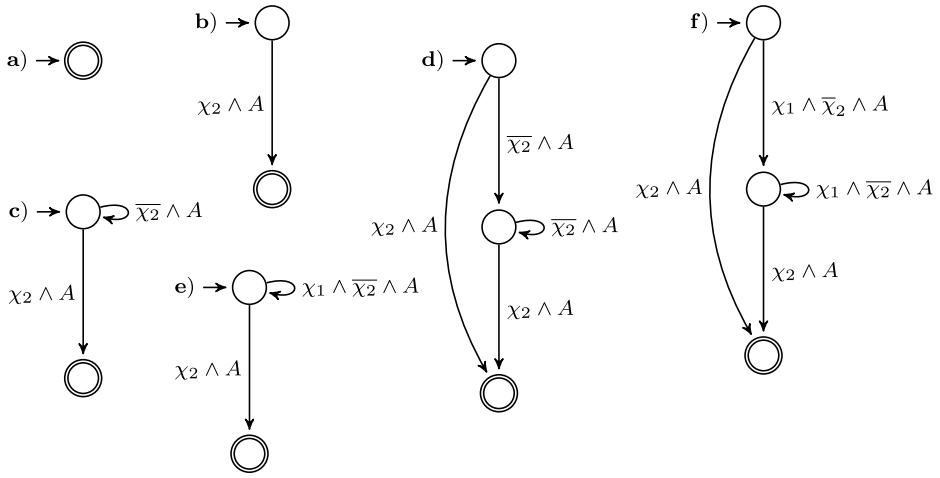


FIGURE 4. Basic forms of NFAs for ACTLW formulae.

formula, but, for ease of presentation, the branches for the unsatisfiability cases are not included.

Included in (4) and in Algorithm 1 are also formulae of the form $\varphi \vee \varphi$, e.g., $\varphi = true \vee true$, which gives automaton (*a*). A more complex example is a formula $\varphi = \mathbf{EEX}\{\chi_1\} \vee \mathbf{EEF}\{\chi_2\}$. Generation of an NFA for such a formula requires alternative composition (denoted with "+") of the NFAs representing its disjuncts. We generate NFAs for subformulae $\mathbf{EEX}\{\chi_1\}$ and $\mathbf{EEF}\{\chi_2\}$ separately. From these NFAs we then construct the NFA for the complete formula by performing the composition of the two. The NFA obtained by alternative composition accepts all strings from the language of one and the other NFA, thus representing the witnesses of both subformulae. To simplify the composition, we do not allow for the loops in the initial state of the composed NFAs. In conclusion, to obtain the NFA for $\mathbf{EEX}\{\chi_1\} \vee \mathbf{EEF}\{\chi_2\}$, we use the NFA of the form (*d*) and not (*c*) from Fig. 4 to represent the disjunct $\mathbf{EEF}\{\chi_2\}$. To express the requirements for the absence of the loop generally, we call the generation of the NFAs for the disjuncts by the loop parameter set to *false* in Algorithm 1.

A sequential composition (denoted with "·") of two NFAs is an NFA that accepts strings consisting of one of the sequences of possible actions from the initial to a final state of the first NFA, followed by one of such sequences for the second one. We use sequential composition for ACTLW formulae with nested subformulae, such as, for example, $\varphi = \mathbf{EEF}\varphi_2$, where $\varphi_2 = \mathbf{EEF}\{\chi\}$ and $\chi \wedge A \neq false$. In this case, we generate an NFA for the nested subformula $\varphi_2 = \mathbf{EEF}\{\chi\}$ (automaton (*c*) from Fig. 4) and an NFA for the outer formula $\mathbf{EEF}\{true\}$, which is short for $\mathbf{EEF}\{true\}true$. Note that here we treat φ_2 as *true*, even though φ_2 is ultimately a nested ACTLW subformula. Therefore, we generate an NFA for the outer formula as (*c*) from Fig. 4, however, as can be seen in Algorithm 1, without the formula $\overline{\chi_2}$ on the loop transition. Finally, we perform sequential composition in such a way that we append the NFA representing the subformula $\mathbf{EEF}\{\chi\}$ to the NFA of $\mathbf{EEF}\{true\}$. Note that the sequentially composed NFA representing the nested subformula is allowed to have the loop in the initial state, which is the reason for calling its generation in Algorithm 1 with the loop parameter equal to *true*. Therefore, in our example, we can use the NFA of the form (*c*) instead of (*d*) from Fig. 4 to represent $\mathbf{EEF}\{\chi\}$. However, the NFA representing the outer formula may contain the loop in the initial state only if it is not a disjunct of another one. That is why the generation of the NFA for an outer formula is called with the loop parameter *loop*, as can be seen from Algorithm 1. The value of the loop parameter is determined by the instance of the procedure that calls it.

Formal definitions of the alternative and sequential compositions we use are given in Fig. 5, where $P = (S_P, \Sigma_P, s_{init_P}, F_P)$ and $Q = (S_Q, \Sigma_Q, s_{init_Q}, F_Q)$. It should be noticed that they do not work for arbitrary NFAs, i.e., with loops in arbitrary states. Such definitions usually require that special empty transitions are introduced (e.g., [31]). We wanted the implementation of the compositions to be simple and the composed automata as small as possible. That is why we introduced the loop parameter. As we need only certain forms of NFAs, we can use the simpler composition definitions and eliminate the loop in the initial states of the components before the compositions, if necessary, in order for them to work properly.

An example of alternative and sequential compositions for automata P and Q representing ACTLW formulae $\mathbf{EEF}\{a\}true$ and, respectively, $\mathbf{EEF}\{d\}true$ for $A = \{a, b, c, d, e\}$, is given in Fig. 6. For the sake of simplicity, we do not label actions with "?" and "!", and represent multiple transitions between two states with one, labeled with multiple actions. Note that the definitions from Fig. 5 include unreachable states, which are not present in Fig. 6. It can be seen that the alternative composition would not be correct if the variants of P and Q with the loops in the initial states were composed.

In the next step of witness automaton generation we combine the NFA $\mathcal{A}$ representing the ACTLW formula and the LTS $\mathcal{M}$ into a witness automaton $\mathcal{WA}$ by calculating the

Algorithm 1: Procedure Automaton($\varphi, A, loop$)

Data: ACTLW formula φ, set of actions A, Boolean parameter *loop*
Result: Automaton for satisfiable ACTLW formula φ
if ($\varphi = true$) **then**
  return automaton (a);
else if ($\varphi = \varphi_1 \vee \varphi_2$) **then**
  if ($\varphi_1 = true \vee \varphi_2 = true$) **then**
    return automaton (a);
  else if (φ_i *for one of* $i = 1, 2$ *is satisfiable*) **then**
    return Automaton($\varphi_i, A, true$);
  else // if φ_1 and φ_2 are satisfiable
    return Automaton($\varphi_1, A, false$) + Automaton($\varphi_2, A, false$);
else if ($\varphi = \mathbf{EEX}\{\chi_2\}\varphi_2$) **and** ($\chi_2 \wedge A \neq false$) **then**
  if $\varphi_2 = true$ **then**
    return automaton (b);
  else // if φ_2 is satisfiable
    return automaton (b) · Automaton($\varphi_2, A, true$);
else if ($\varphi = \mathbf{EEF}\{\chi_2\}\varphi_2$) **and** ($\chi_2 \wedge A \neq false$) **then**
  if ($\varphi_2 = true$) **then**
    if ($\overline{\chi_2} \wedge A \neq false$) **then**
      if ($loop = false$) **then**
        return automaton (d);
      else
        return automaton (c);
    else if ($\overline{\chi_2} \wedge A = false$) **then**
      return automaton (b);
  else // if φ_2 is satisfiable
    if ($loop = false$) **then**
      return automaton (d) with formula A instead of $\overline{\chi_2} \wedge A$ · Automaton($\varphi_2, A, true$);
    else
      return automaton (c) with formula A instead of $\overline{\chi_2} \wedge A$ · Automaton($\varphi_2, A, true$);
else if ($\varphi = \mathbf{EE}[\{false\}\varphi_1 \mathbf{U} \{\chi_2\}\varphi_2]$) **then**
  return the same automaton as for $\varphi = \mathbf{EEX}\{\chi_2\}\varphi_2$;
else if ($\varphi = \mathbf{EE}[\{\chi_1\}false \mathbf{U} \{\chi_2\}\varphi_2]$) **then**
  return the same automaton as for $\varphi = \mathbf{EEX}\{\chi_2\}\varphi_2$;
else if ($\varphi = \mathbf{EE}[\{\chi_1\}true \mathbf{U} \{\chi_2\}\varphi_2]$) **then**
  if ($\chi_1 \wedge A = false$) **then**
    return the same automaton as for $\varphi = \mathbf{EEX}\{\chi_2\}\varphi_2$;
  else if ($\chi_1 = true$) **then**
    return Automaton($\mathbf{EEF}\{\chi_2\}\varphi_2, A, loop$);
  else if ($\chi_2 \wedge A \neq false$) **then**
    if ($\varphi_2 = true$) **then**
      if ($\chi_1 \wedge \overline{\chi_2} \wedge A \neq false$) **then**
        if ($loop = false$) **then**
          return automaton (f);
        else
          return automaton (e);
      else if ($\chi_1 \wedge \overline{\chi_2} \wedge A = false$) **then**
        return automaton (b);
    else // if φ_2 is satisfiable
      if ($loop = false$) **then**
        return automaton (f) with formula $\chi_1 \wedge A$ instead of $\chi_1 \wedge \overline{\chi_2} \wedge A$ · Automaton($\varphi_2, A, true$);
      else
        return automaton (e) with formula $\chi_1 \wedge A$ instead of $\chi_1 \wedge \overline{\chi_2} \wedge A$ · Automaton($\varphi_2, A, true$);

