Witness and Counterexample Automata for ACTL

Robert Meolic[1], Alessandro Fantechi[2], and Stefania Gnesi[3]

[1] Faculty of Electrical Engineering and Computer Science
University of Maribor
Maribor, Slovenia
meolic@uni-mb.si

[2] Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Firenze, Italy
fantechi@dsi.unifi.it

[3] ISTI-CNR
Pisa, Italy
gnesi@isti.cnr.it

Abstract. Witnesses and counterexamples produced by model checkers provide a very useful source of diagnostic information. They are usually returned in the form of a single computation path along the model of the system. However, a single computation path is not enough to explain all reasons of a validity or a failure. Our work in this area is motivated by the application of action-based model checking algorithms to the test case generation for models formally specified with a CCS-like process algebra. There, only linear and finite witnesses and counterexamples are useful and for the given formula and model an efficient representation of the set of witnesses (counterexamples) explaining all reasons of validity (failure) is needed. This paper identifies a fragment of an action-based computation tree logic (ACTL), that guarantee linear witnesses and counterexamples. For it, *witness* and *counterexample automata* are introduced, which are finite automata recognizing linear witnesses and counterexamples, respectively. An algorithm for generating such automata is given.

1 Introduction

Witnesses that show why a formula is satisfied and (more often) counterexamples that show why it is not satisfied over a model have been used as useful diagnostic information since the first applications of model checking technology. They are usually returned by model checkers in the form of a computation path. However, only for certain kinds of formulae a computation path is able to explain completely the reason of satisfaction or missed satisfaction. Only recently, a greater interest was raised on the study of the relations between the formulae and their counterexamples, on one side looking for richer forms, such as tree-like counterexamples [4] or proof-like counterexamples [10], on the other side establishing the subsets of the logics whose formulae guarantee linear computation paths as counterexamples which completely explain the failure [1, 12].

Our work in this field has been motivated by another trend that has consolidated in the recent years, that is the usage of counterexamples as an help to generate test cases [6–8, 11, 14, 15]. When testing or simulation does not reach an adequate level of coverage (defined by some code coverage metrics, such as statement coverage and branch coverage) new test cases have to be defined, but the process of manually producing test cases for "corner-case" scenarios is time consuming and error prone. Model checking and counterexamples can help: if we have a model of the system, and we model-check on it a formula expressing "there is no uncovered point", a counterexample returns a computation path with enough information to generate the proper test case. In [7] it is shown that in the adoption of this principle with a "conquer and divide" approach in order to attack the typical state space explosion problem, a more effective option is to have the model checker generating not a single counterexample, but all the counterexamples for the given formula. We refer to [7] for more details, while here we focus on the problem of the efficient representation and generation of the set of counterexamples of a given formula.

An action-based computation tree logic (ACTL[4]) [13] is a version of the branching time temporal logic CTL [3]. ACTL is suitable to express properties of reactive systems whose behaviour is characterized by the actions they perform and whose semantics is defined by means of LTS's. ACTL is adequate with respect to strong bisimulation equivalence, this means that if $p \sim q$, then p and q satisfy the same set of ACTL formulae. To define ACTL an auxiliary logic of actions is introduced. We limit our study to a subset of ACTL that guarantees linear witnesses and counterexamples; we address both witnesses and counterexamples, since one can switch between them using negation. Moreover, we observe that for the purpose of practical applications (e.g. test case discovery) only finite linear witnesses and counterexamples are interesting. Further, we prove that the desired sets of finite linear witnesses and finite linear counterexamples form a regular language and therefore they can be represented as automata, that will be called *witness automaton* and *counterexample automaton*, respectively.

Formal definitions are given in Section 2. In Section 3, we introduce a *viable* class of witnesses and counterexamples for application in the field of test case generation. Moreover, we define witness and counterexample automata as candidates for such viable class. In Section 4 an algorithm to generate witness and counterexample automata is reported and comprehensively explained on examples. Section 5 discusses complexity, implementation, and several directions of possible extension of our work. In Appendix, we show some additional examples of generated automata.

[4] The acronym ACTL is also used to denote the universal fragment of the CTL, whose original name used in [2] was ∀CTL, later its name has changed for easiness of writing, but generating a conflict with the already used name for action-based computation tree logic.

2 Definitions

Definition 1. (*Labelled transition system*)
A *labelled transition system* (LTS) is a quadruple $\mathcal{M} = (S, Act, D, s_0)$, where S is a set of states, Act is a set of observable actions (an unobservable action τ is not in Act), $D \subseteq S \times Act \cup \{\tau\} \times S$ is the transition relation, and $s_0 \in S$ is the initial state.

For $A \subseteq Act$, we let $D_A(s)$ denote the set of successors of the state s reachable by an action from the set A. Moreover, we let $D^\tau_A(s)$ denote $D_{A\cup\{\tau\}}(s)$. If π is a computation path in an LTS, then $\pi(0)$ is the first state on π and $\pi(i{+}1)$ is the state on the path π immediately after the state $\pi(i)$.

Definition 2. (*Action formulae*)
The syntax of action formulae over Act is defined by the following grammar where χ, χ', range over action formulae, and $a \in Act$:

$$\chi ::= a \mid \neg\chi \mid \chi \vee \chi$$

The satisfaction of an action formula χ by an action a, $a \models \chi$, is inductively defined as follows:

$$\begin{array}{ll} a \models b & \text{iff } a = b; \\ a \models \neg\chi & \text{iff } a \not\models \chi; \\ a \models \chi \vee \chi' & \text{iff } a \models \chi \text{ or } a \models \chi' \end{array}$$

We write true for $\alpha \vee \neg\alpha$, where α is some arbitrarily chosen action, and false stands for $\neg$true. Moreover, we will write $\chi \wedge \chi'$ for $\neg(\neg\chi \vee \neg\chi')$. An action formula permits the expression of constraints on the actions that can be observed (along a path or after next step); for instance, $a \vee b$ says that the only possible observations are a or b, while true stands for "all actions are allowed" and false for "no actions can be observed", that is only silent actions can be performed.

