Counterexamples Revisited: Principles, Algorithms, Applications*

Edmund Clarke[1] and Helmut Veith[2]

[1] School of Computer Science, Carnegie Mellon University, USA
edmund.clarke@cs.cmu.edu
[2] Institut für Informationsysteme, Technische Universität Wien, Austria
veith@dbai.tuwien.ac.at

Abstract. Algorithmic counterexample generation is a central feature of model checking which sets the method apart from other approaches such as theorem proving. The practical value of counterexamples to the verification engineer is evident, and for many years, counterexample generation algorithms have been employed in model checking systems, even though they had not been subject to an adequate fundamental investigation. Recent advances in model checking technology such as counterexample-guided abstraction refinement have put strong emphasis on counterexamples, and have lead to renewed interest both in fundamental and pragmatic aspects of counterexample generation. In this paper, we survey several key contributions to the subject including symbolic algorithms, results about the graph-theoretic structure of counterexamples, and applications to automated abstraction as well as software verification.

Irrefutability is not a virtue of a theory (as people often think) but a vice.

Karl R. Popper

1 Introduction

Disproof by counterexample is an ancient mathematical concept which lends itself naturally to refute universal statements. Formally, a counterexample to a universal formula $\forall x \varphi(x)$ is given by a constant c for which $\varphi(c)$ evaluates to false. In their most visible form, mathematical counterexamples refute long-standing

* This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, the General Motors Collaborative Research Lab at CMU, the Austrian Science Fund Project N Z29-N04, and the EU Research and Training Network GAMES. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

N. Dershowitz (Ed.): Verification (Manna Festschrift), LNCS 2772, pp. 208–224, 2003.

conjectures, but much more often, counterexamples are natural byproducts in the early stages of a mathematical analysis; they typically exhibit pathological cases, refining our techniques and helping to shape the definitions we use.

In formal verification, the situation is similar. The specifications for a system – a piece of hardware or software – essentially amount to conjectures about a formal model of the system; although the specifications are expected to be true, there is a need for a systematic way to check for specification violations, and substantiate possible violations by concrete counterexamples. Experience has shown that counterexamples are the single most effective feature to convince system engineers about the value of formal verification.

Model checking [20,15,52] is an algorithmic framework tailored to perform this verification task; on a high level, model checking can be viewed as an exhaustive search algorithm which exploits various optimization strategies to find a counterexample. Consequently, it is possible, at least in principle, to have a model checker systematically output counterexamples for violated specifications. All practically successful model checkers including SMV [48,10,20] are able to output counterexamples in varying formats, cf. Figure 1.

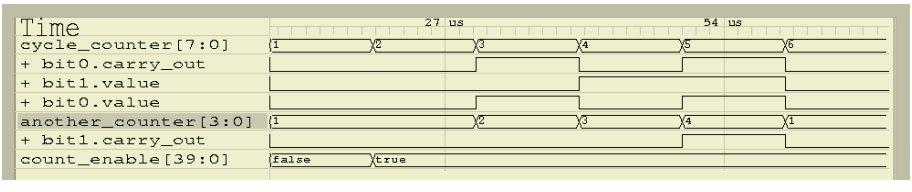

Fig. 1. Screenshot of a hardware counterexample.

The main practical problem in model checking is the combinatorial explosion of system states commonly known as the *state explosion problem*. It is the state explosion problem which renders apparently good theoretical results such as linear time model checking algorithms practically pointless, because the system size is so large that linear time is prohibitively expensive in practice. Most current research in model checking is therefore devoted to combating state explosion [18]. The context of the state explosion problem makes the counterexample feature even more important, because counterexamples are often composed only of several states.

Despite their practical importance, counterexamples often have been dealt with in an ad hoc manner, as if they were only parts of the user interface. In particular, they used to be a blind spot in fundamental research until recently. During the last few years however, counterexamples have attracted renewed interest; they have been increasingly recognized as data structures worth of a close algorithmic and logical analysis. This paper puts a spotlight on counterexamples. We survey theoretical and practical results, and focus in particular on recent contributions.