synchronous product of the two. In the synchronous product, $\mathcal{M}$ and $\mathcal{A}$ are allowed to participate in a transition if and only if the individual transitions in both are labeled with the same action.

Assume $\mathcal{M} = (S_\mathcal{M}, Act_\tau, \delta_\mathcal{M}, s_{init_\mathcal{M}})$ is a finite LTS with the set of reachable actions A and $\mathcal{A} = (S_\mathcal{A}, \Sigma, \delta_\mathcal{A}, s_{init_\mathcal{A}}, F_\mathcal{A})$ is the NFA representing the ACTLW formula for A. The synchronous product of the two, denoted $\mathcal{M}||\mathcal{A}$, is the NFA $\mathcal{WA} = (S_{\mathcal{M}||\mathcal{A}}, \Sigma_{\mathcal{M}||\mathcal{A}}, \delta_{\mathcal{M}||\mathcal{A}}, s_{init_{\mathcal{M}||\mathcal{A}}}, F_{\mathcal{M}||\mathcal{A}})$ where:

$$
\begin{aligned}
S_{\mathcal{M}||\mathcal{A}} &= S_\mathcal{M} \times S_\mathcal{A} \\
\delta_{\mathcal{M}||\mathcal{A}} &= \{((s_\mathcal{M}, s_\mathcal{A}), a, (s'_\mathcal{M}, s'_\mathcal{A}))|(s_\mathcal{M}, a, s'_\mathcal{M}) \in \delta_\mathcal{M} \\
&\qquad \wedge (s_\mathcal{A}, a, s'_\mathcal{A}) \in \delta_\mathcal{A}\} \\
s_{\mathcal{M}||\mathcal{A}} &= (s_{init_\mathcal{M}}, s_{init_\mathcal{A}}) \\
F_{\mathcal{M}||\mathcal{A}} &= S_\mathcal{M} \times F_\mathcal{A}
\end{aligned} \tag{5}
$$

By definition (5), $S_{\mathcal{M}||\mathcal{A}}$ may include unreachable, as well as non-final, states with no outgoing transitions. We implemented the synchronous product in such a way that non-final states with no outgoing transitions are eliminated, whereas unreachable states do not occur at all (see Section VI). If the result contains any states from $F_{\mathcal{M}||\mathcal{A}}$, it is a witness automaton, and it is called a reduced $\mathcal{WA}$. An example of a $\mathcal{WA}$ with unreachable states eliminated and its reduced form for ACTLW formula **EEF EEX**$\{a\}true$, its automaton $\mathcal{A}$ and LTS $\mathcal{M}$ from Fig. 3 is shown in Fig. 7.

Please note that not all the conditions for viable witnesses, described in Section IV, are checked when composing the NFAs representing formulae. The checking is not always possible, because the composed NFAs are generated independently from each other. Consequently, only the conditions from the definition of viable witnesses related to actions are checked. This is one reason why the formula automata, and, thus, witness automata, can contain some non-viable witnesses. Another reason is that, in the case of formulae of the form $\mathbf{EE}[\{\chi_1\}true \mathbf{U} \{\chi_2\}\varphi_2]$ and $\mathbf{EEF}\{\chi_2\}\varphi_2$ with a normal formula φ_2, i.e., not *true*, we have also decided not to check all the conditions for actions. Notice that we left out action formula $\overline{\chi_2}$ on the transitions of the NFA of the external formula leading to a non-final state, thereby allowing all the actions which satisfy formula $\chi_1 \wedge A$ (in the case of **EEU**) and, respectively, all the actions from A (in the case of **EEF**). If we checked the condition expressed with action formula $\overline{\chi_2}$ for these kinds of ACTLW formulae, it could happen that there were no witnesses for them. Therefore, we rather allow for non-viable witnesses than to obtain no witnesses at all. For example, consider the LTS obtained from the one in Fig. 3 by deleting the transition from state $s1$ to $s4$ and ACTLW formula $\mathbf{EEF}\{a\}$ $\mathbf{EEX}\{b\}true$. If we checked the condition, in this case expressed with formula $\overline{a}$, there would be no witness for the validity of this formula in the LTS, and, if not, there would be (two) viable witnesses. If, for example, a loop with actions a and b were added in the initial state of that LTS, there would, additionally, be non-viable witnesses with repetitions of actions a and b in the beginning, and ending with a and b.

VI. IMPLEMENTATION OF WITNESS AUTOMATA GENERATION

The witness automata generation procedure was implemented (1.5 KLoC) in EST (Efficient Symbolic Tools) [32]. EST is a toolbox for formal specification and symbolic verification of finite labeled transition systems, which uses characteristic functions and BDDs internally. The systems can be specified

$$
\begin{aligned}
S_{P+Q} &= S_P \times \{s_{init_Q}\} \cup \{s_{init_P}\} \times S_Q & S_{P\cdot Q} &= S_P \times \{s_{init_Q}\} \cup F_P \times S_Q \\
\Sigma_{P+Q} &= \Sigma_P \cup \Sigma_Q & \Sigma_{P\cdot Q} &= \Sigma_P \cup \Sigma_Q \\
\delta_{P+Q} &= \{((s_P, s_{init_Q}), a, (s'_P, s_{init_Q}))| & \delta_{P\cdot Q} &= \{((s_P, s_{init_Q}), a, (s'_P, s_{init_Q}))| \\
&\quad (s_P, a, s'_P) \in \delta_P\} \cup & &\quad (s_P, a, s'_P) \in \delta_P\} \cup \\
&\quad \{((s_{init_P}, s_Q), a, (s_{init_P}, s'_Q))| & &\quad \{((F_P, s_Q), a, (F_P, s'_Q))| \\
&\quad (s_Q, a, s'_Q) \in \delta_Q\} & &\quad (s_Q, a, s'_Q) \in \delta_Q\} \\
s_{init_{P+Q}} &= (s_{init_P}, s_{init_Q}) & s_{init_{P\cdot Q}} &= (s_{init_P}, s_{init_Q}) \\
F_{P+Q} &= F_P \times \{s_{init_Q}\} \cup \{s_{init_P}\} \times F_Q & F_{P\cdot Q} &= F_P \times F_Q
\end{aligned}
$$

FIGURE 5. Definition of alternative and sequential compositions of NFAs.