Definition 3. (*Action-based computation tree logic*)
The syntax of ACTL is defined by the following grammar, where χ, χ' range over action formulae, $\exists$, $\forall$ are path quantifiers, and $\mathbf{X}$, $\mathbf{U}$ are *next* and *until* operators, respectively:

$$\begin{array}{l} \varphi ::= \text{true} \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists\gamma \mid \forall\gamma \\ \gamma ::= \mathbf{X}_\chi\, \varphi \mid \mathbf{X}_\tau\, \varphi \mid \varphi\, \mathbf{U}_\chi\, \varphi \mid \varphi\, {}_\chi\mathbf{U}_{\chi'}\, \varphi \end{array}$$

Let $\kappa(\chi) = \{a \mid a \models \chi\}$. Being interpreted over an LTS $\mathcal{M} = (S, Act, D, s_0)$ with a total transition relation the satisfaction of a state formula φ by a state s, $s \models_\mathcal{M} \varphi$, and path formula γ by a path π, $\pi \models_\mathcal{M} \gamma$, is inductively defined by:

$$\begin{array}{ll} s \models_\mathcal{M} \text{true} & \text{always;} \\ s \models_\mathcal{M} \neg\varphi & \text{iff } s \not\models_\mathcal{M} \varphi; \\ s \models_\mathcal{M} \varphi \vee \varphi' & \text{iff } s \models_\mathcal{M} \varphi \text{ or } s \models_\mathcal{M} \varphi'; \\ s \models_\mathcal{M} \exists\,\gamma & \text{iff there exists a path } \pi \text{ such that } \pi(0) = s \text{ and } \pi \models_\mathcal{M} \gamma; \\ s \models_\mathcal{M} \forall\,\gamma & \text{iff for all paths } \pi \text{ such that } \pi(0) = s,\ \pi \models_\mathcal{M} \gamma; \\ \pi \models_\mathcal{M} \mathbf{X}_\chi\, \varphi & \text{iff there exists } \pi(1) \text{ such that } \pi(1) \in D_{\kappa(\chi)}(\pi(0)) \text{ and } \pi(1) \models_\mathcal{M} \varphi; \\ \pi \models_\mathcal{M} \mathbf{X}_\tau\, \varphi & \text{iff there exists } \pi(1) \text{ such that } \pi(1) \in D_{\{\tau\}}(\pi(0)) \text{ and } \pi(1) \models_\mathcal{M} \varphi; \end{array}$$

$\pi \models_{\mathcal{M}} \varphi\, {}_{\chi}\mathbf{U}\, \varphi'$ iff there exists $i \geq 0$ such that $\pi(i) \models_{\mathcal{M}} \varphi'$,
and for all $0 \leq j \leq i-1$: $\pi(j) \models_{\mathcal{M}} \varphi$ and $\pi(j+1) \in D^{\tau}_{\kappa(\chi)}(\pi(j))$;

$\pi \models_{\mathcal{M}} \varphi_{\chi}\mathbf{U}_{\chi'}\varphi'$ iff there exists $i \geq 1$ such that
$\pi(0) \models_{\mathcal{M}} \varphi$, $\pi(i) \models_{\mathcal{M}} \varphi'$, $\pi(i) \in D_{\kappa(\chi')}(\pi(i-1))$, and for all $1 \leq j \leq i-1$:
$\pi(j) \models_{\mathcal{M}} \varphi$ and $\pi(j) \in D^{\tau}_{\kappa(\chi)}(\pi(j-1))$.

We write false for $\neg$true and $\varphi \wedge \varphi'$ for $\neg(\neg\varphi \vee \neg\varphi')$. When the transition system is clear from the context, we write $s \models \varphi$ instead of $s \models_{\mathcal{M}} \varphi$. If $a \models \chi$ and $t \models \varphi$ then transition (s, a, t) is called a (χ, φ)-transition. Transitions, which are not (χ, φ)-transitions, are called $\neg(\chi, \varphi)$-transitions. If $s \models \varphi$ we say that φ holds in state s. An ACTL formula φ is satisfied over a given LTS $\mathcal{M}$ ($\mathcal{M} \models \varphi$) iff φ holds in the initial state of $\mathcal{M}$. The satisfaction of ACTL formulae over LTS $\mathcal{M}_f$ having a non-total transition relation (i.e. $\mathcal{M}_f$ contains deadlocked states) is given as follows: let φ be an ACTL formula and $\mathcal{M}'_f$ be an LTS obtained from $\mathcal{M}_f$ by adding τ-loops in all its deadlocked-states; then $\mathcal{M}_f \models \varphi$ if $\mathcal{M}'_f \models \varphi$.

Several useful modalities can be defined, starting from the basic ones:

- $\exists\mathbf{F}\varphi$ for $\exists(\text{true}_{\text{true}}\mathbf{U}\varphi)$, and $\forall\mathbf{F}\varphi$ for $\forall(\text{true}_{\text{true}}\mathbf{U}\varphi)$ (*eventually* operators);
- $<\chi>\varphi$ for $\exists\mathbf{X}_{\chi}\,\varphi$, and $[\chi]\,\varphi$ for $\neg\exists\mathbf{X}_{\chi}\,\neg\varphi$ (Hennessy-Milner modalities[5]);
- $\exists\mathbf{G}\,\varphi$ for $\neg\forall\mathbf{F}\,\neg\varphi$, and $\forall\mathbf{G}\,\varphi$ for $\neg\exists\mathbf{F}\,\neg\varphi$ (*always* operators).

Given a model $\mathcal{M}$ and a formula φ such that $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \varphi$), a witness (counterexample) is a structure $\mathcal{R}$, in relation with $\mathcal{M}$, that completely shows one of the possible reasons why $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \varphi$). A reason why $\mathcal{M} \models \varphi$ will be called *a reason of validity* and a reason why $\mathcal{M} \not\models \varphi$ will be called *a reason of failure*. The type of the relation between $\mathcal{R}$ and $\mathcal{M}$ determines the nature of the witnesses and countererexamples. Linear witnesses and counterexamples are finite or infinite computation paths over $\mathcal{M}$. More complex forms of witnesses and counterexamples [4, 10] are defined as non-linear structures related to the original model $\mathcal{M}$. In our approach, $\mathcal{M}$ is an LTS and φ is an ACTL formula. Further, we formally define linear witnesses and counterexamples for an ACTL formula over an LTS.

Definition 4. (*Linear witness and counterexample for ACTL formula over LTS*) Given an LTS $\mathcal{M}$ and an ACTL formula φ such that $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \varphi$), a linear witness (counterexample) for φ over $\mathcal{M}$ is a sequence of actions that completely shows one of the possible reasons why $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \varphi$).

3 Witness and counterexample automata

In [4], it has been recognized that for practical end-applications not all witnesses and counterexamples are usable. For example, the whole model is always a witness or a counterexample. Following the interpretation of [4], a *viable* class of witnesses (counterexamples) $\mathcal{V}$ meets the criteria of:

[5] In [13], $<\chi>\varphi$ and $[\chi]\,\varphi$ are defined and used instead as the weak version of the diamond and box operators of Hennessy-Milner logic.