Section 3 deals with fundamental aspects, in particular with the graph-theoretic structure of counterexamples and their appropriateness for different temporal specification logics.

In Section 4 we demonstrate on two examples how counterexamples are used to improve the algorithmic efficiency of verification procedures. We first argue that bounded model checking can be viewed as a parameterized counterexample construction. Second, we present the CEGAR (*Counterexample-Guided Abstraction Refinement*) framework, various instantiations of which are used in state-of-the-art model checking systems.

In Section 5 we survey user-oriented applications of counterexamples in different frameworks, most notably in software verification, where ordinary counterexamples are only part of a more complex debugging process.

2 Temporal Logic Model Checking in a Nutshell

A model checking system usually comes with a specialized compiler which translates the system description into a data structure representing a Kripke structure. Given a set P of atomic propositions, a Kripke structure over P is a tuple $K = (S, I, R, L)$, where S is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : S \mapsto 2^P$ is a function that labels each state with a set of atomic propositions.

A *path* $\pi = s_0, s_1, s_2, \ldots$ starting at state s is an infinite sequence of states such that $s = s_0$ and $(s_i, s_{i+1}) \in R$ for all $i \in \mathbb{N}$. We write π^n to denote the suffix of π that begins at state s_n, and $\pi^{(n)}$ to denote the state s_n on π. When the context is clear, π^n sometimes is also used to denote the state $\pi^{(n)}$. We write $\mathrm{Paths}(s)$ to denote the set of paths starting at state s.

Specifications are expressed in terms of temporal logics such as CTL or LTL. (Note that the notation for CTL by Clarke and Emerson [15] was inspired by the paper [5] by Ben-Ari, Manna, and Pnueli.) CTL* is the logic obtained from propositional logic by introducing the unary modalities **X, G, F, E, A** and the binary modality **U**. State formulas and path formulas are defined inductively as follows: (i) Atomic formulas are state formulas. (ii) All state formulas are path formulas. (iii) State formulas are closed under conjunction, disjunction, and negation. (iv) If φ and ψ are state formulas, then $\mathbf{X}\,\varphi$, $\mathbf{G}\,\varphi$, $\mathbf{F}\,\varphi$ and $\varphi\,\mathbf{U}\,\psi$ are path formulas. (v) If φ is a path formula, then $\mathbf{E}\,\varphi$ and $\mathbf{A}\,\varphi$ are state formulas. The semantics of CTL* is defined in Figure 2. When it is clear from the context, we will omit K. We write $K \models \varphi$ to denote $K, s_0 \models \varphi$, where s_0 is the unique initial state in K.

Linear time logic LTL is the fragment of CTL* where each formula has the form $\mathbf{A}\psi$ where ψ does not contain **A** and **E**. CTL is the fragment of CTL* where each path operator **X, F, G, U** is immediately preceded by either **E** or **A**, i.e., temporal operators only occur in the compound form **EX, EG, EF, EU, AX, AG, AF, AU**. ACTL* is the fragment of CTL* where **E** does not occur, and negation is restricted to atomic subformulas. ACTL is the analogous fragment of CTL. We shall call ACTL, ACTL* and LTL *universal fragments* of CTL* .