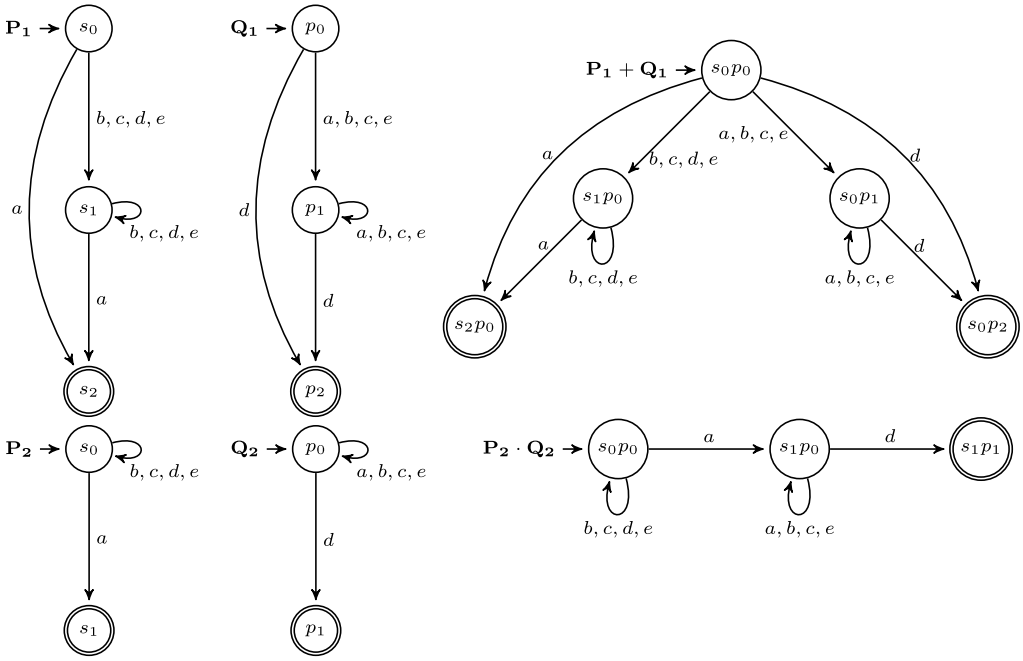


FIGURE 6. Alternative and sequential composition of automata *P* and *Q*.

in a language similar to process algebra CCS (Calculus of Communicating Systems) [33]. Every process and composition of processes in CCS is represented internally with an LTS. EST supports the verification by model-checking ACTLW and ACTL formulae, as well as some operations regarding equivalence relations.

Before generating a witness automaton for a given formula and LTS, we perform model checking. The result is not only the truth value of the formula, but also the description of the structure of the formula in the form of a (binary) subformulae tree. The witness automaton generation is called only if the LTS satisfies the ACTLW formula.

The subformulae tree is used by the symbolic implementation of Algorithm 1 (i.e., of procedure Automaton) for the recursive formula automaton generation. Not included in Algorithm 1 is the checking for syntax errors and unsatisfiability for a given formula. Ultimately, the NFA is generated for subformulae which are in accordance with (4) and meet the conditions from Algorithm 1. We implemented three distinct functions to build the basic kinds of NFAs shown in Fig. 4. Their states are encoded with one or, respectively, two logical variables. Finally, these NFAs are potentially composed depending on the formula structure. The implementation of the alternative and sequential compositions is a direct translation of the definitions given in Section V into set operations with characteristic functions.

The implementation of the synchronous product of an LTS and an NFA represented with BDDs is based on the description in Section V. However, in order to avoid the generation of unreachable states, after generating the initial state, the states and transitions of the product are generated gradually in a similar way to the standard breadth-first search for reachable states [34]. In the first step, the states reached from the initial state by a synchronized transition of the LTS and the NFA are generated and the new transitions added to the transition relation of the product. Afterwards, the states reached

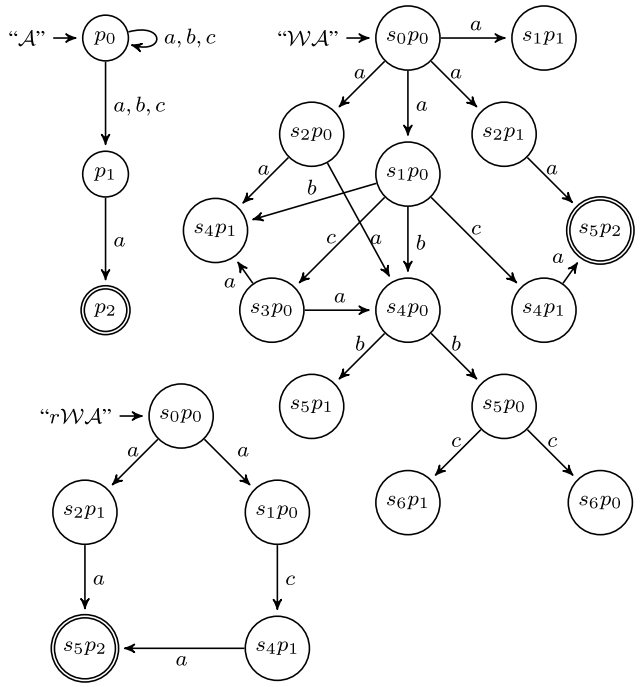


FIGURE 7. NFA $\mathcal{A}$, $\mathcal{WA}$ and the reduced $\mathcal{WA}$.

from any new non-final state generated with such transitions are generated and the transitions added continuously until no new states are generated. The result of the product is a witness automaton ($\mathcal{WA}$). To eliminate from it the non-final states with no outgoing transitions, we implemented a similar symbolic algorithm, which starts in the set of its final states and searches in the breadth-first manner for the states which are backward-reachable from them. The result is the reduced $\mathcal{WA}$. Thanks to the symbolic representation, in every step, both algorithms generate the whole sets of states and transitions at once, which is essential for the time efficiency of the presented witness automata generation procedure.

VII. EMPLOYING WITNESS AUTOMATA

A. ERROR DETECTION

Suppose a system model specified as a parallel composition of processes in the CCS-like language supported by EST is to be verified. Generally, in order to be able to detect errors in it with the help of witness automata, we propose to use a different variant of composition than in CCS. Assume two concurrent processes with actions a? and, respectively, a!, enabled. Originally, they would result in action τ in the composed LTS [33]. Suppose that the required properties are specified in terms of its external actions. Only these and action τ would occur in the generated witness automata, but not the actions participating in the communication, which might be the cause of invalidity of the properties. So-called probes could be introduced in the model in order to be able to observe the internal behavior anyway. These are additional external actions, which add a lot of complexity. We implemented a variant of composition similar to the one of I/O automata [35]), in which the resulting LTS would go to the next state by performing the output action a! instead of τ, thus eliminating the need for a probe. For the purpose of this section, we used this kind of composition exclusively. It makes the specification of systems simpler and the internal behavior observable in the witness automata, but it does not impact the validity of the required properties.

Now, suppose that a system model specified with the alternative kind of composition should satisfy some safety properties, such that their negations can be expressed with formulae in accordance with (4). In EST, to detect errors in the model which falsify some of these properties, one would call model checking with the option of witness automaton generation for the formulae in turn. If a formula was valid in the model, a witness automaton would be generated, and one could explore (possibly internal) causes for the validity. The generation of a witness automaton indicates that the model contains one or more errors causing a violation of the required property. Every witness reveals at least one error, where by the term ''reveal'' we mean that the error is (a part of) the cause of violation of the safety property being verified. Please note that a witness can also exhibit other errors. A long enough segment of the witness has to be explored in order to find the errors revealed by it. All the witnesses have to be explored in order to be sure that all the errors revealed by the automaton were detected. Due to the viability of witnesses, it is most convenient to explore the automaton by starting from every final state and following the transitions backwards. For each final state, as soon as a complete explanation of the violation is found by following a sequence of transitions backwards, the exploration further backwards from the action reached is not needed. Next, we illustrate the error detection with the help of BRP. Throughout this section, we also show how some existing tools can be used for visualization and exploration of witness automata.

The purpose of BRP is reliable transmission of data packets between a producer (P) and a consumer (C) thereof through a lossy channel. For transmission, the packets are divided into consecutive chunks. A mechanism similar to the Alternating Bit Protocol (ABP) is used to transmit the chunks [36]. The transmission of a packet is successful if and only if the chunks are received in the correct order and in a timely manner. A sender and a receiver are in use. The role of the sender is to gather the chunks of a packet from unit P, setting the control bit in them, as well as appending the notifications to the chunks. Possible notifications are I_{FST} – the transmitted chunk is the first and not the last chunk of the packet –, I_{OK} – the transmitted chunk is the last chunk of the packet –, and I_{INC} – for the rest of the chunks. For duplicate detection at the receiver, the control bit in the first chunk is set to 0, and in subsequent fresh chunks it alternates between 0 and 1. Modified chunks are sent from the sender to the receiver in a sequential manner through a lossy channel. The receiver confirms the reception of each of the chunks to the sender by sending an acknowledgment (ACK) through another lossy channel. If the control bit in the received chunk is different

than in the previously received one, the receiver considers it as a fresh chunk and passes its data and notification to unit C. Upon sending a chunk, the sender starts a retransmission timer. It must not expire before the transmitted chunk can reach the receiver and the acknowledgment for it can be received by the sender, i.e., there must be no premature timeout. The sender waits for either *ACK* from the receiver or the expiration of the timer. In the case of expiration, the chunk is sent again if the maximal number of retransmissions has not been reached. Otherwise, the transmission of the current packet is finished unsuccessfully. The receiver starts a timer whenever a chunk is received. If it expires, the process of receiving chunks of the current packet is stopped, and unit C is notified with I_{NOK}. For each successfully transmitted packet, the sender notifies unit P with I_{OK}. A more detailed description of BRP is given in [11]–[13], [37].