- **Completeness**. All reasons of validity (failure) of a given formula over a given model, which are important for the end-application, can be explained by a single witness (counterexample) in $\mathcal{V}$.
- **Intelligibility**. Witnesses (counterexamples) in $\mathcal{V}$ are specific enough to suit the end-application (e.g. simple enough to be analysed by engineers).
- **Effectiveness**. There exist effective algorithms for generating and manipulating witnesses (counterexamples) in $\mathcal{V}$.

Note, that these criteria need to be adapted to the end-application. The test case generation approach in [7] is based on finite linear counterexamples, which are *no longer than necessary*, i.e. they contain only transitions, which contribute to the explanation of a particular reason of failure. Thus, we formally define the viability criteria in the field of test case generation as follows.

Definition 5. (*Viability criteria in the field of test case generation*)
The class of witnesses (counterexamples) $\mathcal{V}$ is a viable class for application in the field of test case generation iff for the given ACTL formula φ and LTS $\mathcal{M}$ there exists a suitable witness (counterexample) in $\mathcal{V}$, which

1. explains all reasons of validity (failure) of φ over $\mathcal{M}$ explainable by finite linear witnesses (counterexamples),
2. is as small as possible, and
3. is computable by an effective algorithm.

Still following [7], we further restrict the approach to the ACTL formulae which guarantee linearity and finiteness, i.e. for which all reasons of validity (failure) can be explained with finite linear witnesses (counterexamples) over all models.

Theorem 1. ACTL formulae of kind φ (ψ) as given by the grammar, when satisfied (not satisfied) over an LTS, guarantee linear witnesses (counterexamples):

$$\varphi ::= \text{true} \mid \neg\psi \mid \varphi \vee \varphi \mid \exists(\text{true}\,{}_{\chi}\mathbf{U}\,\varphi) \mid \exists(\text{true}\,{}_{\chi}\mathbf{U}_{\chi}\,\varphi) \mid \exists\mathbf{X}_{\tau}\,\varphi \mid \exists\mathbf{X}_{\chi}\,\varphi$$
$$\psi ::= \neg\varphi \mid \forall(\psi\,{}_{\chi}\mathbf{U}\,\text{true}) \mid \forall(\psi\,{}_{\chi}\mathbf{U}_{\chi}\,\text{true}) \mid \forall\mathbf{X}_{\tau}\,\text{true} \mid \forall\mathbf{X}_{\tau}\,\psi \mid \forall\mathbf{X}_{\chi}\,\text{true} \mid \forall\mathbf{X}_{\chi}\,\psi$$

Proof. Theorem 1 can be proved by induction on subformulae. The basic cases are formulae true and false, and every path contains sufficient information for explaining them. Due to a short space, we give a proof only for ACTL formulae of kind $\exists(\text{true}\,{}_{\chi}\mathbf{U}_{\chi}\,\varphi)$. Let $\mathcal{M}$ be an LTS, $\gamma = \text{true}\,{}_{\chi}\mathbf{U}_{\chi'}\,\varphi'$, $\varphi = \exists\gamma$, and $\mathcal{M} \models \varphi$. According to Definition 3, the reason of validity is the existence of paths starting in the initial state of $\mathcal{M}$, which satisfy path formula γ. Thus, a witness is a proof that a particular path π satisfies path formula γ. If the path π itself contains sufficient information to show why $\pi \models \gamma$ then π is a linear witness. According to Definition 3 again, for each $\pi \models \gamma$ there exists $i \geq 1$ such that $\pi(i) \models \varphi'$ and $\pi(i) \in D_{\kappa(\chi')}(\pi(i-1))$, and for all $0 \leq j \leq i-1 : \pi(j) \in D^{\tau}_{\kappa(\chi)}(\pi(j-1))$. Now, the path from the first state until state $\pi(i)$ needs no additional explanation. We must only show $\pi(i) \models \varphi'$. But, due to the induction hypothesis, subformula φ' guarantees linear witnesses and therefore the suffix of the path from state $\pi(i)$

contains sufficient information for explaining $\pi(i) \models \varphi'$. Hence, any path π such that $\pi \models \gamma$ is a linear witness.

Some formulae with derived modalities can also be used in the presented approach. Indeed, formulae $\exists \mathbf{F}\, \varphi$ and $<\chi>\, \varphi$ guarantee linear witnesses, while formulae $\forall \mathbf{G}\, \psi$ and $[\,\chi\,]\, \psi$ guarantee linear counterexamples.

Theorem 1 assures linearity but not finiteness. In fact, all ACTL formulae of the given sublogic, except those of kind $\forall(\psi\, {}_{\chi}\mathbf{U}_{\chi}\, \mathrm{true})$, guarantee also finiteness. Therefore, we exclude formulae of this kind from this approach. Moreover, formulae of kind $\forall(\psi\, {}_{\chi}\mathbf{U}\, \mathrm{true})$ always hold and have no counterexamples. For all other kind of formuale of the given sublogic and an LTS $\mathcal{M}$ with a total transition relation, we define in a constructive way $\mathcal{V}$-witnesses and $\mathcal{V}$-counterexamples, respectively.

Definition 6. (*$\mathcal{V}$-witness and $\mathcal{V}$-counterexample*)

(a) A $\mathcal{V}$-witness for ACTL formula true is a path consisting of only one state and no transitions.

(b) A path π in $\mathcal{M}$ is a $\mathcal{V}$-witness for ACTL formula $\varphi = \neg\psi$ iff π is a $\mathcal{V}$-counterexample for ACTL formula ψ.

(c) A path π in $\mathcal{M}$ is a $\mathcal{V}$-witness for ACTL formula $\varphi = \varphi_1 \vee \varphi_2$ iff π is a $\mathcal{V}$-witness for φ_1 or π is a $\mathcal{V}$-witness for φ_2.

(d) A path π in $\mathcal{M}$ is a $\mathcal{V}$-witness for ACTL formula $\varphi = \exists(\mathrm{true}\, {}_{\chi}\mathbf{U}\, \varphi')$ iff there exists $i \geq 0$ such that suffix of π starting in $\pi(i)$ is a $\mathcal{V}$-witness for ACTL formula φ', and for all $0 \leq j \leq i-1$: $\pi(j+1) \in D^{\tau}_{\kappa(\chi)}(\pi(j))$.

