Tree-Like Counterexamples in Model Checking *

Edmund Clarke
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
edmund.clarke@cs.cmu.edu

Somesh Jha
Department of Computer Sciences
University of Wisconsin
Madison, WI 53706, USA
sjha@cs.wisc.edu

Yuan Lu
Network Switch Department
Broadcom COM
San Jose, CA 95134, USA
ylu@broadcom.com

Helmut Veith
Institut für Informationssysteme
Technische Universität Wien
A-1040 Vienna, Austria
veith@dbai.tuwien.ac.at

Abstract

Counterexamples for specification violations provide engineers with important debugging information. Although counterexamples are considered one of the main advantages of model checking, state-of the art model checkers are restricted to relatively simple counterexamples, and surprisingly little research effort has been put into counterexamples. In this paper, we introduce a new general framework for counterexamples. The paper has three main contributions: (i) We determine the general form of ACTL counterexamples. To this end, we investigate the notion of counterexample and show that a large class of temporal logics beyond ACTL admits counterexamples with a simple tree-like *transition relation. We show that the existence of tree-like counterexamples is related to a universal fragment of extended branching time logic based on $\omega-$regular temporal operators. (ii) We present new symbolic algorithms to generate tree-like counterexamples for ACTL specifications. (iii) Based on tree-like counterexamples we extend the abstraction refinement methodology developed recently by Clarke* et al. *(CAV'2000) to full ACTL. This demonstrates the conceptual simplicity and elegance of tree-like counterexamples.*

*This work was supported by the SRC under contract No. 99-TJ-684, the GSRC, the NSF under Grant No. CCR-9803774, the Austrian Science Fund Project N Z29-INF, and the EU Research and Training Network GAMES. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, GSRC, NSF, or the United Stated Government.

1. Introduction

When a specification is violated, it is important for engineers to have access to an example where this violation occurs. In the model checking literature, the possibility of automatically generating counterexamples has always been considered one of the major advantages of model checking in comparison to methods such as theorem proving. State-of the art model checkers, however, are restricted to counterexamples of a relatively simple non-branching structure, i.e., finite or infinite paths. We will call such counterexamples *linear counterexamples*.

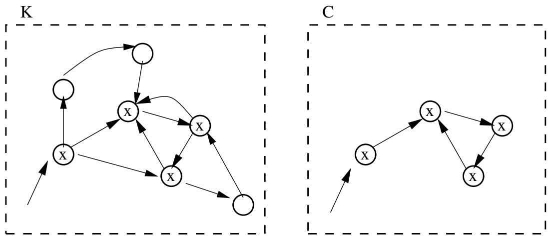


Figure 1. A linear counterexample for $\mathbf{AF}\,\neg x$

Example 1.1 *The ACTL specification* $\mathbf{AF}\neg x$ *denotes "*On all paths, x is violated sometime in the future.*" If the specification* $\mathbf{AF}\neg x$ *is violated, then there exists an infinite path where* x *always holds. This path is called a counterexample of* $\mathbf{AF}\neg x$.

Figure 1 shows an example of a Kripke structure K *where* $\mathbf{AF}\neg x$ *is violated, and a counterexample Kripke structure* C. C *describes an infinite path witnessing* $\mathbf{EG}x$.

Thus, C is a counterexample for $\mathbf{AF}\neg x$*. Note that the violation on C explains the violation on K.*

Algorithms for generating linear counterexamples and their complexity were first studied by Clarke *et al.* [6] and Hojati *et al.* [16], and are widely used in practice.

Unfortunately, the fragment of ACTL which has linear counterexamples is very weak, because only properties in ACTL $\cap$ LTL have linear counterexamples. This fragment has been syntactically characterized by Maidl [19]. Results by Buccafurri *et al.* [3] however show that recognizing ACTL formulas with linear counterexamples is PSPACE-hard. Consequently, an ACTL model checker must solve an infeasible problem in order to detect those cases where the construction of a linear counterexample is possible.

Example 1.2 *A counterexample for* $\mathbf{AFAX}p$ *has to show that there exists an infinite path π such that from every state of π a state with property $\neg p$ is reachable in one step. Therefore,* $\mathbf{AFAX}p$ *does in general not have linear counterexamples.*

We conclude that linear counterexamples are not complete for ACTL (i.e., not every violation of an ACTL specification is witnessed by a linear counterexample), and that recognizing the cases where linear counterexamples are applicable at all is algorithmically very hard.

In this paper, we investigate the question if there is a stable and useful notion of counterexamples which does not bear the insufficiencies of classical linear counterexamples described above. Thus, we are looking for a class $\mathcal{C}$ of Kripke structures subject to the following conditions:

- **Completeness.** The class $\mathcal{C}$ of counterexamples should be complete for a large class of temporal languages containing ACTL, i.e., each violation of an ACTL specification is witnessed by a suitable counterexample in $\mathcal{C}$.
- **Intelligibility.** The elements of $\mathcal{C}$ should be simple and specific enough to be analysed by human engineers, possibly with the aid of automated tools and suitable annotations. Note that in principle, the original Kripke structure is always a counterexample, albeit of no worth.
- **Effectiveness.** There should be effective (possibly symbolic) algorithms for generating and manipulating counterexamples in the class $\mathcal{C}$.

We say that $\mathcal{C}$ is a *viable* class, if $\mathcal{C}$ fulfills these criteria. The main contribution of this paper is the introduction of *tree-like counterexamples* as a candidate for a viable class. As indicated by the example in Figure 2, the SCCs (strongly connected components) of the transition relation in a tree-like counterexample form a finite tree, and the individual SCCs are directed cycles. We argue that tree-like counterexamples fulfill the required conditions. In particular, we show the following results about tree-like counterexamples:

- We prove that ACTL has tree-like counterexamples. To this end, we analyse the notion of counterexamples in model checking, and provide a formal framework for counterexamples.
- We define a family of *universal branching logics* $A\Omega$ which are extensions of ACTL by sets Ω of $\omega-$regular temporal operators and infinitary conjunction. We show that each logic in this family has tree-like counterexamples.

  As a consequence, the tree-like counterexample property follows for the universal fragments of many temporal logics including the *Buy One Get One Free Logic* by Grumberg and Kupferman [17] and other temporal logics with $\omega-$regular operators as defined and analysed in [15, 22, 10, 12, 24].

Further results are shown to substantiate our claim that $A\Omega$ is a family of useful and well-behaved specification logics.

- In Section 7, we devise symbolic algorithms to compute tree-like counterexamples for ACTL. These algorithms can be used to extend the model checker SMV [20] to compute tree-like counterexamples.
- Our interest in this topic was originally motivated by a new counterexample-guided refinement methodology for model checking [9]. In this methodology, increasingly accurate approximations of Kripke structures are obtained by analysing counterexamples. The method however suffered from the incompleteness of the counterexample feature in SMV. In Section 8, we extend the refinement methodology to tree-like counterexamples, thus achieving completeness of the counterexample-guided refinement methodology.