We first used a modified specification of BRP from [37] written in the language of EST with packets containing 3 chunks and the maximal number of retransmissions equal to 2. We introduced an error into the specification. Normally, BRP alternates the control bit regardless of the chunk position, even for the first chunk of the packets. In our case, each first chunk had a control bit of value 0, which means that there can be two consecutive chunks with no bit alternation. The following safety property is valid in the correct system: there does not exist a path where, after a transmission request (denoted in the specification as external action *REQ*?) from unit P, the receiver passes one of the notifications I_{INC}, I_{OK}, I_{NOK} (*RINC*!, *ROK*!, *RNOK*!) to unit C without passing I_{FST} (*RFST*!) first. In the case of the inserted error, the negation of this property becomes valid, which yields that such a path does exist [38]. We can specify this property in EST as follows:

$$\begin{aligned} \textit{property } F1 == \ & \mathbf{EEF}\{\textit{REQ}?\}\mathbf{EE}[\{\text{NOT } \textit{RFST}!\} \\ & \mathbf{U}\ \{\textit{RINC}! \text{ OR } \textit{ROK}! \text{ OR } \textit{RNOK}!\}]; \end{aligned}$$

TABLE 1. LTS size for BRP and lactose operon example.

LTS	States	Transitions	Non-τ transitions
BRP_W	213	282	282
BRP	2315	5257	1901
BRP_T	6144	16383	8807
SF_W	8930	42197	35623
SF	9284	43763	36901

Let us call the generated LTS *BRP*. We carried out weak minimization over it, thus obtaining the smallest weakly observationally equivalent LTS, denoted with *BRP_W* (please see Table 1 for the sizes of the LTSs). After performing model checking for formula $F1$, witness automata $\mathcal{WA\_BRP}$ and $\mathcal{WA\_BRP\_W}$ were generated for LTS *BRP* and, respectively, *BRP_W*. Despite its size (please see Table 2), $\mathcal{WA\_BRP\_W}$ could be explored in a 2D view by using the *dot* program, which is a part of Graphviz (Graph visualization software) [39]. To obtain the graph, we converted EST's internal representation of this witness automaton to the *dot* format. Fig. 8 shows a small part of the automaton, including its final states and the transitions that lead to them. The internal behavior can be seen as a result of the alternative variant of parallel composition. Fig. 9 shows a part of the witness automaton containing sequences of actions indicating the control bit error in the model. It lies just above the part given in Fig. 8 and is backward reachable from some of the final states. In Fig. 9 one can see faulty paths starting with the pair of actions $F0!$ and $G0!$, which represent the transmission of a chunk with the control bit 0 from the sender into the lossy channel and, respectively, its delivery to the receiver. Actions *ACK*! and *B*! represent the transmission of a positive acknowledgment back. They are followed by external action *SOK*!, which represents notification I_{OK} of unit P, indicating that the transmitted chunk is the last one of the packet being sent. In state 198, the transmission of another packet is started by external action *REQ*? and its first chunk sent with the control bit equal to 0 again. Assuming the informal description of the protocol is correct, one can see immediately that this is an error, because it is not in accordance with the description. By observing a longer segment of the witness, it can be confirmed that the action sequence is, indeed, the cause of the validity of formula $F1$, be it due to the invalid model, i.e., not following the informal description, or due to the incorrect protocol design, assuming the model is valid. Otherwise, the witness should be explored further backwards. Basically, the reason for the validity of the formula is that the receiver takes the new chunk with the control bit 0 as a duplicate of the last one of the previous packet. As faulty paths exhibiting the same reason can be followed backwards from all the seven final states, one can be sure that the control bit error is the one and only one revealed by this witness automaton.

TABLE 2. Formula and witness automata size for BRP and lactose operon example.

WA	LTS	Formula	States (initial/final)	Transitions
		F1	3 (1/1)	42 (44 for *BRP_T*)
$\mathcal{WA\_BRP\_W}$	*BRP_W*	*F1*	275 (1/7)	357
$\mathcal{WA\_BRP}$	*BRP*	*F1*	2979 (1/66)	6648
$\mathcal{WA\_BRP\_T}$	*BRP_T*	*F1*	10476 (1/959)	25562
		A21	3 (1/1)	60
		A22	3 (1/1)	90
$\mathcal{WA\_SF\_W}$	*SF_W*	*A21*	11498 (1/880)	49805
$\mathcal{WA\_SF}$	*SF*	*A21*	11996 (1/940)	51741
$\mathcal{WA\_SF\_T}$	*SF*	*A22*	19508 (1/940)	131289

To observe the other witness automata considered in this section, the previous representation is inadequate, since they are much bigger in size. We have found that some tools from the *mCRL2* toolset [40], [41], which were originally meant for LTSs, could be employed for this purpose, provided the internal representation of witness automata is converted to the ALDEBARAN format [42]. One such tool is *ltsgraph*. Instead of drawing an LTS, we used it to draw and explore $\mathcal{WA\_BRP}$. We have found *ltsgraph* to be appropriate to handle it through its exploration mode, which allows the exploration of small parts of an LTS stepwise. To visualize

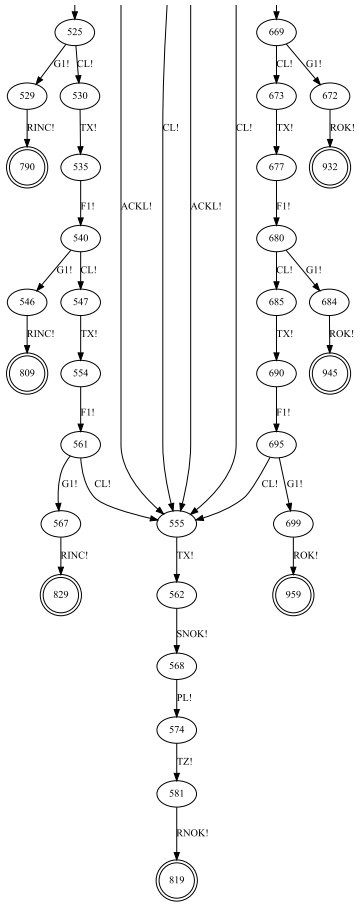


FIGURE 8. A part of witness automaton $\mathcal{BRP\_W}$ including its final states and transitions that lead to them.

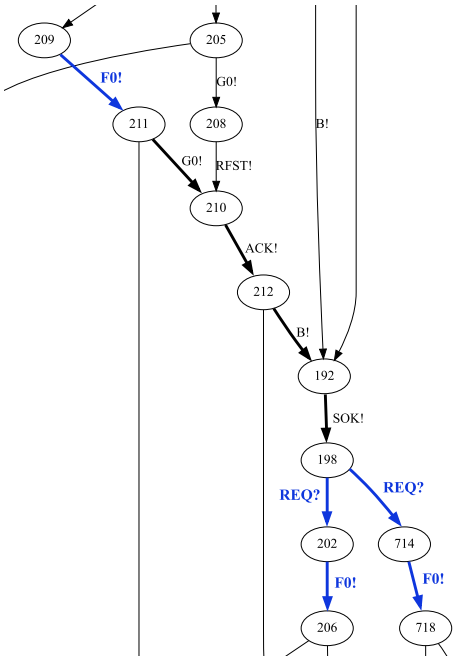


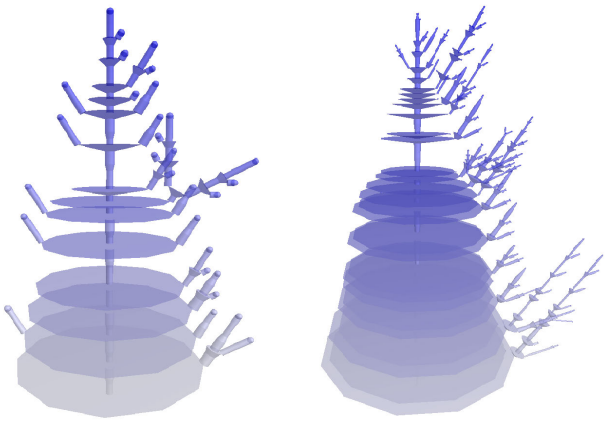
FIGURE 9. A part of witness automaton $\mathcal{WA\_BRP\_W}$ with faulty paths.

and explore the larger witness automata, the *ltsview* tool of the *mCRL2* toolset had to be used. In contrast to *ltsgraph*, it is able to visualize LTSs and, thereby, witness automata using a 3D view while handling very large state spaces [40]. According to [43], *ltsview* clusters states based on structural properties. These clusters are then visualized to form a backbone, on which the states and transitions are drawn.