(e) A path π in $\mathcal{M}$ is a $\mathcal{V}$-witness for ACTL formula $\varphi = \exists(\mathrm{true}\, {}_{\chi}\mathbf{U}_{\chi'}\, \varphi')$ iff there exists $i \geq 1$ such that $\pi(i) \in D_{\kappa(\chi')}(\pi(i-1))$ and suffix of π starting in $\pi(i)$ is a $\mathcal{V}$-witness for ACTL formula φ', and for all $1 \leq j \leq i-1$: $\pi(j) \in D^{\tau}_{\kappa(\chi)}(\pi(j-1))$.

(f) A path π in $\mathcal{M}$ is a $\mathcal{V}$-witness for ACTL formula $\varphi = \exists \mathbf{X}_{\tau}\, \varphi'$ iff $\pi(1) \in D_{\{\tau\}}(\pi(0))$ and suffix of π starting in $\pi(1)$ is a $\mathcal{V}$-witness for ACTL formula φ'.

(g) A path π in $\mathcal{M}$ is a $\mathcal{V}$-witness for ACTL formula $\varphi = \exists \mathbf{X}_{\chi}\, \varphi'$ iff $\pi(1) \in D_{\kappa(\chi)}(\pi(0))$ and suffix of π starting in $\pi(1)$ is a $\mathcal{V}$-witness for ACTL formula φ'.

(h) A path π in $\mathcal{M}$ is a $\mathcal{V}$-counterexample for ACTL formula $\psi = \neg\varphi$ iff π is a $\mathcal{V}$-witness for ACTL formula φ.

(i) A path π in $\mathcal{M}$ is a $\mathcal{V}$-counterexample for ACTL formula $\psi = \forall \mathbf{X}_{\tau}\, \mathrm{true}$ iff π contains only two states and $\pi(1) \notin D_{\{\tau\}}(\pi(0))$.

(j) A path π in $\mathcal{M}$ is a $\mathcal{V}$-counterexample for ACTL formula $\psi = \forall \mathbf{X}_{\tau}\, \psi'$ iff π contains only two states and $\pi(1) \notin D_{\{\tau\}}(\pi(0))$, or $\pi(1) \in D_{\{\tau\}}(\pi(0))$ and suffix of π starting in $\pi(1)$ is a $\mathcal{V}$-counterexample for ACTL formula φ'.

(k) A path π in $\mathcal{M}$ is a $\mathcal{V}$-counterexample for ACTL formula $\psi = \forall \mathbf{X}_{\chi}\, \mathrm{true}$ iff π contains only two states and $\pi(1) \notin D_{\kappa(\chi)}(\pi(0))$.

(l) A path π in $\mathcal{M}$ is a $\mathcal{V}$-counterexample for ACTL formula $\psi = \forall \mathbf{X}_{\chi}\, \psi'$ iff π contains only two states and $\pi(1) \notin D_{\kappa(\chi)}(\pi(0))$, or $\pi(1) \in D_{\kappa(\chi)}(\pi(0))$ and suffix of π starting in $\pi(1)$ is a $\mathcal{V}$-counterexample for ACTL formula φ'.

It is straightforward to show that all $\mathcal{V}$-witnesses ($\mathcal{V}$-counterexamples) are finite linear witnesses (counterexamples). The following theorem shows that they are suitable for our approach.

Theorem 2. Let π be a finite linear witness (counterexample) explaining a reason of validity (failure) of an ACTL formula φ (ψ) over an LTS $\mathcal{M}$. Then, there exists $\mathcal{V}$-witness ($\mathcal{V}$-counterexample), which shows that $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \psi$).

Proof. We claim, that the maximal (not necessary proper and therefore always existing) prefix of π, which shows that $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \psi$) is a $\mathcal{V}$-witness ($\mathcal{V}$-counterexample). To show this, we observe finite linear witnesses (counterexamples), which are not $\mathcal{V}$-witnesses ($\mathcal{V}$-counterexamples) and prove, that a proper prefix of them exists, which explain $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \psi$). We omit the details of the proof due to the lack of space.

Further, we show that we can characterize not only a single $\mathcal{V}$-witness and $\mathcal{V}$-counterexample, but also the set of *all* the possible ones. Actually, the number of the possible $\mathcal{V}$-witnesses and $\mathcal{V}$-counterexamples may be infinite and we are interested to a finite representation of them all.

Theorem 3. Let $\mathcal{M}$ be a finite state LTS and φ (ψ) an ACTL formula such that $\mathcal{M} \models \varphi$ ($\mathcal{M} \not\models \psi$). Then, there exists a finite-state automaton which recognizes all $\mathcal{V}$-witnesses ($\mathcal{V}$-counterexamples) for formula φ (ψ) over $\mathcal{M}$.

Proof. In order to check whether a path is a $\mathcal{V}$-witness ($\mathcal{V}$-counterexample), we just need to see whether it is a path over $\mathcal{M}$ and if it has the form given in Definition 6. Since the characterizations given in Definition 6 are given with a single right recursion, they can be expressed as a regular grammar. The language of $\mathcal{V}$-witnesses ($\mathcal{V}$-counterexamples) is the intersection of the regular language recognized by this grammar and the regular language of the finite paths over $\mathcal{M}$. Hence, it is a regular language and can be recognized by a finite state automaton.

Definition 7. (*Witness and counterexample automata*)
A witness (counterexample) automaton for an LTS $\mathcal{M}$ and an ACTL formula φ is an automaton which recognizes the language of all $\mathcal{V}$-witnesses ($\mathcal{V}$-counterexamples) of φ over $\mathcal{M}$.

Now, only an effective algorithm for generation of witness and counterexample automata is missing to fit the viability criteria in Definition 5.

4 Implementation

We present here an elegant recursive algorithm that, given a LTS and a formula φ from the subset of ACTL given in Theorem 1, generates the witness or counterexample automaton WCA for the formula φ over the given LTS. If the formula φ holds in the initial state of LTS, the generated WCA is a witness automaton, otherwise, it is a counterexample automaton. The algorithm is a direct implementation of the definitions of $\mathcal{V}$-witnesses and $\mathcal{V}$-counterexamples and it is given in a C-like pseudocode. It consists of the main function *WCAgenerator*, two auxiliary functions *generate* and *conbuild* (Fig. 1), and functions processing ACTL formulae (Fig. 2, 3). For the purpose of further work, the part for generation of counterexample automata is extended with formulae of type $\forall(\psi\ {}_{\chi}\mathbf{U}_{\chi}$ true). For them, $\mathcal{V}$-counterexamples have not been defined. Paths recognized by the obtained automaton for this kind of ACTL formulae explain only those reasons of failure which can be explained with finite linear counterexamples. Due to the lack of space, we are not able to discuss details of this feature.