$$
\begin{aligned}
K, s &\models p && :\Leftrightarrow p \in L(s)\\
K, \pi &\models \varphi_1 && :\Leftrightarrow K, \pi^{(0)} \models \varphi_1\\
K, s &\models \neg\varphi_1 && :\Leftrightarrow K, s \not\models \varphi_1\\
K, s &\models \varphi_1 \wedge \varphi_2 && :\Leftrightarrow K, s \models \varphi_1 \text{ and } K, s \models \varphi_2\\
K, s &\models \varphi_1 \vee \varphi_2 && :\Leftrightarrow K, s \models \varphi_1 \text{ or } K, s \models \varphi_2\\
K, \pi &\models \mathbf{X}\,\psi_1 && :\Leftrightarrow K, \pi^1 \models \psi_1\\
K, \pi &\models \mathbf{G}\,\psi_1 && :\Leftrightarrow \forall i \in \mathbb{N}.\ K, \pi^i \models \psi_1\\
K, \pi &\models \mathbf{F}\,\psi_1 && :\Leftrightarrow \exists i \in \mathbb{N}.\ K, \pi^i \models \psi_1\\
K, \pi &\models \psi_1\,\mathbf{U}\,\psi_2 && :\Leftrightarrow \exists n \in \mathbb{N}\ \forall i < n.\ K, \pi^i \models \psi_1 \text{ and } K, \pi^n \models \psi_2\\
K, s &\models \mathbf{E}\,\psi_1 && :\Leftrightarrow \exists \pi \in \mathrm{Paths}(s).\ K, \pi \models \psi_1\\
K, s &\models \mathbf{A}\,\psi_1 && :\Leftrightarrow \forall \pi \in \mathrm{Paths}(s).\ K, \pi \models \psi_1
\end{aligned}
$$

Fig. 2. Semantics of CTL*, assuming that φ_1 and φ_2 are state formulas, and that ψ_1 and ψ_2 are path formulas.

The model checking problem for a logic L is to decide if for given K, s and $\varphi \in L$ it holds that $K, s \models \varphi$. For CTL* and LTL, this problem is PSPACE-complete [56], while for CTL, the time complexity is linear in the size of both K and φ. Note that due to state explosion, however, the dominant factor in this analysis is often K.

Model checking algorithms can be classified into *explicit* and *symbolic* algorithms. Symbolic algorithms employ data structures such as Binary Decision Diagrams [8] to describe sets of states; in many cases, symbolic algorithms achieve great reductions in the size of the data structures, and thus help to alleviate the state explosion problem [10]. Explicit algorithms in contrast are algorithms which work directly on the Kripke structure, and construct necessary parts of the Kripke structure on the fly, using methods such as partial order reduction to prune the search space. Traditionally, symbolic methods have been used primarily for CTL, while explicit methods are typical of LTL model checking. Examples of symbolic and explicit model checkers are SMV [48,11,20] and SPIN [38,39] respectively.

3 What Is a Counterexample?

Suppose that your favorite model checking tool determines that the specification φ is violated over Kripke structure K, i.e., $K \not\models \varphi$. Since K is very large, we expect the model checker to provide a counterexample C which explains the violation [12]. As φ ranges over paths, φ is essentially a second order formula over Kripke structures, and thus, the counterexample C will be a Kripke structure. In order for a structure C to be a counterexample, C has to satisfy two properties, cf. [22]:

(i) C *violates* φ, i.e., $C \not\models \varphi$.
(ii) The violation of φ on C *"explains"* the violation on K in a rigorous manner.

The simplest solution for C would be to use $C = K$ as a counterexample. While mathematically correct, this is evidently a simple-minded choice. Consequently, it is natural to investigate restricted classes of counterexamples which are simple enough to be practically useful, but rich enough to exhibit all behaviors which are relevant for the violation of a specification. Based on the analysis in [22] we identify the following criteria for good classes C of counterexamples:

- **Completeness.** C should be complete for a tangible class L of specifications, i.e., each violation of a specification in L is witnessed by a suitable counterexample in C.
- **Intelligibility.** The elements of C should be simple and specific enough to be analysed by human engineers, possibly with the aid of automated tools and suitable annotations.
- **Uniformity.** All counterexamples should be related to the specification and the system by a uniform principle which explains the property violation; for example, in [22], this principle was given by the simulation relation.
- **Effectiveness.** There should be effective algorithms for generating and manipulating counterexamples in the class C; in particular, computation of counterexamples for a logic L should not be harder than the model checking problem for L.

We say that C is a *viable* class for L, if C fulfills these criteria. Note that we do not give a formal definition of viable classes of counterexamples, as notions such as intelligibility cannot be captured mathematically.