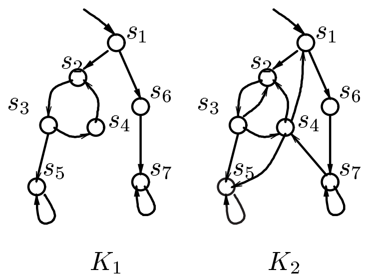


Figure 2. K_1 **is tree-like,** K_2 **is not.**

2. Technical Preliminaries

We assume that the reader is familiar with the basics of model checking and temporal logic, in particular CTL, LTL and Büchi automata. In this section, we recall some specific concepts, and fix the notation. For more background, consult [7, 8, 14, 23].

ACTL is the universal fragment of CTL where only the path quantifier **A** is allowed, and negation is restricted to atomic subformulas. ECTL is the dual existential fragment of CTL. Note that the name "ECTL" has been frequently used in the literature for other logics.

Throughout the paper, we fix a finite set of atomic propositions A. A Kripke structure is a tuple $K = (S, R, L, I)$ where S is the set of states, $R \subseteq S \times S$ is the set of transitions, $I \subseteq S$ is the non-empty set of initial states, and $L : S \to 2^A$ labels each state by a set of atomic propositions. We assume that the transition relation R is *total*, i.e., that all states have positive outdegree, and that there is a unique initial state, i.e., $I = \{s_{\text{init}}\}$. Thus, a Kripke structure can be viewed as a rooted directed graph (S, R) whose vertices are labeled by L.

A *path* is an *infinite* sequence of states, $\pi = s_0, s_1, \ldots$ such that for $i \geq 0$, $(s_i, s_{i+1}) \in R$. Given a path π, π^i denotes the infinite path $s_i, s_{i+1}, \ldots$, and $\pi(i)$ denotes the state s_i. A *finite path* is a finite prefix of a path. Since R is total, each finite path can be extended into an infinite path. Note that paths over a Kripke structure can be considered words over alphabet S. Given a word $s = s_0, s_1, \ldots$ and an integer i, $s(i)$ denotes the symbol s_i. If s is infinite, the size $|s|$ is ω, otherwise $|s|$ is the number of symbols. Given a set S, $\mathcal{P}(S)$ denotes the powerset of S.

3. What are Counterexamples?

Proof by counterexample is an ancient mathematical concept. A property φ claimed to hold for each element of a given set S can be disproved by choosing a single element $s \in S$ such that φ does not hold true for s. In contrast, existential properties can in general not be disproved by counterexamples.

Thus, for temporal logics, we can expect counterexamples only for universal fragments. Therefore, we will initially concentrate on counterexamples for specifications in the well-studied logic ACTL; at the end of this section, we will give a general definition of universality.

Suppose that an ACTL property φ is violated over a Kripke structure K, i.e., $K \not\models \varphi$. What do we expect of a counterexample C ?

(i) The counterexample C *violates* φ.

(ii) The violation of φ on C *"explains"* the violation on K.

(iii) The counterexample C is *viable*.

It remains to distill these informal postulates into a formal definition. Postulate (i) can be formally stated in the obvious way:

(I) $C \not\models \varphi$, or, equivalently, $C \models \neg\varphi$.

Note that $\neg\varphi$ is a formula in ECTL, the existential fragment of CTL, and that C is a **witness** of $\neg\varphi$.

Postulate (ii) requires that the violation of φ in the counterexample C explains the violation in the Kripke structure K. This link is provided by the simulation relation $\succeq$ in a natural way, because the simulation relation preserves ACTL formulas.

Theorem 3.1 (Simulation Theorem) *Let ψ be an ACTL formula. If $K \succeq C$ and $K \models \psi$ then $C \models \psi$.*

By contraposition it follows that if a counterexample C violates the ACTL formula $\varphi = \neg\psi$, then K violates φ as well. Hence, we fix the following definition:

(II) $K \succeq C$.

It is easy to see that the intuitively generated linear counterexamples in previous literature and implementations indeed are simulated by the Kripke structure.

Postulate (iii) leads us back to the conditions for counterexamples discussed in Section 1, i.e., *Intelligibility, Effectiveness, and Completeness*, and the proposed notion of tree-like counterexamples informally introduced above.

(III) The counterexample C is tree-like.

In the rest of this paper, we will demonstrate that finite tree-like Kripke structures are a natural and elegant choice for the viability criterion. In this manner, a very powerful, practically useful and versatile definition of counterexamples is obtained.

What is a universal logic ? We are now in a position to return to the question what constitutes a universal logic. The crucial property of ACTL needed above was the simulation theorem which said that for a universal property ψ,

$$K \succeq C \quad \text{and} \quad K \models \psi \quad \text{implies} \quad C \models \psi.$$

Consequently, we say that a logic $\mathcal{L}$ is *universal* if the simulation theorem holds for $\mathcal{L}$.

4. Tree-Like Counterexamples

Let G be a directed graph. The component graph $c(G)$ of G is the graph where the vertices are given by the strongly connected components (SCCs) of G, and where two vertices of $c(G)$ are connected by an edge if there is exists an edge between vertices in the corresponding SCCs. A graph is tree-like, if (i) all SCCs are cycles, and (ii) the component graph is a directed tree. A Kripke structure $K = (S, R, L, \{s_{\text{init}}\})$ is tree-like if the graph (S, R) is a *finite* tree-like graph whose root is the initial state s_{init} of K. Figure 2 shows an example of a tree-like Kripke structure.

Intuitively, a Kripke structure is tree-like if it is obtained by glueing finite cycles to leaves of a tree, glueing finite trees to vertices in the cycles, etc.

Our first main result states the completeness of tree-like counterexamples for ACTL.

Theorem 4.1 *ACTL has tree-like counterexamples.*

The proof of this result will follow from a more general result described in Sections 5 and 6. An effective symbolic algorithm to construct ACTL counterexamples will be presented in Section 7.