The algorithm proceeds by visiting a portion of the state space of LTS. The visit is guided by the structural analysis of the formula itself, hence it is terminated when the leaves of the formula are reached. In ACTL, leaves can only be the formula true or the formula false. LTS is unfolded if a sequence of subformulae of φ matches with a loop in it. The visit is implemented by a

```
WCAgenerator (LTS, $\varphi$) {
  forall subformulae $\varphi'$ of formula $\varphi$ {
    create subset of LTS states $S_{\varphi'}$ in which formula $\varphi'$ holds;
    create empty relation $R_{\varphi'}$;
  }
  let $s$ be the initial state of LTS;
  create empty automaton WCA;
  create the initial state $t$ in WCA;
  if ($s \in S_{\varphi}$) generate (LTS, $\varphi$, $s$, $t$, witness);
    else generate (LTS, $\varphi$, $s$, $t$, counterexample);
}

generate (LTS, $\varphi$, $s$, $t$, type) {
  if (type == null) {
    add $t$ to the set of WCA final states;
  } else {
    add the pair $(t, s)$ to $R_{\varphi}$ ;
    if (type == witness) WAgen (LTS, $\varphi$, $s$, $t$);
    if (type == counterexample) CAgen (LTS, $\varphi$, $s$, $t$);
  }
}

conbuild (LTS, $\varphi$, $a$, $s'$, $t$, type) {
  if ((type == null) || (there is no state related to state $s'$ in $R_{\varphi}$)) {
    create a new state $t'$ in WCA;
    if not exists, add the transition $(t, a, t')$ to WCA;
    generate (LTS, $\varphi$, $s'$, $t'$, type);
  } else {
    let $t'$ be a state in WCA related to state $s'$ in $R_{\varphi}$;
    if not exists, add the transition $(t, a, t')$ to WCA;
  }
}
```

Fig. 1. Main function and two auxiliary functions of the algorithm

depth-first search by recursion, and hence it has to be remembered which states of LTS have already been visited with a particular subformula of φ.

The algorithm is further explained in details using a simple LTS and two simple ACTL formulae (Fig. 4). It starts in function *WCAgenerator*. The first action is actually a call to a model checker, which computes S and initializes R for all subformulae of φ. S contains for each subformula the subset of states that satisfy the subformula. Thus, the algorithm takes for granted the information

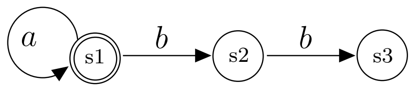
```
WAgen (LTS, $\varphi$, $s$, $t$) {
  case ($\varphi$ == false) or ($\varphi$ == true):
    generate (LTS, $\varphi$, $s$, $t$, null);
    break;
  case $\varphi == \neg\varphi'$:
    generate (LTS, $\varphi'$, $s$, $t$, counterexample);
    break;
  case $\varphi == \varphi_1 \lor \varphi_2$:
    if ($s \in S_{\varphi_1}$) generate (LTS, $\varphi_1$, $s$, $t$, witness);
    if ($s \in S_{\varphi_2}$) generate (LTS, $\varphi_2$, $s$, $t$, witness);
    break;
  case $\varphi == \exists\mathbf{X}_\chi\, \varphi'$:
    WAbuild (LTS, $\varphi$, $s$, $t$, false, false, $\chi$, $\varphi'$);
    break;
  case $\varphi == \exists\mathbf{X}_\tau\, \varphi'$:
    if ($s$ is a deadlocked state) generate (LTS, $\varphi'$, $s$, $t$, witness);
      else WAbuild (LTS, $\varphi$, $s$, $t$, false, false, $\tau$, $\varphi'$);
    break;
  case $\varphi == \exists(\text{true}\,{}_\chi\mathbf{U}\, \varphi')$:
    if ($s \in S_{\varphi'}$) generate (LTS, $\varphi'$, $s$, $t$, witness);
    WAbuild (LTS, $\varphi$, $s$, $t$, $(\chi \lor \tau)$, true, $(\chi \lor \tau)$, $\varphi'$);
    break;
  case $\varphi == \exists(\text{true}\,{}_\chi\mathbf{U}_{\chi'}\, \varphi')$:
    WAbuild (LTS, $\varphi$, $s$, $t$, $(\chi \lor \tau)$, true, $\chi'$, $\varphi'$);
    break;
}
WAbuild (LTS, $\varphi$, $s$, $t$, $\chi_1$, $\varphi_1$, $\chi_2$, $\varphi_2$) {
  let $\delta_1$ be the set of $(\chi_1, \varphi_1)$-transitions from $s$;
  forall transitions $(s, a, s') \in \delta_1$ {
    if ($s' \in S_\varphi$) conbuild (LTS, $\varphi$, $a$, $s'$, $t$, witness);
  }
  let $\delta_2$ be the set of $(\chi_2, \varphi_2)$-transitions from $s$;
  forall transitions $(s, a, s') \in \delta_2$ {
    conbuild (LTS, $\varphi_2$, $a$, $s'$, $t$, witness);
  }
}
```

Fig. 2. The part of the algorithm, which generates witness automaton

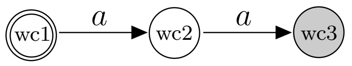
about validity of the subformulae on each state. R is a relation implemented as a set of pairs, where for each state of the LTS visited with a particular subformula the related state in WCA is stored. Variables WCA, S, and R are global, all others are local.

```
CAgen (LTS, $\varphi$, $s$, $t$) {
  case ($\varphi$ == false) or ($\varphi$ == true):
    generate (LTS, $\varphi$, $s$, $t$, null);
    break;
  case $\varphi == \neg\varphi'$:
    generate (LTS, $\varphi'$, $s$, $t$, witness);
    break;
  case $\varphi == \varphi_1 \wedge \varphi_2$:
    if ($s \notin S_{\varphi_1}$) generate (LTS, $\varphi_1$, $s$, $t$, counterexample);
    if ($s \notin S_{\varphi_2}$) generate (LTS, $\varphi_2$, $s$, $t$, counterexample);
    break;
  case $\varphi == \forall\mathbf{X}_\chi\ \varphi'$:
    if ($s$ is a deadlocked state) generate (LTS, $\varphi$, $s$, $t$, null);
      else CAbuild (LTS, $\varphi$, $s$, $t$, false, false, $\chi$, $\varphi'$);
    break;
  case $\varphi == \forall\mathbf{X}_\tau\ \varphi'$:
    if ($s$ is a deadlocked state) generate (LTS, $\varphi'$, $s$, $t$, counterexample);
      else CAbuild (LTS, $\varphi$, $s$, $t$, false, false, $\tau$, $\varphi'$);
    break;
  case $\varphi == \forall(\varphi'\ {}_\chi\mathbf{U}_{\chi'}$ true):
    if ($s$ is a deadlocked state) generate (LTS, $\varphi$, $s$, $t$, null);
    else {
      if ($s \notin S_{\varphi'}$) generate (LTS, $\varphi'$, $s$, $t$, counterexample);
      CAbuild (LTS, $\varphi$, $s$, $t$, $(\chi \vee \tau)$, $\varphi'$, $\chi'$, true);
    }
    break;
}
CAbuild (LTS, $\varphi$, $s$, $t$, $\chi_1$, $\varphi_1$, $\chi_2$, $\varphi_2$) {
  let $\delta_1$ be the set of $(\chi_1, \varphi_1)$-transitions from $s$;
  forall transitions $(s, a, s') \in \delta_1$ do {
    if ($s' \notin S_\varphi$) conbuild (LTS, $\varphi$, $a$, $s'$, $t$, counterexample);
  }
  let $\delta_2$ be the set of $\neg(\chi_2, \varphi_2)$-trans. from $s$ which are $\neg(\chi_1, \varphi_1)$-trans.;
  forall transitions $(s, a, s') \in \delta_2$ {
    if ($a \not\models \chi_1 \wedge a \not\models \chi_2$) conbuild (LTS, $\varphi$, $a$, $s'$, $t$, null);
    if ($a \models \chi_1$) conbuild (LTS, $\varphi_1$, $a$, $s'$, $t$, counterexample);
    if ($a \models \chi_2$) conbuild (LTS, $\varphi_2$, $a$, $s'$, $t$, counterexample);
  }
}
```