In the rest of this paper we will concentrate on counterexamples for universal logics such as ACTL, LTL, and ACTL*, because we can expect to have simple natural counterexamples only for universal specifications. Moreover, universal logics are the formalism of choice for counterexample-guided abstraction refinement.

3.1 Counterexamples for Linear Time Logic

The simplest kind of counterexamples refute invariants, i.e., finite paths where the last state in the path exhibits the violation of the alleged invariant. The class of LTL properties which are refutable by finite paths has been studied under the name of safety properties [47].

Fact 1. *Finite paths are viable counterexamples for LTL safety properties.*

Finite paths obviously satisfy our requirements. The uniform relationship between the path π and the Kripke structure K is given by the fact that the path is contained in $\mathrm{Paths}(s_0)$ where s_0 is the initial state of K.

Note that π is not a substructure of K, as π may contain the same state several times. Indeed, we can view π as a substructure of the *indexed Kripke structure* K^ω defined in [22]:

Definition 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* $L^\omega(s^i) = L(s)$.

Intuitively, K^ω is obtained 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. It is easy to see that the Kripke structures K and K^ω are bisimilar. *Unless noted otherwise, all counterexamples will be substructures of* K^ω.

A natural generalization of finite paths are *traces*. A trace is either a finite path or a loop, i.e., a finite path which leads to a finite cycle, cf. Figure 3.

Fact 2. *Traces are viable counterexamples for LTL.*

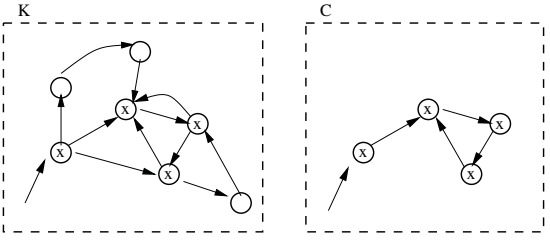

Fig. 3. A simple trace counterexample for $\mathbf{AF}\,\neg x$

Again, it is simple to observe that the requirements are fulfilled. Indeed, completeness follows easily from automata-theoretic methods: as both $\mathrm{Paths}(s_0)$ and the set of traces where φ is violated are recognized by Büchi automata, there exists a Büchi automaton for the intersection from which a trace can be extracted; see [20] for details. Consequently, traces are also viable counterexamples for extensions of LTL by regular temporal operators, e.g. ETL [59]. With explicit-state model checkers, trace counterexamples are a natural byproduct of the search procedure and are provided automatically with little overhead. A systematic analysis of the length of LTL counterexamples has recently been done in [23].

3.2 Counterexamples for Branching Time Logic

For branching time logics the situation is more complicated than for linear time logics. Consider for example the formula $\mathbf{AF}x \vee \mathbf{AF}y$. Semantically, a counterexample for this formula has to describe two traces, one where $\neg x$ holds globally, and one where $\neg y$ holds globally. In other words, a counterexample for $\mathbf{AF}x \vee \mathbf{AF}y$ has to be a model for $\mathbf{EG}\neg x \wedge \mathbf{EG}\neg y$. A more involved example is

given by formulas such as $\mathbf{AFAX}x$ where a counterexample has to provide an infinite path along which from every state $\neg x$ can be reached in one step.

(a) Trace Counterexamples. Most model checkers including SMV compute trace counterexamples for CTL specifications. Symbolic algorithms for computing trace counterexamples have been described early on [28,37]. Trace counterexamples are certainly very useful in practice because they provide an easy and intuitive means to study system behavior. The above examples however demonstrate that trace counterexamples are *not complete* even for ACTL.

This startling mismatch has been a blind spot in model checking for many years; only recent papers [9,22] have explicitly considered the adequacy of trace counterexamples for ACTL. Trace counterexamples are closely related to the linear fragment of ACTL, i.e. ACTL $\cap$ LTL. To our best knowledge, this relationship was never made explicit in the literature, and is expressed in the following proposition:

Proposition 1. *Let φ be an ACTL formula. φ is trace-refutable iff φ is expressible in LTL.*

Following [14], we introduce the following notation: For a CTL* formula φ, let φ^d denote the formula obtained by removing all occurrences of $\mathbf{E}$ and $\mathbf{A}$.