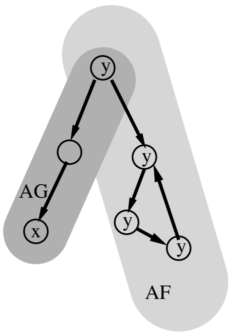


Figure 3. Counterexample for $\mathbf{AG}\,\neg x \vee \mathbf{AF}\,\neg y$

Example 4.1 *A counterexample for the ACTL specification $\mathbf{AG}\,\neg x \vee \mathbf{AF}\,\neg y$ has to demonstrate that there is a finite path leading to a state satisfying x,* **and** *that there is an infinite path along which y is always true. Thus, a counterexample is a model of the ECTL formula $\mathbf{EF}\,x \wedge \mathbf{EG}\,y$. A tree-like counterexample typically has the form described in Figure 3. The shaded areas indicate which subformula is disproved.*

Example 4.2 *For the specification $\mathbf{AF}(\neg y \wedge \mathbf{AX}\,\neg x)$ a counterexample is a model of $\mathbf{EG}(y \vee \mathbf{EX}\,x)$. A typical tree-like counterexample is shown in Figure 4.*

The examples demonstrate that counterexamples for ACTL are closely related to finite models for ECTL. CTL

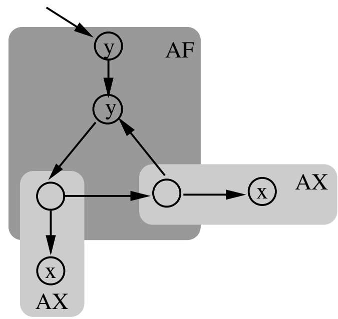


Figure 4. Counterexample for $\mathbf{AF}(\neg y \wedge \mathbf{AX}\,\neg x)$

is well-known to have both the *tree model property* and the *finite model property*: if a CTL formula has a model at all, then it has (i) a model whose underlying graph is a (typically infinite) tree, and (ii) a finite model.

The *tree-like model property* introduced here is stronger than the mentioned properties. It requires that the formula has a finite model which is tree-like. Obviously, the tree-like model property implies the other two properties.

Corollary 4.1 *ECTL has the tree-like model property.*

Remark 4.1 (Counterexample Skeletons) *The tree-like model property provides us with a way to generate* skeletons *for possible counterexamples: Let φ be an ACTL formula, and $\neg\varphi$ its ECTL negation. Then the tree-like models of $\neg\varphi$ describe all possible counterexamples of the formula φ over different Kripke structures.*

The duality between generation of counterexamples for $ACTL$ and generating models for $ECTL$ will be exploited in Section 7.

5. From ACTL to $A\Omega$

The major shortcoming of ACTL and ACTL* is the weakness of the path formulas which can employ only (un)nested LTL connectives. Many interesting properties, in particular cyclic events cannot be expressed in this framework. Various authors have therefore investigated extensions of ACTL* by ω-regular linear time operators [15, 22, 10, 12, 24].

The logic we shall define is closely related to the mentioned logics. Essentially, we also extend ACTL by ω-regular linear time operators. However, our logic retains the *monotonicity of the linear time operators*, a subtle but important detail of ACTL and ACTL*. To understand this feature, consider the operator $\mathbf{F}p$. Suppose that $\pi \models \mathbf{F}p$, and consider a path σ which is obtained from π by adding the label p to zero or more states. (In symbols, $\pi \subseteq_p \sigma$.) Then it is easy to see that $\sigma \models \mathbf{F}p$, whence $\mathbf{F}$ is monotonic.

The monotonicity of the standard LTL operators makes it possible to define universal logics such as ACTL and ACTL*, since otherwise one could use non-monotonic operators to implicitly introduce negation. Thus, *monotonicity of the operators is a key to the counterexample property*. The definition of extended temporal operators put forward in this paper improves on previous proposals because it also allows for natural universal fragments of extended branching time logics.

We view LTL operators as *patterns* on the time line, i.e., patterns which can be observed on paths. For example, the LTL formula $\mathbf{F}\varphi$ ("φ will become true in the future") describes the path patterns

$$M_1, \quad \bot M_1, \quad \bot\bot M_1, \quad \bot\bot\bot M_1, \quad \ldots$$

where M_1 denotes the position where φ actually occurs, and $\bot$ denotes unconstrained positions ("*don't care*"). This set can be described by the regular expression $\bot^\star M_1$.

Similarly, the LTL formula $\varphi\mathbf{U}\psi$ corresponds to the regular pattern $M_1^\star M_2$ where M_1 and M_2 describe the positions where φ and ψ must hold respectively. M_1, M_2 are called *markers*. Thus, we obtain the following $\omega-$regular patterns for the standard LTL operators:

X	$\bot M_1$	**F**	$(\bot)^\star M_1$
G	$(M_1)^\omega$	**U**	$(M_1)^\star M_2$

Formally, we define a *temporal operator* O with n input formulas by a (possibly infinite) set of (possibly infinite) words over the alphabet $\Sigma_n = \mathcal{P}(\{M_1, \ldots, M_n\})$. Thus, each symbol in the alphabet expresses zero or more marks. For simplicity, we write $\bot$ for $\emptyset$, and M_i for the singleton $\{M_i\}$. If O is defined by an $\omega-$regular expression, we say that O is a *Büchi operator*. Similarly, we can say that O is regular, computable, etc.

Given a temporal operator O, the corresponding temporal connective is denoted by $\mathbf{O}$. Satisfaction of $\mathbf{O}$ is defined in the natural way: Let $\pi = p_0, p_1, \ldots$ be a path in a Kripke structure K, and $\varphi_1, \ldots, \varphi_n$ be a sequence of formulas. Then

$K, \pi \models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$ if there exists a pattern $o \in O$ such that for all positions $i < |o|$, and for all $M_k \in o(i)$ it holds that $K, \pi^i \models \varphi_k$.

Remark 5.1 (Comments on the Definition)

(i) *Note that in contrast to previous literature, we only require that the presence of a marker symbol M_k* **implies** *satisfaction of the corresponding formula φ_k, and not vice versa. Thus, we use the patterns as* **constraints***, and not as a one-to-one description of the whole path with respect to the subformulas $\varphi_1, \ldots, \varphi_n$. In particular, we do not have markers which enforce the negation of a subformula to hold. Thus, all operators we define are* **monotonic***. As argued above, this monotonicity property is crucial to our approach.*

(ii) *It is not important that we choose to define the patterns in terms of $\omega-$regular expressions. We can use Büchi automata or other equivalent formalisms such as S1S or quantified temporal logic [21].*

Example 5.1 *We describe a new connective $\mathbf{K}\varphi$ which requires that φ holds at all even time points, and another connective $\mathbf{L}\varphi$ which requires there should be no more than four time units between two occurrences of φ.*

K	$(\bot M_1)^\omega$
L	$(M_1 \vert \bot M_1 \vert \bot\bot M_1 \vert \bot\bot\bot M_1 \vert \bot\bot\bot\bot M_1)^\omega$

In order to make the notion of monotonicity precise, we need to compare paths. For a path π and a formula φ, φ^π denotes the set of states in π where φ holds. For a sequence of formulas $\varphi_1, \ldots, \varphi_n$, we define $\pi \subseteq_{\varphi_1,\ldots,\varphi_n} \sigma$ iff $\bigwedge_{i=1}^n \varphi_i^\pi \subseteq \varphi_i^\sigma$.