$\mathcal{WA\_BRP}$ and $\mathcal{WA\_BRP\_W}$ are visualized in Fig. 10. The graphs start with an initial state at the bottom and are followed by clusters of states denoted by the circle shapes. From the initial state, transitions lead to the final states at the graph's tips. Zooming in and out, as well as rotation, of the graphs is possible. A simple way to perform simulation of a witness automaton with *ltsview* is by starting in the initial state and executing actions that lead to the next states, up to the point of interest or until reaching a final state. Another feature of *ltsview* which proved especially useful to explore the automata is the possibility of finding a trace to a state by so-called backtracing. One can select any state and generate the path from the initial state towards it automatically, thus identifying a witness. The path is shown in the graph. It is also given in a table in such a way that it can be examined in a backward stepwise manner by undoing transitions.

To trace a complete witness, we exploited the tool's additional functionality for marking deadlocks in LTSs and, thus, final states in the witness automaton. In the case of $\mathcal{WA\_BRP\_W}$, there were seven deadlocks, denoted by the red color (please see Fig. 11). By selecting a deadlock and performing backtracing, we obtained and examined the path or the witness shown in purple in Fig. 11. It was the witness from Fig. 8 ending in state 829 and eventually reaching backwards a path indicated in Fig. 9.

In order to try to employ a witness automaton to detect more than one error in a model, we changed the components of *BRP* so that the retransmission timer could expire prematurely. As the modeling language in EST, like CCS, does



(a) $\mathcal{WA\_BRP\_W}$. (b) $\mathcal{WA\_BRP}$.

FIGURE 10. Witness automata visualization for the BRP example.

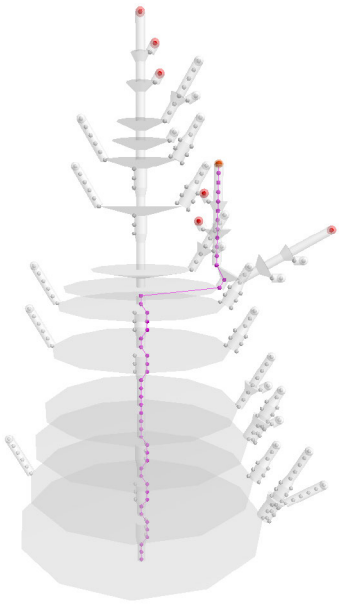


FIGURE 11. A witness in $\mathcal{WA}\_\mathcal{BRP}\_\mathcal{W}$ identified by backtracing in *ltsview*.

not support explicit modeling of time, the functioning of this timer in LTS *BRP* is modeled with one action (*TX*!), which represents expiration of the timer and happens only if either the data chunk or the acknowledgement is lost (cf. [11]). In the new LTS, named *BRP_T*, the sender starts the timer explicitly with action *StartTX*! upon sending a chunk. After the timer is started, it can execute action *TX*! independently of the losses, i.e., either prematurely or not. After starting the timer, the sender waits for the acknowledgement or the timeout. If it receives the acknowledgement, it stops the timer with action *StopTX*!.

We again called model checking with the witness automata generation for formula $F1$, this time with LTS *BRP_T* (the sizes of the NFA and the LTS are given in Table 2 and, respectively, 1). Also in this LTS, containing two errors, the result of model checking was positive. The witness automaton $\mathcal{WA}\_\mathcal{BRP}\_\mathcal{T}$ generated for it consists of approximately 10k states and 25k transitions, and has 959 final states (Table 2). Due to the size of the automaton and the large number of final states, it is, of course, infeasible to check all the witnesses. By backtracing from a few final states using *ltsview*, we found witnesses showing premature timeouts to be a reason of the validity of $F1$, as well as ones revealing the control bit error. Fig. 12 shows a witness which, at least if assuming the informal protocol description to be correct, seems to reveal two errors, but reveals only one. After the first action *REQ*!, the first chunk is transmitted three times (actions $F0$!), because, for the first two times, there was a premature timeout (*TX*!). Each time, the transmitted chunk is received successfully (action $G0$!). For the first chunk, the receiver issues *RFST*!. However, since acknowledgements in the BRP protocol, in contrast to ABP [36], do not carry an alternating bit analogous to the one in the data chunks, the acknowledgement of the first chunk received for the second time is recognized by the sender as the acknowledgement of the second chunk of the packet (action $F1$!), which has been lost (please notice action *TAU*). Consequently, that chunk is not retransmitted. The third chunk of the packet is transmitted instead and taken as acknowledged by the acknowledgement of the first chunk received for the third time. Generally, we found witnesses which, like the presented one, show that unit *P* is informed wrongly by the sender about the success of the transmission of the first packet (by action *SOK*!) and starts the transmission of a new packet by *REQ*!. The receiver does not recognize that the chunks of the new packet are received and issues to unit *C* indications other than *RFST*!. By backtracing the witness in Fig. 12 from action *RINC*!, we notice the control bit error, as the last chunk of the first packet holds the same bit value ($F0$!) as the first chunk of the second packet. However, even if the control bit of the first chunk of the second packet was set correctly, i.e., if there was action $F1$! instead of $F0$! after the second *REQ*!, the receiver would think that was the second chunk of the first packet and perform *RINC*!, not *RFST*!. It follows that this witness reveals only the timeout interval error. Please notice that, assuming the model is valid, one could conclude that the error in the current design is not the possibility of premature timeouts, but rather the absence of the alternating bit in the acknowledgements.

As BRP can be scaled to any number of chunks and retransmissions, we used it to evaluate the new witness automata generation procedure (`WA`) against the existing one (`WCA`). To show the efficiency of the new procedure, we generated witness automata for formula $F1$ applied to *BRP* with the premature timeout error but without the control bit one, because it gives much larger LTSs and witness automata than *BRP* with both kinds of errors. The results, showing the witness automaton generation time versus the number of transitions of the generated automaton, are given in Fig. 13 (please note that all the experiments presented in this paper were performed on a computer with 16 GB of RAM and Intel Core i7-8700 at the frequency of 3.2 GHz). They include, from left to right, the generation time for the witness automaton (excluding the preliminary model checking) for the LTS representing the protocol with packets consisting of 3 chunks and with 2 retransmissions, and for the witness automata generated for the LTS extended to model the transmission of packets of 4, 6, 8, and, respectively, 10 chunks, the latter also with 3 and, respectively, 5 retransmissions. The results show significant improvement in terms of witness automata generation time, even for very large system models and witness automata. For example, the LTS for the variant of BRP with 5 retransmissions, giving the witness automaton with almost 300k transitions, contains 69,171 states and 186,141 transitions. It should be noticed that the implementation of the existing procedure in EST allows only the generation of a witness automaton for a process. The conversion of a composition LTS into a process is supported. It makes the states of the former monolithic. The graph contains the times

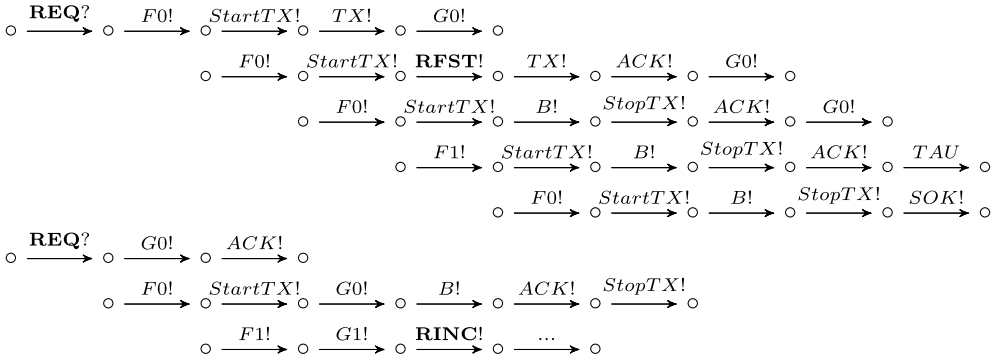


FIGURE 12. A witness from $\mathcal{WA}\_BRP\_T$ revealing the timeout interval error in the model.