Proof. If φ is expressible in LTL, then it has trace counterexamples by Fact 2. For the other direction, consider the ECTL formula $\bar{\varphi}$ obtained from $\neg\varphi$ by distributing the negation to atomic subformulas. By assumption, we know that φ is violated on a model M iff $\bar{\varphi}$ is true on M iff there exists a single path π starting at the initial state of M such that all witnesses for the existential quantifiers $\mathbf{E}$ in $\bar{\varphi}$ are suffixes of π. Consequently, $\bar{\varphi}$ is equivalent to $\mathbf{E}\varphi^d$, and φ is equivalent to $\mathbf{A}\neg\varphi^d$ which is in LTL.

Consequently, investigating trace-refutable ACTL amounts to investigating the linear time fragment of ACTL. We can therefore draw from quite a rich body of results dealing with the relationship between branching and linear time.

- Clarke and Draghicescu have shown that an ACTL formula φ is expressible in LTL iff φ is equivalent to $\mathbf{A}\varphi^d$ [14]. Thus, if φ is expressible in LTL, it is easy to find the corresponding LTL formula.
- Concerning this equivalence, it is easy to see that $\varphi \Rightarrow \mathbf{A}\varphi^d$ holds for ACTL specifications φ. Kupferman and Vardi however have shown that the decision problem whether φ is implied by $\mathbf{A}\varphi^d$ is PSPACE-hard, and contained in EXPSPACE; over a fixed Kripke structure M, the problem to decide if $\mathbf{A}\varphi^d \Rightarrow \varphi$ is PSPACE-complete [43]. Partial hardness results were independently proved by Buccafurri et al. in [9].
- Maidl finally solved the open problem of finding a logic capturing the intersection of ACTL and LTL. She semantically characterized ACTL $\cap$ LTL by a fragment ACTL$^{\mathrm{det}}$ [46] defined as follows:

If φ_1 and φ_2 are in ACTL$^{\text{det}}$, and p is an atomic formula, then p, $\varphi_1 \wedge \varphi_2$, $(p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)$, $\mathbf{AX}\varphi_1$, $\mathbf{A}(p \wedge \varphi_1)\mathbf{U}(\neg p \wedge \varphi_2)$, $\mathbf{A}(p \wedge \varphi_1)\mathbf{W}(\neg p \wedge \varphi_2)$ are in ACTL$^{\text{det}}$.

She also showed that given an ACTL formula φ, it is PSPACE-complete to decide if φ is equivalent to a formula in ACTL$^{\text{det}}$ which by Theorem 1 below closes the gap between PSPACE and EXPSPACE left open by [43].

Although most of the mentioned papers do not deal with counterexamples explicitly, in conjunction with Proposition 1 they provide us with a clear picture about the relationship between trace counterexamples, ACTL and CTL:

Theorem 1. *Let φ be an ACTL formula. The following are equivalent:*

1. *φ has trace counterexamples.*
2. *φ is expressible in LTL.*
3. *φ is equivalent to a formula in ACTL$^{\text{det}}$.*
4. *$\mathbf{A}\varphi^d \Rightarrow \varphi$.*
5. *φ is equivalent to $\mathbf{A}\varphi^d$.*

The corresponding decision problem is PSPACE-complete.

Let K be a Kripke structure, and φ be an ACTL formula, s.t. $K \not\models \varphi$. The following are equivalent:

1. *φ has a trace counterexample on K.*
2. *$K \models \mathbf{A}\varphi^d \Rightarrow \varphi$.*
3. *On K, φ is equivalent to $\mathbf{A}\varphi^d$.*

The corresponding decision problem is PSPACE-complete.