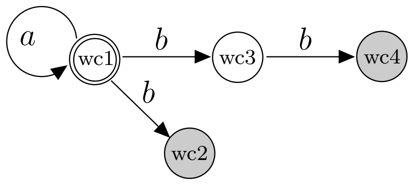
Fig. 3. The part of the algorithm, which generates counterexample automaton



(a) The model: $S = a.S + b.b.\mathrm{nil}$



(b) Witness automaton for $\exists\mathbf{X}_a\, \exists\mathbf{X}_a$ true



(c) Witness automaton for $\exists\mathbf{F}\, \exists\mathbf{X}_b$ true

Fig. 4. Two examples of generated witness automata. For the second ACTL formula, the recognized witnesses are *b*, *bb*, *ab*, *abb*, *aab*, *aabb*, *aaab*, *aaabb*, etc.

Here is a program trace for the example in Fig. 4b.

```
ACTL model checking on S
EX{a} EX{a} true ==> TRUE
@@ WCAgenerator: created empty witness automaton WC1S
@@ WCAgenerator: created initial state wc1
@@ generate: starting formula ‘EX{a} EX{a} true’ for (s=s1, t=wc1)
@@   generate: added pair (wc1,s1) to R for the current formula
@@   WAbuild: chosen transition s1-a->s1 from delta2
@@   conbuild: created state wc2, created transition wc1-a->wc2
@@   generate: starting formula ‘EX{a} true’ for (s=s1, t=wc2)
@@     generate: added pair (wc2,s1) to R for the current formula
@@     WAbuild: chosen transition s1-a->s1 from delta2
@@     conbuild: created state wc3, created transition wc2-a->wc3
@@     generate: starting formula ‘true’ for (s=s1, t=wc3)
@@       generate: added pair (wc3,s1) to R for the current formula
@@       generate: state wc3 marked as final
@@ Witness automaton has been constructed.
```

After an ACTL model checker determines that formula $\exists\mathbf{X}_a\, \exists\mathbf{X}_a$ true holds in the initial state `s1` of `S`, a generation of the witness automaton is started. A new automaton `WC1S` and its initial state `wc1` are created. Function *generate* makes initial states of `S` and `WC1S` to be related for the formula $\exists\mathbf{X}_a\, \exists\mathbf{X}_a$ true. Then, function *WAgen* is started. The outermost operator is $\exists\mathbf{X}_a$ and therefore function *WAbuild* is called with parameters $\chi_1 = \text{false}$, $\varphi_1 = \text{false}$, $\chi_2 = a$, and $\varphi_2 = \exists\mathbf{X}_a$ true. Set δ_1 is empty, because there is no (χ_1, φ_1)-transition. Set δ_2 contains only the transition (`s1`, a, `s1`). State `s1` has not been visited with formula $\exists\mathbf{X}_a$ true, yet, and therefore there is no state in `WC1S` related to it. Function *conbuild* creates a new state `wc2` and the transition (`wc1`, a, `wc2`) in

WC1S. Then, function *generate* is recursively called for the formula $\exists\mathbf{X}_a$ true. In this call, subformula φ_2 = true. Again, set δ_1 is empty, while set δ_2 contains the transition (s1, a, s1). Because state s1 has not been visited with formula true, yet, function *conbuild* creates a new state wc3 and transition (wc2, a, wc3). Further, state wc3 is marked as final and the recursive calls end.

The usage of relation R is better shown on a program trace for ACTL formula $\exists\mathbf{F}\,\exists\mathbf{X}_b$ true. Actually, this is an abbreviation of formula $\exists(\text{true}\,{}_{\text{true}}\mathbf{U}\,\exists\mathbf{X}_b\,\text{true})$.

```
ACTL model checking on S
EF EX{b?} true ==> TRUE
@@ WCAgenerator: created empty witness automaton WC2S
@@ WCAgenerator: created initial state wc1
@@ generate: starting formula 'EF EX{b?} true' for (s=s1, t=wc1)
@@   generate: added pair (wc1,s1) to R for the current formula
@@   generate: starting formula 'EX{b?} true' for (s=s1, t=wc1)
@@     generate: added pair (wc1,s1) to R for the current formula
@@     WAbuild: chosen transition s1-b?->s2 from delta2
@@     conbuild: created state wc2, created transition wc1-b?->wc2
@@     generate: starting formula 'true' for (s=s2, t=wc2)
@@       generate: added pair (wc2,s2) to R for the current formula
@@       generate: state wc2 marked as final
@@   WAbuild: chosen transition s1-b?->s2 from delta1
@@   conbuild: created state wc3, created transition wc1-b?->wc3
@@   generate: starting formula 'EF EX{b?} true' for (s=s2, t=wc3)
@@     generate: added pair (wc3,s2) to R for the current formula
@@     generate: starting formula 'EX{b?} true' for (s=s2, t=wc3)
@@       generate: added pair (wc3,s2) to R for the current formula
@@       WAbuild: chosen transition s2-b?->s3 from delta2
@@       conbuild: created state wc4, created transition wc3-b?->wc4
@@       generate: starting formula 'true' for (s=s3, t=wc4)
@@         generate: added pair (wc4,s3) to R for the current formula
@@         generate: state wc4 marked as final
@@   WAbuild: chosen transition s1-a?->s1 from delta1
@@   conbuild: created transition wc1-a?->wc1
@@   WAbuild: chosen transition s1-b?->s2 from delta2
@@   conbuild: transition wc1-b?->wc3 exists
@@   WAbuild: chosen transition s1-a?->s1 from delta2
@@   conbuild: transition wc1-a?->wc1 exists
@@ Witness automaton has been constructed.
```