Lemma 5.1 (Monotonicity) *If $K, \pi \models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$ and $\pi \subseteq_{\varphi_1,\ldots,\varphi_n} \sigma$ then $K, \sigma \models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$.*

As we have seen above, monotonicity of operators is necessary to obtain logics with a clear notion of *universal statements*. The following corollary shows that monotonicity is also closely connected to the existence of counterexamples.

Corollary 5.1 *If $K, \pi \not\models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$ and $\sigma \subseteq_{\varphi_1,\ldots,\varphi_n} \pi$, then $K, \sigma \not\models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$. In other words, the refutation of $\mathbf{O}(\varphi_1, \ldots, \varphi_n)$ on π does* **not** *depend on satisfied subformulas, but* **only** *on violated subformulas.*

Example 5.2 *(a) Suppose that $K, \pi \not\models \mathbf{F}\varphi$. $K, \pi \models \mathbf{F}\varphi$ can be disproved by showing that for all j, $K, \pi^j \not\models \varphi$.*

(b) If $K, \pi \not\models \mathbf{G}\varphi$, then $K, \pi \models \mathbf{G}\varphi$ can be disproved by showing that $K, \pi^j \not\models \varphi$ for suitable j.

Note however that there may be many positions j with $K, \pi^j \not\models \varphi$. According to the above argument, all of these positions are used to disprove $K, \pi \models \mathbf{F}\varphi$, although in this concrete case only one such position is needed.

(c) If $K, \pi \not\models \mathbf{K}\varphi$, then $K, \pi \models \mathbf{K}\varphi$ can be disproved by finding an even position j such that $K, \pi^j \not\models \varphi$.

The corollary will be applied to *counterexample generation* as follows:

(a) Suppose that $K, \pi \not\models \mathbf{O}(\varphi_1, \ldots, \varphi_n)$.

(b) Then Corollary 5.1 says that identifying all **violations** of the formulas $\varphi_1, \ldots, \varphi_n$ on the path π suffices to disprove $\mathbf{O}(\varphi_1, \ldots, \varphi_n)$.

(c) Consequently, a counterexample for $\mathbf{O}(\varphi_1, \ldots, \varphi_k)$ is reduced to a collection of counterexamples for the single φ_i's.

Finally we are ready to define the logics $A\Omega$.

Definition 5.1 *Let Ω denote a (possibly infinite) set of temporal operators. Then $A\Omega$ is the universal branching time time logic given by the following set of formulas:*

(i) Each atomic formula is in $A\Omega$.

(ii) Each negated atomic formula is in $A\Omega$.

(iii) If $O \in \Omega$ is an operator of arity k, and $\varphi_1, \ldots, \varphi_k$ are in $A\Omega$, then $\mathbf{AO}(\varphi_1, \ldots, \varphi_k)$ is in $A\Omega$.

(iv) If φ_1, φ_2 are in $A\Omega$, then $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are in $A\Omega$.

(v) If $\varphi_1, \varphi_2, \ldots$ are in $A\Omega$, then the infinitary formula $\bigwedge_{i \geq 1} \varphi_i$ is in $A\Omega$.

Let K be a Kripke structure, and s a state. The semantics of $A\Omega$ is defined as follows:

(i) If φ is atomic then $K, s \models \varphi$ iff $\varphi \in L(s)$.

(ii) $K, s \models \neg\varphi$ iff $K, s \not\models \varphi$.

(iii) $K, s \models \mathbf{AO}(\varphi_1, \ldots, \varphi_k)$ iff for all paths $\pi = s, \ldots$ starting at s it holds that $K, \pi \models \mathbf{O}(\varphi_1, \ldots, \varphi_k)$.

(iv) $K, s \models \varphi_1 \vee \varphi_2$ iff $K, s \models \varphi_1$ or $K, s \models \varphi_2$, and similarly for $\wedge$.

(v) $K, s \models \bigwedge_{i \geq 1} \varphi_i$ iff for all $i \geq 1$ it holds that $K, s \models \varphi_i$.

The logic $E\Omega$ is defined likewise, with $\mathbf{E}$ instead of $\mathbf{A}$, and $\bigvee_{i \geq 1}$ instead of $\bigwedge_{i \geq 1}$.

We observe that ACTL and ACTL* can be defined as subsets of $A\Omega$ with finitary conjunction.

Lemma 5.2 *Let Ω be a set of temporal operators. Then the logic $A\Omega$ is universal.*

Finally, we can state the main counterexample theorem:

Theorem 5.1 (Counterexample Theorem) *Let Ω be a set of temporal Büchi operators. Then $A\Omega$ has tree-like counterexamples. Furthermore, the tree-like counterexamples are effectively computable.*

Corollary 5.2 *$E\Omega$ has the tree-like model property.*

The following subsection is devoted to the proof and algorithm for this theorem. Along the way, we will introduce several notions which are also important later on.

6. Constructing Counterexamples

Let us start with a property of Büchi operators which is crucial to our approach.

Lemma 6.1 *Let $\mathbf{O}$ be a Büchi operator, K a Kripke structure, and s_0 a state such that $K, s_0 \not\models \mathbf{AO}(\psi_1, \ldots, \psi_k)$. Then there exists a path $\sigma = s_0, \ldots$ such that*

(i) $K, \sigma \not\models \mathbf{O}(\psi_1, \ldots, \psi_k)$.

(ii) σ has the form
$$s_0, \ldots, s_N, \quad s_N, \ldots, s_{N+M}, \quad s_N, \ldots, s_{N+M}, \ldots.$$

Proof Sketch. Recall that O is a set of patterns over alphabet $\Sigma_k = \mathcal{P}(\{M_1, \ldots, M_k\})$. The proof idea is to construct a Büchi automaton for the patterns of $\neg\mathbf{O}$, and use an accepting run of the automaton to obtain a path σ with the required property.

In order to obtain the patterns for $\neg\mathbf{O}$, we cannot use the set-theoretic complement of O. For example, the complement of the pattern $(\bot)^\star M_1$for $\mathbf{F}$ contains the pattern $M_1 M_1$, although a path where the constraint $M_1 M_1$ holds will satisfy $\mathbf{F}$.

We conclude that we need the complement of the set O' where O' is the "monotonic hull" of O in the sense of Lemma 5.1.

O' is constructed as follows: For each $\sigma \in \Sigma_k$, let $\sigma_1, \ldots, \sigma_{2^{k-|\sigma|}}$ be the set of all symbols of the alphabet which are supersets of σ, and denote the regular expression

$$(\sigma_1 | \cdots | \sigma_{2^{k-|\sigma|}})$$

by σ'. Let $\mathcal{R}$ be the ω-regular expression describing O. We define a new regular expression $\mathcal{R}'$ by replacing all occurrences of alphabet elements σ by the expression σ'. $\mathcal{R}'$ describes the following language O': If a pattern o is in O, then all patterns obtained from o by adding zero or more additional markers are in O'.