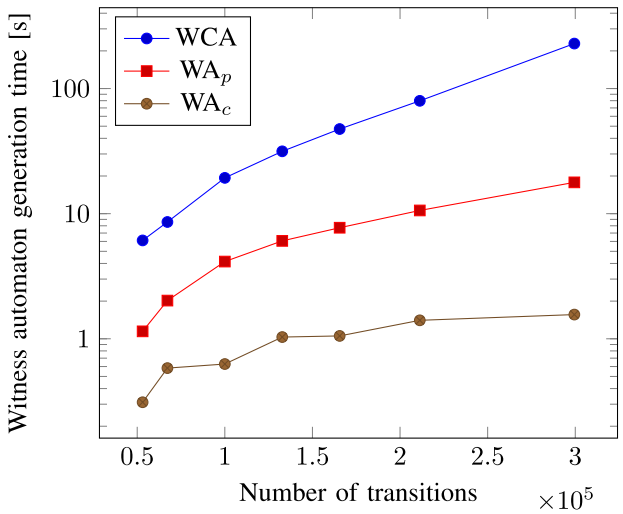


FIGURE 13. Evaluation of the witness automata generation procedures on the BRP benchmark.

for the new procedure applied to the processes ($\mathtt{WA_p}$) and compositions ($\mathtt{WA_c}$) and for the old one to the processes. It should be noticed that the existing procedure generates the states and transitions of the witness automaton one by one and uncoded, whereas the new one generates a witness automaton represented symbolically. The time needed to decode it is not included in the results. The decoding took 9.3 s for the largest witness automaton. The time needed for the initial model checking is equal to 90 ms for the composition LTS with 2 retransmissions and 292 ms for the one with 5 retransmissions. The time needed for the generation of the formula automaton is included in the times for $\mathtt{WA}$ shown in the graph. For typical formulae, such as used in this section, the automaton is generated in less than a millisecond.

B. DETECTION OF UNKNOWN PROPERTIES

Suppose that the behavior of single components of a system is known and that the latter is modeled as a parallel composition of them. Typically, the purpose of this would be to model-check the system for some (un)desired (or, in the case of a non-human-made one, better to say, (un)expected or hypothetical) properties, i.e., sequences of actions, by describing them explicitly with temporal formulae. However, another purpose could be to use model checking to see what sequences of actions lead to some allowed or expected actions, i.e., to detect unknown properties. A typical example would be a complex biological system in which a certain substance can occur, and one would like to know what sequences of reactions lead to that event. The detection of unknown properties could also be useful as an additional means for model validation and system verification. Next, a lactose operon regulatory system model is used to exemplify the idea of detecting unknown properties. Of course, as this system is one of the most studied biological systems, we did not detect any previously unknown system properties, but rather some which were a consequence of the modeling approach.

Operon is the basic unit for gene transcription. The lactose operon in the bacteria *Escherichia coli* grows very well when glucose is present. In addition to glucose there is a substance called Cyclic Adenosine Monophosphate (*cAMP*), which acts as a co-activator of the activator protein in the lactose operon regulation process. The content of *cAMP* depends on the amount of glucose in the cell. The connection between the two is as follows: the higher the glucose content in the cell, the lower the *cAMP* content, and vice versa. This is just an outline of the basic operation of lactose operon regulation. The whole regulation process is described in more detail in [16], [17].

In [17], we presented a formal specification of the lactose operon regulatory system written in the CCS-like language of EST, which is based on the specification from [16] and includes many probes. For the purpose of this paper, we employed the alternative kind of composition, enabling us to largely reduce the number of probes. The generated LTS and its minimized form are denoted as *SF* and *SF_W*, respectively (please see Table 1). One would expect that in these LTSs, whenever the content of *cAMP* is high, it does not decrease without the glucose content increasing first. However, in [17], we found, surprisingly, that negation of this property was valid: a path exists on which the content of *cAMP* is high, and it decreases without the increase of glucose content. Its formal specification is as follows (please note that

in [17], the expected property is called $A21$, not its negation):

$$\begin{aligned} &\textit{property } A21 \\ &== \mathbf{EEF}\{L\_to\_H!\}\ \mathbf{EE}[\{NOT\ GLU\_L\_to\_H!\} \\ &\quad \mathbf{U}\ \{H\_to\_L!\}]; \end{aligned}$$

Here, actions $L\_to\_H!$, $H\_to\_L!$ and $GLU\_L\_to\_H!$ represent the increase in *cAMP* content, the decrease in *cAMP*, and the increase in glucose content, respectively. Based on a counterexample generated by EST, we guessed the reason was that the inquiry into the level of glucose, made by the process representing *cAMP*, and its reaction to the information do not form an indivisible action. Thus, the $GLU\_L\_to\_H!$ action does not necessarily express the glucose level taken into account when the *cAMP* level is changed. We confirmed that by using the witness automata generation procedure for formula $A21$ and both LTSs. The sizes of the generated witness automata are given in Table 2. Unfortunately, transition labels cannot be displayed in a 3D view in *ltsview*. To explore witness automaton $\mathcal{WA}\_\mathcal{SF}$ with it, we, therefore, exploited its capability to indicate transitions labeled with certain actions with different colors (please see Fig. 14). Thereby, we could explore only the paths where all the actions $GLU\_L\_to\_H!$, $H\_to\_L!$ and, respectively, $L\_to\_H!$ took place. It could be seen from $\mathcal{WA}\_SF$ that action $GLU\_L\_to\_H!$ is followed immediately by action $L\_to\_H!$, whereas action $H\_to\_L!$ is performed some time after the latter, suggesting that, after the increase of glucose, the amount of *cAMP* is increased first before the decrease. This motivated us to ask, by using a proper formula, about all the possible paths leading to a decrease of *cAMP*, in order to illustrate the detection of unknown properties.

The general form of the formula is **EEF EEF**$\{action\}$, where *action* is an action formula denoting the action (or possibly a combination of actions) for which the paths are being sought. Please note that, if the formula **EEF**$\{action\}$ was employed, due to the viability of the witnesses, the obtained witness automaton would contain only witnesses which ended with actions satisfying formula *action* and in which this was the first occurrence of such actions. However, it could be the case that unknown (sub)sequences of actions occur in the model after the first or even several occurrences of those actions. Only the first kind of formula allows witnesses with the repeating actions and, thus, detecting such properties.

We generated witness automaton $\mathcal{WA}\_\mathcal{SF}\_\mathcal{T}$ for LTS *SF* and formula $A22$ defined in EST as *property* $A22$ $=$ **EEF EEF**$\{H\_to\_L!\}$. The formula and witness automaton's size are given in Table 2. The generation (without model checking) for the original LTS took 0.07 s, for the process obtained from it 0.747 s, and the decoding 0.043 s. With the old procedure, the generation took 17 s. The time needed for the model checking of the original LTS was 31 ms. By using simulation and backtracing in *ltsview*, we found witnesses exhibiting only the expected functioning of lactose operon regulation, witnesses showing the behavior detected with formula $A21$, as well as witnesses showing two kinds of unknown properties. One kind was that, after an increase of the glucose content ($GLU\_L\_to\_H!$), followed by an inquiry into and an answer about the glucose content (*lev*! and respectively *high*!), a decrease of the glucose content (action *DGLU*! preceeded by τ) followed before a decrease of *cAMP* ($H\_to\_L!$). In contrast to the latter, the other kind of property was, indeed, surprising. The witness automaton showed that, after an occurrence of actions $GLU\_L\_to\_H!$, *lev*! and *high*! and a decrease of the glucose content, another increase and decrease of it could follow, and only after them, but without inquiry into the level of glucose, could the decrease of *cAMP* occur. We found that the reason for both was again the nonatomicity of the inquiry into the level of glucose and the reaction to it.