Because subformula $\exists\mathbf{X}_b$ true holds in the initial state of S, first a $\mathcal{V}$-witness for it is generated. State wc2 and transition (wc1, b, wc2) are created. Then, function *WAgen* continues and calls function *WAbuild* with parameters χ_1 = true, φ_1 = true, χ_2 = true, and $\varphi_2 = \exists\mathbf{X}_b$ true. Set δ_1 and set δ_2 both contain transitions (s1, a, s1) and (s1, b, s2). From the set δ_1 the transition with action b is chosen first. Formula $\exists\mathbf{F}\,\exists\mathbf{X}_b$ true has been already visited in state s1, but not in state s2. Therefore, state wc3 and transition (wc1, b, wc3) are created. Afterwards, the algorithm continues in the state s2. State wc4 and transition (wc3, b, wc4) are created. Because subformula true has been reached, state wc4 is marked as final and the path is finished. Now, function *WAbuild* must also process transition (`s1`, a, `s1`) from the set δ_1. State `s1` has been visited with formula $\exists\mathbf{F}\ \exists\mathbf{X}_b$ true before, thus a new state is not created. State `s1` is related to state `wc1` in relation $R_{\exists\mathbf{F}\ \exists\mathbf{X}_b\ \mathrm{true}}$, therefore transition (`wc1`, a, `wc1`) is created without further recursive calls. This was the last transition in the set δ_1 and function *WAbuild* continues with processsing of the transitions from the set δ_2. However, a corresponding transitions already exist in the witness automaton and further recursive calls are also not needed.

5 Discussion

The algorithm for witness and counterexample automata generation basically works by following the given LTS and using unfolding when necessary, with an unfolding depth of at most the length of the formula. Therefore, the complexity is not higher than the size of the LTS (states and transitions) times the length of the formula. This is exactly the same complexity of an explicit model checking algorithm which has to be employed to compute the labeling of the LTS.

We have implemented the algorithm as an extension of a BDD-based ACTL model checker. Although LTSs are represented by BDDs and BDD-based functions are used for navigating the LTS, the algorithm indirectly still involves an implicit enumeration in functions *WAbuild* and *CAbuild*, where transitions are chosen from δ_1 and δ_2 one by one and then for each next state on the path a new recursive call is made. An open question remains whether a more efficient symbolic algorithm exists. The examples given in Appendix show that the resulting automata may contain redundancy. Some equal paths may be presented more than once. This is not very disturbing and among other reasons it appears because the program does not identify two semantic equal subformulae as the same one. For example, in the witness automaton generated for the ACTL formula $(\exists\mathbf{X}_\chi\,\varphi) \vee (\exists\mathbf{X}_\chi\,\varphi)$, all paths are doubled.

A result which is related to our work is the definition of more expressive *tree-like* counterexamples for Kripke Structures and CTL; such counterexamples are used as a support to guide a refinement technique [4]. The main difference with respect to our approach is that a tree-like counterexample is in its entirety a proof that the formula is not satisfied. Our counterexample automaton gives instead the set of linear counterexamples, each of which can be taken separately as a traditional counterexample. An evolution of tree-like counterexamples is represented by *proof-like* counterexamples [10], used to extract proofs for the non satisfiability of a formula over a model. Closer to our approach is the multiple counterexamples generation of [5, 9], which generates all the counterexamples up to a given length, expressed as a single counterexample trace annotated with possible values of binary variables.

There are some possible directions for further work. We have considered only finite witnesses and counterexamples, which are the ones suitable to be used as actual test cases (see [7]). Having more rich notions of acceptance than linear languages could provide the possibility of characterizing sets of more informative witnesses and counterexamples. In order to deal with infinite counterexamples and witnesses the same approach can be followed, for example, using Büchi automata for recognizing a language of infinite words. In this way, if the transition relation is total and if witnesses and counterexamples are extended to become infinite paths, our work becomes adequate to the work presented for CTL in [1].

An interesting extension of the given algorithm would be a generation of non-linear forms of witnesses and counterexamples. The core of the algorithm are functions *WAbuild* and *CAbuild*. We implemented them in a more general form w.r.t. what needed in this approach. For example, function *WAbuild* will also process ACTL formula $\exists(\varphi_1\, {}_{\chi}\mathbf{U}_{\chi'}\, \varphi_2)$, where parameter φ_1 is not just a simple formula true or false, although a witness for this formula is not always a linear computation path. The algorithm will produce an automaton recognizing linear witnesses and the main paths (sometimes referred as backbones) of non-linear witnesses. There will be no extra information given about which recognized witness completely explains the validity and which one is only a main path of a non-linear witness. Such general implementation allows extensions. If parameter φ_1 is not a simple formula true or false and if an explanation of validity is added to all states on the main path, we get richer non-linear forms of witnesses. Thus, the given algorithm can serve also as a basis for generation of tree-like witnesses and counterexamples. Note that for the formulae which guarantee linearity and finiteness of witnesses (counterexamples) the presented witness and counterexample automata explain all reasons of validity (failure) over a given model and thus they are comparable to the tree-like witnesses and counterexamples, respectively.

6 Conclusions

We have defined witness and counterexample automata, which are intended to be used in the field of test case generation. These automata recognize $\mathcal{V}$-witnesses and $\mathcal{V}$-counterexamples which are finite linear witnesses and counterexamples for a given formula over a given LTS. The main result of the paper is the algorithm for generating witness and counterexample automata for a given LTS and a given ACTL formula from a subset of ACTL formulae which guarantee finite linear witnesses and counterexamples. The algorithm has been implemented and a stand-alone demo application has been made available online on `http://fmt.isti.cnr.it/WCA/`.

It seems reasonable that the given approach works as well with a state based formalism (such as Kripke structures) and a state based temporal logic (such as CTL). This needs to be verified: the very definition of witnesses, counterexamples, and automata recognizing them is actually highly sensitive to the logic used and to the assumptions on the models.