The complement of O' is the set of all patterns which violate the operator. Note however, that in the context of $A\Omega$, the operators $\mathbf{O}$ and $\mathbf{O}'$ are identical, due to the inherent monotonicity in the definition of $A\Omega$, cf. Remark 5.1 (i).

Let $\mathcal{A}$ be the Büchi automaton recognizing the complement of O', and let $\pi = s_0, \ldots$ be a path such that $K, \pi \not\models \mathbf{O}(\psi_1, \ldots, \psi_k)$.

Consider the infinite string s_π over alphabet Σ_k where $s_\pi(i) = \{M_j : K, \pi^i \not\models \psi_j\}$ for all $i \geq 0$. By construction $\mathcal{A}$ accepts s_π. Consider one accepting run of $\mathcal{A}$, and let q be an accepting state of $\mathcal{A}$ which appears infinitely often in this run. (There must exist such a state because this is the acceptance condition for a Büchi automaton, and we assumed an accepting run of $\mathcal{A}$.)

Let $a_1 < a_2 < \dots$ be the indices of those positions where $\mathcal{A}$ is in state q, and consider the sequence

$$\pi(a_1), \pi(a_2), \pi(a_3), \dots$$

of corresponding Kripke structure states in the path π. Since K has only a finite number of states, there must exist two indices $J < J'$ such that $\pi(a_J) = \pi(a_{J'})$. Choose the smallest such J and J', and consider the infinite path σ given by

$\pi(0), \dots, \pi(a_J), \quad \pi(a_J), \dots, \pi(a_{J'} - 1), \quad \pi(a_J), \dots, \pi(a_{J'} - 1), \dots$.

Construct an infinite word s_σ in the same way as s_π. By construction, $\mathcal{A}$ accepts s_σ, and therefore $K, \sigma \not\models \mathbf{O}(\psi_1, \dots, \psi_k)$. Setting $N := a_J$ and $M = (a_{J'} - 1) - N$ concludes the proof. $\Box$

It is not hard to come up with an algorithm which realizes the result of Lemma 6.1. The algorithm gradually unwinds the Kripke structure starting at s_0 like a breadth search algorithm, and evaluates the Büchi automaton $\mathcal{A}$ on all paths simultaneously until it finds a repetition as indicated in the proof.

Consider the path σ obtained from Lemma 6.1. If the states $s_0, \dots, s_{N+M}$ are all mutually different, then the path σ is easily seen to describe a simple tree-like substructure of K containing a path $s_0, \dots, s_N$ leading to a loop with states $s_N, \dots, s_{N+M}$. In general, however, this won't be the case, because states may appear repeatedly in the sequence $s_0, \dots, s_{N+M}$. Therefore, for each state s of K we will introduce isomorphic copies $s^1, s^2, \dots$, and make sure that repeated states are avoided. We call this process *unraveling* the path. Note that copies of states have also been used for traditional linear counterexamples, when the counterexample trace leads through the same state several times.

Formally, we introduce an *indexed Kripke structure* K^ω obtained from K by creating a countable number of isomorphic copies of each state. These copies are distinguished by an index, but cannot be distinguished by temporal properties.

Definition 6.1 *Given a Kripke structure $K = (S, I, R, L)$, the* indexed *Kripke structure $K^\omega = (S^\omega, I^\omega, R^\omega, L^\omega)$ is defined as follows.*

(i) $S^\omega = S \times \mathbb{N}$, i.e., the states have the form $\langle s, i\rangle$. By convention, we write s^i instead of $\langle s, i\rangle$. i is called the index of the state s^i.

(ii) $I^\omega = I \times \mathbb{N}$.

(iii) For any two states $s_1^i, s_2^j \in S^\omega$, $(s_1^i, s_2^j) \in R^\omega$ if and only if $(s_1, s_2) \in R$;

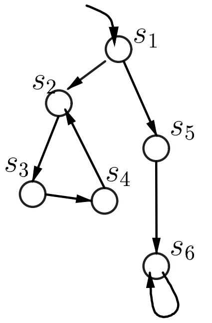
(iv) For all states $s^i \in S^\omega$ we have $L^\omega(s^i) = L(s)$.

Let $\pi = p_0, p_1, \dots, p_k$ be a finite path on K, and C an integer number. The unraveled path *at C is the finite path* $\text{unravel}(C, \pi) = p_0^C, p_1^{C+1}, \dots, p_N^{C+N}$ *over K^ω.*

Lemma 6.2 *The Kripke structures K and K^ω are bisimilar.*

Algorithm CEX. We will now describe the algorithm **CEX** depicted in Figure 6. Given K, s and φ, such that $K, s \not\models \varphi$, a call $\mathbf{CEX}(K, s^0, \varphi)$ computes a tree-like counterexample for $K, s \models \varphi$. Formally, the tree-like counterexample is constructed as a substructure of K^ω. Internally, the algorithm does not construct K^ω, but only keeps a global variable C to denote the index of states. Initially, C is set to 0.

CEX: Output Format. To describe the tree-like counterexample, the algorithm uses the following simple formalism: It outputs a sequence of *descriptors* of the form $\langle s_0, \dots, s_n\rangle$ (*path descriptor*) and $\langle s_0, \dots, s_n, s_0\rangle$ (*loop descriptor*) where the s_i are mutually different states in K^ω. For a descriptor $\text{desc} = \langle s_0, \dots\rangle$, $\text{desc}(i)$ denotes the state s_i. A set of descriptors Π is called a *description*. A description Π describes a Kripke structure $\bigcup \Pi$ whose states are the states occurring in the Π, and whose transitions are the transitions occurring in Π. We will usually write $\langle s_0, \dots, s_n\rangle^\omega$ to denote the loop descriptor $\langle s_0, \dots, s_n, s_0\rangle$. See Figure 5 for an example of this formalism.



$\langle s_1, s_2\rangle$
$\langle s_1, s_5, s_6\rangle$
$\langle s_6\rangle^\omega$
$\langle s_2, s_3, s_4\rangle^\omega$

Figure 5. Description of a tree-like Kripke structure

In particular, we see that the path generated by Lemma 6.1 can always be unraveled to obtain a description of the form $\langle p_0, \dots, p_N\rangle, \langle p_N, \dots, p_{N+M}\rangle^\omega$.