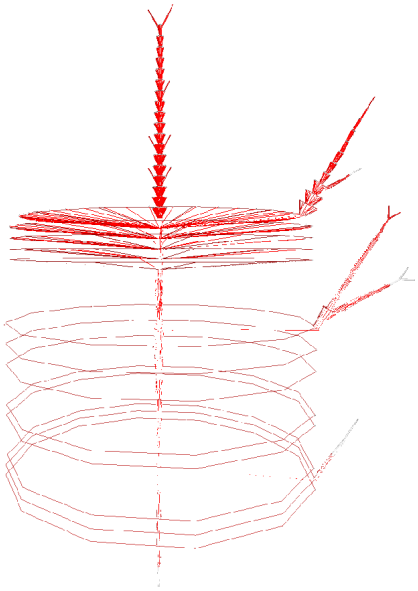


FIGURE 14. Witness automaton $\mathcal{WA}\_SF$ in *ltsview* with indicated transitions labeled with appropriate actions.

VIII. DISCUSSION

The presented experimental results show that the proposed BDD-based breadth-first generation of witness automata takes much less time for large LTSs and witness automata than the old algorithm [5], [6]. Unlike the latter, it could also do without model checking. One reason for the preliminary model checking is that, in EST, the formula parsing is implemented as part of model checking, and we wanted to exploit this feature. Another possibility would be to implement the parsing without model checking and generate the formula automaton during the parsing. If the result of the synchronous product of the automaton and the LTS was a witness automaton, this would mean that the formula was valid in the LTS.

Similar to the old algorithm, it can happen that the generated witness automata contain some non-viable witnesses, as well as several paths containing the same witness. In [5],

it is proposed to minimize the automata to eliminate both. However, this, as well as preliminary minimization of the LTSs being verified, could only help to some extent. The resulting automata could still be very large, because they can contain many similar witnesses as a consequence of different interleavings of concurrent system events. In order for *ltsview* to ease the employment of witness automata for the detection of errors and unknown properties, we would wish it to also be able to display action labels on the transitions in the 3D graph, when zoomed in sufficiently, allow to explore it by clicking the transitions, and indicate, as well as remember, the transitions already explored in this way or by backtracing.

It would be useful to introduce witness automata into practice as another means for error detection besides the usual model checking, giving single witnesses (or, in fact, counterexamples). According to [10], to speed up the debugging using model checking, upon checking the validity of a formula, the designer should be given as much feedback as possible before he or she modifies the model. The availability of multiple counterexamples in the automaton would not only allow the designer to detect multiple errors before having to modify the model, but could also facilitate the detection of a single error. Note that multiple counterexamples can show different patterns of violation of the required property even if the reason for it is a single error. If the designer is debugging a valid model to achieve the system correctness, the multiple patterns can better help him or her to get an idea about the error than a single one. As can be seen from the evaluation results, even for larger LTSs, the times needed to generate witness automata with the proposed procedure are such that it can be rather acceptable for the designer to perform debugging by using witness automata instead of, or in combination with, the usual model checking. Even if the designer explored only a few witnesses in an automaton, it would, generally, be more helpful in terms of diagnostics than having only the one obtained with the usual model checking.

With a proper tool for visual exploration, witness automata could be very useful in discovering previously unknown properties of complex biological systems. It should be noticed that the usual model checking is not appropriate for this purpose. Given a valid system model, in order to obtain all the witnesses leading to a chosen action, the model checking should be repeated for the same formula, with the model changed every time to prevent the last returned witness, until all the witnesses from the original model were obtained. Making such changes in the model would be very difficult, if not impossible, and, in fact, unreasonable, because, in contrast to the debugging, the goal is not to eliminate the discovered action sequences from the system's behavior.

Currently, definition (4) does not include formulae of the form $\mathbf{EE}[\{\chi\}\mathit{false}\ \mathbf{W}\ \{\chi\}\varphi]$ and $\mathbf{EE}[\{\mathit{false}\}\varphi\ \mathbf{W}\ \{\chi\}\varphi]$, which are included in [6]. For LTSs without deadlock states, they are equivalent to the analogous formulae with operator **EEU** in (4). In contrast to the latter, they are true in a deadlock state, and a witness for that would be an empty sequence [6]. To define NFAs for them, a special action leading to a final state could be introduced to ''catch'' the deadlock states of an LTS in a synchronous product. We leave this for future work. A subset of ACTLW formulae having finite linear counterexamples is also defined in [6]. The existing algorithm for the generation of witness automata can also generate counterexample automata for these, in a similar way to the algorithm for ACTL [5], [6]. Removing from them the formulae with operator **AAU** similar to the above mentioned formulae with operator **EEW**, which have an empty counterexample in a deadlock state, only the formulae containing operator **AAW** (and possibly abbreviations **AAX** and **AAG**) remain, whereas the formulae defined by (4) contain the ACTLW operator **EEU** (or are abbreviations based on it). Negations of all the formulae with operator **AAW** can be converted into formulae allowed by (4). It follows that an extension of the parser to support the conversion could be one way to allow the user to apply the presented witness automata generation procedure for the generation of counterexample automata directly for the **AAW** formulae. Instead of the conversion, the generation of NFAs could additionally be defined and implemented for the latter, such that the NFAs would contain possible viable counterexamples for them, i.e., viable witnesses of their negations.

Subsets of ACTL formulae having finite linear witnesses and, respectively, finite linear counterexamples are identified in [5]. All the formulae or, respectively, negations thereof can be expressed with the ACTLW formulae allowed by (4). Please note that the semantics of ACTL is defined only for LTSs with a total transition relation. Thus, the presented witness automata generation procedure could also be applied for the generation of witness and counterexample automata for ACTL for such LTSs by implementing the conversion, or NFAs similar to those for ACTLW could be generated and used in the synchronous product with LTSs. Notice, however, that not all the ACTLW formulae having finite linear witnesses or counterexamples can be expressed with the ACTL formulae from the subsets. Assuming LTSs with a total transition relation, the reason is that, in ACTLW, in contrast to ACTL, action τ is not treated differently than the external actions [6].

A subset of formulae of the universal fragment of CTL (shortly ACTL with A for ''universal'') having linear counterexamples has also been identified, as well as their duals from the Existential fragment of CTL (ECTL), which have linear witnesses [44], [45]. Those formulae having finite linear witnesses could be identified and witness automata defined. The latter could be generated in a similar way to the presented procedure, but with NFAs with the alphabets containing subsets of atomic propositions and a synchronous product defined differently due to the state-based setting (cf. [1]). Counterexample automata could also be obtained analogously to the two ways presented for ACTLW.

IX. CONCLUSION

We developed a symbolic procedure for the generation of witness automata for ACTLW formulae. The procedure's

employment on the bounded retransmission protocol and lactose operon regulatory system shows the new procedure to be at least several times faster than the current method. Some existing tools were shown to be rather appropriate for the exploration of witness automata, although not dedicated for this purpose. Two novel uses for witness automata were introduced, namely, for multiple error detection and discovery of unknown properties in models of concurrent systems. This makes witness automata a promising means to be employed in debugging performed with model checking and in investigating the behavior of complex biological systems.