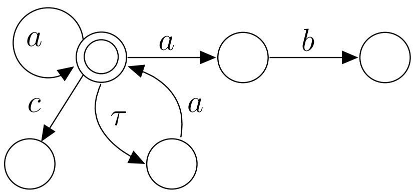
Acknowledgements
This work has been partially supported by Italian MIUR PRIN 2002 COVER project. The first author has been partially supported by a grant from Government of Italy through Italian Cultural Institute in Slovenia. We wish to thank

Tatjana Kapus for many discussions on the topics of the paper and her help in eliminating mistakes. Also, we wish to thank Gianluca Trentanni for his help in the realization of the on-line demo application.

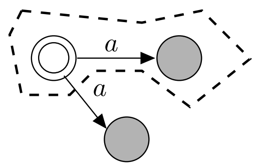
References

1. F. Buccafurri, T. Eiter, G. Gottlob, N. Leone. On ACTL formulas having linear counterexamples. Journal of computer and syst. sciences, 62(3), 2001, pp. 463–515.
2. E. M. Clarke, O. Grumberg, D. E. Long. Model Checking and Abstraction ACM Transaction on Programming Languages and Systems, (16)5, 1994, pp. 1512-1542.
3. E. M. Clarke, E. A. Emerson, A. P. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications. ACM Transaction on Programming Languages and Systems, 8(2), 1986, pp. 244–263.
4. E. M. Clarke, S. Jha, Y. Lu, H. Veith. Tree-like Counterexamples in Model Checking. In *17th IEEE Symp. on Logic in Computer Science (LICS)*, 2002, pp. 19–29.
5. F. Copty, A. Irron, O. Weissberg, N. Kropp, G. Kamhi. Efficient Debugging in a Formal Verification Environment. In *Conf. On Correct Hardware Design and Verification Methods (CHARME)*, LNCS 2144, 2001, pp. 275–292.
6. A. Časar, Z. Brezočnik, T. Kapus. Exploiting Symbolic Model Checking for Sensing Stuck-at Faults in Digital Circuits. Informacije MIDEM, 32(3), 2002, pp. 171–180.
7. A. Fantechi, S. Gnesi, A. Maggiore. Enhancing test coverage by back-tracing model-checker counterexamples. In *Int. Workshop on Test and Analysis of Component Based Syst. (TACOS)*, 2004, to appear in Electronic Notes in Theoretical Computer Science.
8. D. Geist, M. Farkas, A. Landver, Y. Lichenstein, S. Ur, Y. Wolfsthal. Coverage-Directed Test Generation Using Symbolic Techniques. In *First Int. Conf. on Formal Method in Computer-Aided Design (FMCAD)*, LNCS 1166, 1996, pp. 143-158.
9. M. Glusman, G. Kamhi, S. Mador-Heim, R. Fraer, M. Vardi. Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. In *Tools and Algorithms for the construction and analysis of syst. (TACAS)*, LNCS 2619, 2003, pp. 176-191.
10. A. Gurfinkel, M. Chechik. Proof-Like Counter-Examples. In *Tools and Algorithms for the construction and analysis of syst. (TACAS)*, LNCS 2619, 2003, pp. 160-175.
11. P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, J. Long. Smart Simulation Using Collaborative Formal and Simulation Engines. In *Int. Conf. on Computer Aided Design (ICCAD)*, 2000.
12. M. Maidl. The Common Fragment of CTL and LTL. In *Proc. 41th Symp. on Foundations of Computer Science (FOCS)*, pp. 643-652, 2000.
13. R. De Nicola, F.W. Vaandrager. Actions versus State Based Logics for Transition Systems. Proc. Ecole de Printemps on Semantics of Concurrency, Lecture Notes in Computer Science, vol. 469, 1990, pp. 407-419.
14. G. Ratzaby, S. Ur, Y. Wolfsthal, Coverability Analysis Using Symbolic Model Checking. In *Conf. On Correct Hardware Design and Verification Methods (CHARME)*, LNCS 2144, 2001.
15. G. Ratsaby, B. Sterin, S. Ur. Improvements in Coverabiliy Analysis. In *Int. Symp. of Formal Methods Europe (FME)*, LNCS 2391, 2002.

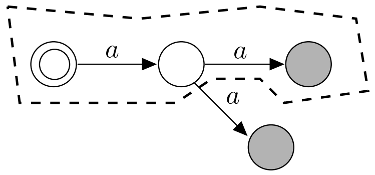
Appendix



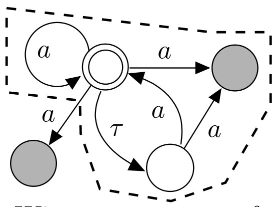
(a) The model: $S = a.S + \tau.a.S + a.b.\text{nil} + c.\text{nil}$



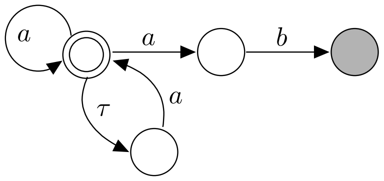
(b) Witness automaton for $\exists \mathbf{X}_a$ true



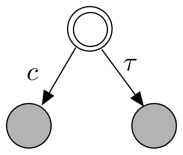
(c) Witness automaton for $\exists \mathbf{X}_a \, \exists \mathbf{X}_a$ true



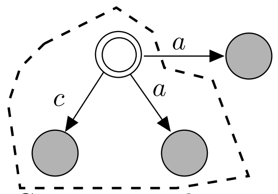
(d) Witness automaton for $\exists \mathbf{F} \, \exists \mathbf{X}_a$ true



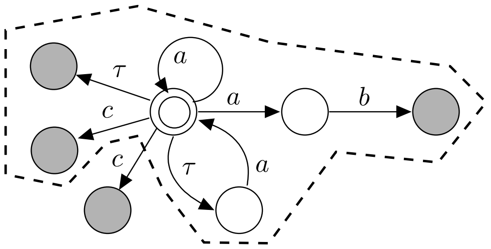
(e) Witness automaton for $\exists(\text{true} \, {}_a\mathbf{U}_b \, \text{true})$



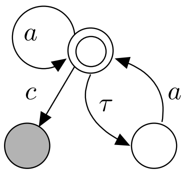
(f) Counterexample automaton for $\forall \mathbf{X}_a$ true



(g) Counterexample automaton for $\forall \mathbf{X}_\tau \, \forall \mathbf{X}_a$ true



(h) Counterexample automaton for $\forall \mathbf{G} \, \forall \mathbf{X}_a$ true



(i) An automaton generated for $\forall(\text{true} \, {}_a\mathbf{U}_b \, \text{true})$

NOTE: Automata in dashed polygons are obtained by a minimization.