CEX: Overview. Starting with the original specification, and recursively constructing counterexamples for subformulas of the specification, the algorithm **CEX** traverses the specification formula in a depth-first manner. For

Algorithm $\mathbf{CEX}(K, s_0^i, \varphi)$
 case φ **of**
  $\varphi_1 \vee \varphi_2$:
   $\mathbf{CEX}(K, s_0^i, \varphi_1)$
   $\mathbf{CEX}(K, s_0^i, \varphi_2)$

  $\bigwedge_{i \geq 1} \varphi_i$:
  $\varphi_1 \wedge \varphi_2$:
   Select j **such that** $K, s_0 \not\models \varphi_j$.
   $\mathbf{CEX}(K, s_0^i, \varphi_j)$

  $A\mathbf{O}(\psi_1, \ldots, \psi_k)$:
   Determine $s_0, \ldots, s_N, \ldots, s_{N+M}$
    according to Lemma 6.1
   $\mathrm{desc}_1 := \langle s_0^i, \mathrm{unravel}(C, s_1, \ldots, s_N) \rangle$
   $\mathrm{desc}_2 := \langle \mathrm{unravel}(C+N, s_N, \ldots, s_{N+M}) \rangle^\omega$
   output desc_1 **and** desc_2
   $C := C + N + M + 1$
   for all states p **in** $\bigcup\{\mathrm{desc}_1, \mathrm{desc}_2\}$
    for $j \in \{1, \ldots, k\}$
     if $K, p \not\models \psi_j$ **then** $\mathbf{CEX}(K, p, \psi_j)$
 esac

Figure 6. CEX: The generic counterexample algorithm for $A\Omega$.

each subformula φ, the algorithm computes the description of a counterexample, and ensures that all the small counterexamples are glued together in the correct manner by assigning the right indices to newly generated sub-counterexamples.

The algorithm assumes that $K, s \not\models \neg\varphi$, and it assumes a model checking procedure for $A\Omega$ as a subprogram. For the case of propositional connectives, the construction of the counterexample is straightforward, and demonstrated in Figure 6: a conjunction is disproved by a path violating one of the conjuncts, while a disjunction is disproved by a pair of paths starting at the same state.

Let us turn to the important case of the operator $\mathbf{AO}(\psi_1, \ldots, \psi_k)$: In this case, the algorithm determines a path σ violating $\mathbf{AO}(\psi_1, \ldots, \psi_k)$ as in Lemma 6.1, and unravels σ to obtain two descriptors desc_1 and desc_2. Therefore, $\bigcup\{\mathrm{desc}_1, \mathrm{desc}_2\}$ describes a finite path which leads to a cycle. Recall that C is a global constant. Unraveling σ at C makes sure that no states are repeated. Note that the state s_0^i is not changed in the unraveling, because it is an input parameter which connects desc_1 to other parts of the global counterexample.

Afterwards, counterexamples for the subformulas $\psi_1, \ldots, \psi_k$ are generated. To this end, all states p in the newly generated counterexample $\bigcup\{\mathrm{desc}_1, \mathrm{desc}_2\}$ are verified with respect to each subformula ψ_i, and in case of failure, $\mathbf{CEX}(K, p, \psi_i)$ is evoked recursively.

Remark 6.1 **CEX** *can be easily extended to output additional annotations or comments together with the descriptors. These annotations can be used to improve the readability of the counterexample as in Examples 4.1 and 4.2.*

CEX: Correctness. The correctness of the algorithm relies on three basic insights:

(i) The *monotonicity* property (cf. Corollary 5.1 and the comments thereafter) enables us to reduce counterexample computation for a formula φ to counterexample computation for the subformulas of φ. This guarantees that during the recursive traversal of the specification, only counterexamples for formulas in $A\Omega$ (and *not for formulas in* $E\Omega$) have to be constructed.

(ii) The *indexed* Kripke structure K^ω allows us to unravel intersecting, but logically independent parts of the counterexample. This can be seen as a partial (or lazy) unwinding of the Kripke structure, and is the key to the tree-like model property in Corollaries 4.1 and 5.2.

(iii) The ω-regularity of the Büchi operators allows us to build the global counterexample from small counterexamples of the form $\langle s_0, \ldots, x \rangle, \langle x, \ldots \rangle^\omega$, cf. Lemma 6.1.

Lemma 6.3 *Let $\varphi \in A\Omega$, where Ω is a set of Büchi operators, and assume that $K, s \not\models \varphi$. Let Π be the set of descriptors computed by $\mathbf{CEX}(K, s^0, \varphi)$. Then $\bigcup \Pi$ is a tree-like counterexample.*

This concludes the proof of the Counterexample Theorem.

Remark 6.2 (Optimization of Counterexamples) *There are several evident points where* **CEX** *can be optimized to obtain smaller and simpler counterexamples. (i) $\bigwedge_{i \geq 1} \varphi_i$: There may be several formulas φ_i which violate the conjunction. In this case, the algorithm can choose one of them which leads to a possibly small sub-counterexample. (ii) $A\mathbf{O}(\psi_1, \ldots, \psi_k)$: As argued above in Example 5.2, the recursion on the subformulas ψ_i can often be restricted to a small number of those states where the ψ_i are violated. In particular, we can exit the* **for** *loop over the states in $\bigcup\{\mathrm{desc}_1, \mathrm{desc}_2\}$, if at a given moment the generated sub-counterexamples already suffice to disprove $A\mathbf{O}(\psi_1, \ldots, \psi_k)$. However, a counterexample generated in this way may be harder to read. (iii) $A\mathbf{O}(\psi_1, \ldots, \psi_k)$: Depending on* $\mathbf{O}$*, it is not always necessary to generate two descriptors. For example, in the case of $\mathbf{O} = \mathbf{G}$, the counterexample contains only one path descriptor.*

We omit the details due to space restrictions. Note however that it has been shown in [13] that in general, minimizing the size of counterexamples is NP-hard.

7. Application I: Symbolic Counterexample Generation

Symbolic model checking [4, 20, 7] was a breakthrough which tremendously increased industrial acceptance of formal methods. In symbolic model checking, states of the Kripke structure are not handled explicitly. Instead, sets of states are described by Boolean functions (using Binary Decision Diagrams [2] as a data structure), and manipulated by the algorithm using set operations. Similarly, the transition relation R is encoded by a Boolean function. An algorithm in this framework is called a *symbolic algorithm*. Symbolic algorithms typically do not iterate through states. Instead, they often perform fixpoint computations of set operators. This is motivated by the fact that equality of sets (such as the fixpoint equality $\tau(X) = X$) can be efficiently computed for Binary Decision Diagrams. Symbolic algorithms are also allowed to use the operator $Img(T)$ which for a set T of states computes the set of successors of the states in T with respect to the transition relation R. The preimage $Img^{-1}(T)$ is defined analogously for predecessors.

The system SMV [20] was the first symbolic verification system, and has been the basis for many state-of-the-art model checkers. In this section, we will show how to extend SMV to compute tree-like counterexamples.

In the context of SMV, we will not construct counterexamples for ACTL directly, but rather construct witnesses for the dual logic ECTL. Recall that ECTL contains the operators **EX**, **EF**, **EG** and **EU**. The algorithm **print_witness** (Figure 7) for ECTL is modeled after the algorithm $\mathbf{CEX}$ for $A\Omega$. For propositional connectives and literals, no witness is necessary. For the four operators **EX**, **EF**, **EG** and **EU**, there are specialized subprocedures. $\mathrm{SAT}(\varphi)$ denotes the set of states where φ holds.