REFERENCES

[1] C. Baier and J. P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.
[2] R. Meolic, T. Kapus, and Z. Brezočnik, "ACTLW–An action-based computation tree logic with *unless* operator," *Inf. Sci.*, vol. 178, pp. 1542–1557, Mar. 2008.
[3] R. De Nicola and F. Vaandrager, "Action versus state based logics for transition systems," in *LITP Spring School on Theoretical Computer Science* (Lecture Notes in Computer Science), vol. 469. Berlin, Germany: Springer, 1990, pp. 407–419.
[4] A. Fantechi, S. Gnesi, and A. Maggiore, "Enhancing test coverage by backtracing model-checker counterexamples," *Electron. Notes Theor. Comput. Sci.*, vol. 116, pp. 199–211, Jan. 2005.
[5] R. Meolic, A. Fantechi, and S. Gnesi, "Witness and counterexample automata for ACTL," in *Formal Techniques for Networked and Distributed Systems (FORTE)* (Lecture Notes in Computer Science), vol. 3235. Berlin, Germany: Springer, 2004, pp. 259–275.
[6] R. Meolic, "Action computation tree logic with unless operator," Ph.D. dissertation, Fac. Elect. Eng. Comput. Sci., Univ. Maribor, Maribor, Slovenia, 2005.
[7] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. 27, no. 6, pp. 509–516, Jun. 1978.
[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} States and beyond," *Inf. Comput.*, vol. 98, pp. 142–170, Jun. 1992.
[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics* (Lecture Notes in Computer Science), vol. 2000, R. Wilhelm, Ed. Berlin, Germany: Springer, 2001, pp. 176–194.
[10] K. Cabrera Castillos, H. Waeselynck, and V. Wiels, "Show me new counterexamples: A path-based approach," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–10.
[11] J. F. Groote and J. van de Pol, "A bounded retransmission protocol for large data packets: A case study in computer checked algebraic verification," in *Proc. 5th Int. Conf. Algebr. Methodol. Softw. Technol. (AMAST)* (Lecture Notes in Computer Science), vol. 1101. Berlin, Germany: Springer, 1996, pp. 536–550.
[12] P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans, "Modeling and verifying a bounded retransmission protocol," in *Proc. Int. Workshop Appl. Formal Methods Syst. Design (COST)*, Maribor, Slovenia, 1997, pp. 114–128.
[13] R. Meolic, T. Kapus, and Z. Brezočnik, "Exploring properties of a bounded retransmission protocol with VIS," *J. Comput. Inf. Technol.*, vol. 7, no. 4, pp. 311–321, Jan. 1999.
[14] P. F. Castro, P. D'Argenio, R. Demasi, and L. Putruele, "Measuring masking fault-tolerance," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Lecture Notes in Computer Science), vol. 11428. Cham, Switzerland: Springer, 2019, pp. 375–392.
[15] P. van den Bos and J. Tretmans, "Coverage-based testing with symbolic transition systems," in *Tests Proofs (TAP)* (Lecture Notes in Computer Science), vol. 11823. Cham, Switzerland: Springer, 2019, pp. 64–82.
[16] M. C. Pinto, L. Foss, J. C. M. Mombach, and L. Ribeiro, "Modelling, property verification and behavioural equivalence of lactose operon regulation," *Comput. Biol. Med.*, vol. 37, no. 2, pp. 134–148, Feb. 2007.
[17] R. Vogrin, R. Meolic, and T. Kapus, "Formalna specifikacija in verifikacija lastnosti uravnavanja laktoznega operona z orodjem EST," *Elektrotehniski Vestnik*, vol. 84, no. 5, pp. 268–276, Sep. 2007.
[18] M. Falaschi and G. Palma, "A logic programming approach to reaction systems," in *Recent Developments in the Design and Implementation of Programming Languages* (OpenAccess Series in Informatics, Schloss Dagstuhl–Leibniz Zentrum für Informatik). Wadern, Germany: Dagstuhl Publishing, 2020, p. 6.
[19] K. L. McMillan, *Symbolic Model Checking*. Michigan, MI, USA: Kluwer, 1992.
[20] T. A. Henzinger, O. Kupferman, and S. Qadeer, "From *pre*-historic to post-modern symbolic model checking," in *Proc. 10th Int. Conf. Comput. Aided Verification (CAV)* (Lecture Notes in Computer Science), vol. 1427. Berlin, Germany: Springer, 1998, pp. 195–206.
[21] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Proc. 8th Int. Conf. Comput. Aided Verification (CAV)* (Lecture Notes in Computer Science), vol. 1102. Berlin, Germany: Springer, 1996, pp. 419–422.
[22] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 4, pp. 401–424, Apr. 1994.
[23] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. Mckenzie, "SMV–symbolic model checking," in *Systems and Software Verification*. Berlin, Germany: Springer, 2001, pp. 131–138.
[24] R. E. Bryant, "A view from the engine room: Computational support for symbolic model checking," in *25 Years of Model Checking* (Lecture Notes in Computer Science), vol. 5000. Berlin, Germany: Springer, 2008, pp. 145–149.
[25] E. M. Clarke, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *Proc. 32nd Design Autom. Conf.*, New York, NY, USA, Dec. 1995, pp. 427–432.
[26] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.
[27] E. Clarke and S. Jha, "Tree-like counterexamples in model checking," in *Proc. 17th Annu. IEEE Symp. Log. Comput. Sci.*, Jul. 2002, pp. 19–29.
[28] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, "Witness validation and stepwise testification across software verifiers," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, Aug. 2015, pp. 721–733.
[29] D. Beyer and M. Dangl, "Verification-aided debugging: An interactive web-service for exploring error witnesses," in *Proc. 28th Int. Conf. Comput. Aided Verification (CAV)* (Lecture Notes in Computer Science), vol. 9780. Berlin, Germany: Springer, 2016, pp. 502–509.
[30] M. Sipser, *Introduction to the Theory of Computation*. Boston, MA, USA: Thomson Course Technology, 2006.
[31] A. Silva. (2013). *Languages and Automata*. Lecture Notes, Draft Version. Accessed: Oct. 22, 2020. [Online.] Available:. [Online]. Available: http://www.cs.ru.nl/~herman/onderwijs/2016TnA/Silva_TnA.pdf
[32] *Efficient Symbolic Tools Tool*. Accessed: Oct. 22, 2020. [Online]. Available: http://est.meolic.com/home/
[33] R. Milner, *Communication and Concurrency*. New York, NY, USA: Prentice-Hall, 1989.
[34] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. New York, NY, USA: Kluwer, 1996.
[35] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quart.*, vol. 2, pp. 219–246, Nov. 1989.
[36] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. Boston, MA, USA: Pearson, 2017.
[37] R. Meolic, "Checking correctness of concurrent systems behaviour," M.S. thesis, Fac. Elect. Eng. Comput. Sci., Univ. Maribor, Maribor, Slovenia, 1999.
[38] R. Vogrin, "Generating linear finite witness automata for ACTLW formulae with EST," M.S. thesis, Fac. Elect. Eng. Comput. Sci., Univ. Maribor, Maribor, Slovenia, 2018.
[39] (Oct. 22, 2020). *Graphviz Tool*. Accessed: Jul. 6, 2020. [Online]. Available: https://graphviz.org/about/
[40] *mCRL2*. Accessed: Oct. 22, 2020. [Online.] Available:. [Online]. Available: https://www.mcrl2.org/web/user_manual/index.html
[41] J. F. Groote and M. R. Mousavi, *Modeling and Analysis of Communicating Systems*. Boston, MA, USA: MIT Press, 2014.

[42] Inria. *Aldebaran Webpage*. Accessed: Oct. 22, 2020. [Online]. Available: https://cadp.inria.fr/man/aldebaran.html
[43] F. van Ham, H. van de Wetering, and J. J. van Wijk, "Visualization of state transition graphs," in *Proc. IEEE Symp. Inf. Vis. (INFOVIS)*, Oct. 2001, pp. 59–66.
[44] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "On ACTL formulas having linear counterexamples," *J. Comput. Syst. Sci.*, vol. 62, no. 3, pp. 463–515, May 2001.
[45] Z. Xu and W. Zhang, "Linear templates of ACTL formulas with an application to SAT-based verification," *Inf. Process. Lett.*, vol. 127, pp. 6–16, Nov. 2017.

ROK VOGRIN (Member, IEEE) received the M.Sc. degree in telecommunications from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 2018, where he is currently pursuing the Ph.D. degree in electrical engineering.

Since 2019, he has been a Teaching Assistant at the Institute of Electronics and Telecommunications, University of Maribor. His research interests include formal verification of communications protocols and biological systems, as well as network verification.

ROBERT MEOLIC (Member, IEEE) received the M.Sc. degree in computer science and the Ph.D. degree in electrical engineering from the University of Maribor, Slovenia, in 1999 and 2005, respectively.

He has been actively researching various topics within formal methods since he was the coauthor of a paper on binary decision diagrams, which won Second Prize in the 1993 IEEE Region 8 Student Paper Contest. Most of his work is devoted to the development of software tools, mathematical logic, algorithms with binary decision diagrams, formal specification of concurrent systems, and model checking. He is the main author of the tool EST.

TATJANA KAPUS (Member, IEEE) received the M.Sc. and Ph.D. degrees in electrical engineering from the Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia, in 1991 and 1994, respectively.

She is currently a Full Professor at the Faculty of Electrical Engineering and Computer Science, Institute of Electronics and Telecommunications, University of Maribor. She mainly teaches courses on communications networks and protocols. Her research interests include formal methods for specification and verification of reactive systems.