Let us first consider the case of **EF**; most other cases except **EG** are similar. For a formula $\mathbf{EF}\varphi$, the model checking procedure computes the least fixpoint of the following monotone operator τ:

$$\tau(X) = \mathrm{SAT}(\varphi) \cup Img^{-1}(X).$$

To obtain the least fixpoint of τ, one computes the sequence $S_1 = \tau(\emptyset), S_2 = \tau(\tau(\emptyset)), \ldots$ until it converges after a finite number of steps with a set S_n. It is easy to see that $S_n = \mathrm{SAT}(\mathbf{EF}\varphi)$, and that each S_i contains the set of states from where $\mathrm{SAT}(\varphi)$ is reachable in at most $i-1$ steps. (For details, consult [7].)

The sets $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_n$ are called the *stages* of the fixpoint computation. The idea is to use these stages to compute a witness. It is clear that the witness for **EF** will be given by a path descriptor. Consider the algorithm **print_witnessEF** of Figure 8. We know that $K, s_0 \models \mathbf{EF}\,\varphi$, and therefore $s_0 \in S_n$. The procedure looks at the successor states of s_0 to obtain an element of S_{n-1}.

Algorithm print_witness(φ, s^i)
- **if** $\varphi = \mathbf{EX}\,\psi$ **then**
  - return **print_witnessEX**(ψ, s^i)
- **if** $\varphi = \mathbf{EF}\,\psi$ **then**
  - return **print_witnessEF**(ψ, s^i)
- **if** $\varphi = \mathbf{EG}\,\psi$ **then**
  - return **print_witnessEG**(ψ, s^i)
- **if** $\varphi = \mathbf{EU}\psi$ **then**
  - return **print_witnessEU**(ψ, s^i)
- **if** $\varphi = \psi_1 \wedge \psi_2$ **then**
  - **print_witness**(ψ_1, s^i)
  - **print_witness**(ψ_2, s^i)
- **if** $\varphi = \psi_1 \vee \psi_2$ **then**
  - **if** $s^i \in \mathrm{SAT}(\psi_1)$
    - then **print_witness**(ψ_1, s^i)
    - else **print_witness**(ψ_2, s^i)

Figure 7. Algorithm print_witness

Note that all elements of S_{n-1} can reach $\mathrm{SAT}(\varphi)$ within at most $n-2$ steps. Iterating this procedure, we obtain a finite path $s_0, \ldots, s_j$, $j < n$, and unravel it in the same way as in $\mathbf{CEX}$. Finally, we recursively call the procedure **print_witness** to obtain a witness for φ starting at state s_j.

Algorithm print_witnessEF(φ, s_0^i)
- **Determine the stages** $(S_1, \ldots, S_n)$ **of computing EF** φ.
- $j := 0$
- **repeat**
  - $j := j + 1$
  - $S := Img(s_{j-1}) \cap S_{n-j}$
  - **choose** $s_j \in S$
- **until** $s_j \in S_1$
- desc := $\langle s_0^i, \mathrm{unravel}(C, s_1, \ldots, s_j)\rangle$
- **output** desc
- $C := C + j$
- $\mathbf{print\_witness}(\varphi, s_j^C)$

Figure 8. Algorithm print_witnessEF

The algorithm $\mathbf{print\_witnessEU}$ is based on least fixpoints, and therefore very similar to $\mathbf{print\_witnessEF}$. The algorithm $\mathbf{print\_witnessEX}$ is very simple, because the computation has only one stage. Due to space restrictions we omit both of them.

Let us finally turn to $\mathbf{EG}\varphi$. In contrast to the other operators, $\mathbf{EG}$ is computed by a greatest fixpoint, and the stages do not help us to generate a witness. Thus, the algorithm uses only the last stage S_n of the computation. By definition, $S_n = \mathrm{SAT}(\mathbf{EG}\,\varphi)$, and therefore, all states in S_n satisfy φ and $\mathbf{EG}\,\varphi$. The Kripke structure restricted to the states in S_n is total, i.e., it does not contain any dead ends.

We use a greedy algorithm to find a cycle in S_n as follows: Starting at s_0, we go along an arbitrary path within S_n and collect all states visited so far in a set T. As soon as we reach an element of T, we have found a loop, and output it. Then, we proceed recursively as described for $\mathbf{CEX}$. The algorithm **print_witnessEG** is shown in Figure 9.

Theorem 7.1 *The algorithm* **print_witness** *computes correct tree-like counterexamples for ACTL.*

Algorithm print_witnessEG(φ, s_0^i)
  $S_n := \mathrm{SAT}(\mathbf{EG}\,\varphi)$ (last stage of the greatest fixpoint)
  $T := \{s_0\}$
  $j := 0$
  repeat
    $j := j + 1$
    $S := Img(s_{j-1}) \cap S_n$
    choose $s_j \in S$
    $T := T \cup \{s_j\}$
    $Q := Img(s_j) \cap T$
  until $Q \neq \emptyset$
  choose N **where** $s_N \in Q$
  $\mathrm{desc}_1 := \langle s_0^i, \mathrm{unravel}(C, s_1, \ldots, s_N)\rangle$
  $\mathrm{desc}_2 := \langle \mathrm{unravel}(C + N, s_N, \ldots, s_j)\rangle^\omega$
  output desc_1 **and** desc_2
  $C := C + j + 1$
  for all states p **in** $\bigcup\{\mathrm{desc}_1, \mathrm{desc}_2\}$
    if $K, p \models \varphi$ **then print_witness**(φ, p)

Figure 9. Algorithm print_witnessEG

8. Application II: Counterexample-Guided Refinement

Despite the success of symbolic methods, the state explosion problem remains a major hurdle in verification, because the size of the OBBDs may become too large. Existential abstraction [11] is an important complementary method to attack the state explosion problem. In this framework, the set S of states of a Kripke structure K is mapped to a (typically smaller) set of *abstract* states $\widehat{S}$ to obtain an *abstract* Kripke structure $\widehat{K}$. h induces an equivalence relation on S in the natural way, where the equivalence classes (called *clusters*) are the preimages $h^{-1}(\widehat{s})$ of the abstract states. In $\widehat{K}$, there is a transition between two abstract states $\widehat{s}$ and $\widehat{t}$, if there exists a transition between any pair of concrete states in $h^{-1}(\widehat{s})$ and $h^{-1}(\widehat{t})$.

Since we lose information by switching from K to $\widehat{K}$, we need a preservation theorem which enables us to model check $\widehat{K}$ instead of K. Therefore, we will define $\widehat{K}$ in such a way that $\widehat{K}$ simulates K. Simulation is guaranteed by using an abstraction function h where the clusters of h cannot be distinguished by the *atomic properties* occurring in the specification φ. The simulation theorem then guarantees that $\widehat{K} \models \varphi$ implies $K \models \varphi$ for ACTL formulas φ.

Algorithm CheckRefine$(\widehat{s}_0, \widehat{Q})$
  $T = h^{-1}(\widehat{s}_0)$ *(The result will be a subset of T.)*
  for each $\widehat{q} \in \widehat{Q}$
    if $\widehat{q}(0) = \widehat{s}_0$ **then** *(If $\widehat{s}_0$ is the beginning of $\widehat{q}$)*
      $l := |\widehat{q}|$ *(number of different states in $\widehat{q}$)*
      $S_1 := h^{-1}(\widehat{s}_0)$
      for $(i := 2$ **to** $l)$
        $S_i :=$ **CheckRefine**$(\widehat{q}(i-1), \widehat{Q})$
        (Depth-First Recursion)
      if $\widehat{q}$ **is a path descriptor then**
        $T := T \cap \mathbf{CheckPATH}(\widehat{q}, S_1, S_2, \ldots, S_l)$
      if $\widehat{q}$ **is a loop descriptor then**
        $T := T \cap \mathbf{CheckLOOP}(\widehat{q}, S_1, S_2, \ldots, S_l)$
  return T

Figure 10. Refinement algorithm for all ACTL formulas

The implication however, holds only in one direction. In other words, if $\widehat{K} \not\models \varphi$, the only information we have is a counterexample on the abstract model $\widehat{K}$. It is not clear however, if this *abstract counterexample* induces a real counterexample, or if it is a result of the information loss brought upon by the abstraction. Such counterexamples are called *spurious*.

In [5] an automatic method called *counterexample-guided abstraction refinement* was presented. Starting with an initial abstraction h obtained from the SMV program text, the model checker verifies the specification φ on $\widehat{K}$. If $\widehat{K} \models \varphi$, the algorithm terminates. Otherwise, an abstract counterexample $\widehat{T}$ is generated, and the algorithm checks whether $\widehat{T}$ is spurious. If $\widehat{T}$ is not spurious, the algorithm knows that $K \not\models \varphi$ and terminates. Otherwise, the reason for $\widehat{T}$ being a spurious counterexample is identified, and the whole process is iterated with a refined abstraction function h'. Counterexample analysis has been introduced previously in [18, 1].

The counterexample-guided abstraction refinement was implemented in the system aSMV based on SMV. Originally, aSMV was only complete for the fragment of ACTL with linear counterexamples. In the rest of this section, we describe how to extend the method to full ACTL.

aSMV uses two symbolic procedures, **CheckPath** and **CheckLoop**. Let $\langle \widehat{p_1}, \ldots, \widehat{p_n}\rangle$ be a path descriptor, and $S_1, \ldots, S_n$ be a sequence of sets of states such that $S_i \subseteq h^{-1}(\widehat{p_i})$. Then $\mathbf{CheckPath}(\langle \widehat{p_1}, \ldots, \widehat{p_n}\rangle, S_1, \ldots, S_n)$ returns the set of all states s in S_1 from which a state in S_n can be reached along a path π of size n which leads successively through $S_1, \ldots, S_n$. *If the result of* $\mathbf{CheckPath}(\langle \widehat{p_1}, \ldots, \widehat{p_n} \rangle, S_1, \ldots, S_n)$ *is the empty set, then the counterexample is spurious.* In this case, **CheckPath** calls a refinement procedure **PolyRefine** described in [5], and initiates a new round of model checking with the refined abstraction function. The algorithm $\mathbf{CheckLoop}$ performs an analogous (but more complicated) procedure for loop descriptors instead of path descriptors. **CheckPath** and **CheckLoop** are described in [5].

To obtain completeness for ACTL, it just remains to put everything together. Consider the algorithm **CheckRefine** in Figure 10. It takes as input an abstract state $\widehat{s_0}$ and a set of descriptors $\widehat{Q}$ which describes an abstract counterexample $\bigcup \widehat{Q}$. Algorithm **CheckRefine** generalizes **CheckPath** and **CheckLoop**, i.e., it outputs the set of all *concrete* states from which a *concrete* tree-like counterexample consistent with $\widehat{Q}$ can be found. To this end, **CheckRefine** traverses $\widehat{Q}$ in a bottom-up fashion, and uses **CheckPath** and **CheckLoop** for path and loop descriptors respectively.

9. Conclusion

We have shown that tree-like counterexamples are a natural and practically useful extension of the counterexample feature in model checking. In particular, we demonstrated that the presented framework is applicable to a large class of temporal logics, and can be used efficiently in the context of symbolic verification and counterexample-guided abstraction. Our future work will concentrate on the implementation and practical improvement of the presented methods.

References

[1] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, 1993.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[3] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. On ACTL formulas having deterministic counterexamples. *Journal of Computer and System Sciences*, 62(3):463–515, 2001.

[4] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, volume 1855 of *LNCS*, pages 154–169, 2000.

[6] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automation Conference (DAC)*, pages 427–432. IEEE Computer Society, 1995.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Publishers, 1999.

[8] E. Clarke and H. Schlingloff. Model checking. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2000. to appear.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[10] E. M. Clarke, O. Grumberg, and R. P. Kurshan. A synthesis of two approaches for verifying finite-state concurrent systems. In A. R. Meyer and M. A. Taitslin, editors, *Logic at Botik '89: Symposium on Logical Foundations of Computer Science*, volume 363, pages 81–90, Pereslavl-Zalessky, Russia, 1989. Springer-Verlag.

[11] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, Vol.16(No.5):pp.1512 – pp.1542, September 1994.

[12] M. Dam. CTL* and ECTL* as fragments of the modal $\mu-$calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[13] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.

[14] E. Emerson. Temporal and modal logic. In *Handbook of Theor.Comp.Science. Vol. B.*, pages 995–1072. Elsevier, 1990.

[15] T. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Proc. 14th Int. Coll. Automata, Languages, and Programming (ICALP)*, volume 267 of *LNCS*, pages 269–279. Springer, 1987.

[16] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In *Proc. International Conference on Computer Aided Verification (CAV)*, LNCS, 1993.

[17] O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996.

[18] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[19] M. Maidl. The common fragment of CTL and LTL. In *Proc. 41th Symp. on Foundations of Computer Science (FOCS)*, pages 643–652, 2000.

[20] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] A. P. Sistla. Theoretical issues in the design and analysis of distributed systems. In *PhD Thesis*. Harvard University, 1983.

[22] W. Thomas. Computation tree logic and regular ω-languages. In J. de Bakker, W.-P. de Roever, and G. Rosenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 690–713. Springer, 1989.

[23] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume 3, pages 389–455. Springer, 1997.

[24] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.