We conclude that although CTL model checking is in linear time, computing a trace counterexample is a hard problem. Consequently, unless P = PSPACE, for every polynomial CTL model checker which outputs trace counterexamples, there exist infinitely many cases where the trace counterexample produced by the model checker is not complete. Moreover it follows that the syntactic fragment of ACTL which guarantees the existence of a trace counterexample cannot be captured by simple syntactic means, as the decision procedure is PSPACE-complete.

This motivates the concept of ACTL templates introduced by Buccafurri et al. [9]. An ACTL template is an ACTL formula where $\star$ is the single atomic proposition used. An instantiation of a template is obtained by replacing each occurrence of $\star$ in the template by a pure state formula. Buccafurri et al. show that there exists a unique maximal set of ACTL templates whose instantiations guarantee trace counterexamples. This set is given by the context-free grammar LIN in BNF notation

$$\begin{aligned}
\text{LIN} &= \text{PSF} \mid (\text{LIN} \wedge \text{LIN}) \mid (\text{LIN} \vee \text{PSF}) \mid (\text{PSF} \vee \text{LIN}) \mid \\
&\quad \mathbf{AX}(\text{LIN}) \mid \mathbf{A}(\text{PSF}\ \mathbf{V}\ \text{LIN}) \mid \text{UL} \\
\text{UL} &= \mathbf{A}(\text{LIN}\ \mathbf{U}\ \text{PSF}) \mid \mathbf{A}(\text{PSF}\ \mathbf{U}\ \text{UL}) \mid (\text{UL} \vee \text{PSF}) \mid (\text{PSF} \vee \text{UL}) \\
\text{PSF} &= (\text{PSF} \wedge \text{PSF}) \mid (\text{PSF} \vee \text{PSF}) \mid \neg(\text{PSF}) \mid \star
\end{aligned}$$

where PSF denotes the set of pure state formulas. (The operator $\mathbf{V}$ is defined by $\varphi \mathbf{U} \psi$ iff $\neg(\neg\varphi \mathbf{V} \neg\psi)$.) Membership in LIN is efficiently decidable; moreover they show that for all specifications in LIN, counterexamples can be efficiently constructed.

In conclusion, we have two possibilities to use trace counterexamples as viable counterexamples for a fragment of ACTL:

Fact 3. *Traces are viable counterexamples for LIN and for ACTL*$^{\mathrm{det}}$.

There is a trade-off between the two languages: In the case of LIN, we have a flexible syntax which makes it possible to formulate many ACTL specifications in the usual manner at the cost of restricting the expressive power; in the case of ACTL$^{\mathrm{det}}$ we obtain the maximum expressive power, yet we need to express the specifications in a somewhat artificial language into which the linear fragment of ACTL cannot be efficiently translated unless PSPACE collapses to P.

The most significant disadvantage of trace counterexamples however is that we restrict the expressive power of branching time logic to its linear fragment; hence the results described in this section shed a clear light on an unsatisfactory situation.

(b) Tree-Like Counterexamples. In a recent paper [22], Clarke et al. suggested a new notion of counterexamples for ACTL. They introduced tree-like counterexamples which generalize both trace counterexamples and tree models. Intuitively, tree-like Kripke structures are obtained by gluing together traces in a finite tree, cf. Figure 4.

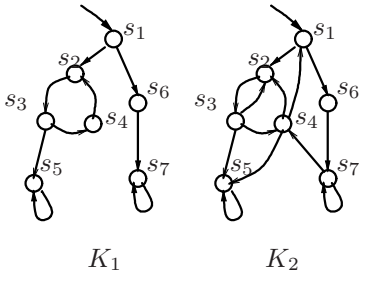

Fig. 4. K_1 is tree-like, K_2 is not.

Formally, let G be a directed graph. The component graph $c(G)$ is a graph whose vertices are given by the strongly connected components (SCCs) of G, and where two vertices are connected by an edge if there exists an edge between vertices in the corresponding SCCs. A graph is tree-like, if (i) all SCCs are either cycles or simple nodes, and (ii) the component graph is a directed tree. A Kripke structure C is tree-like if its transition relation is tree-like, and its root is the initial state s_0 of C.

Theorem 2. *[22] Tree-like Kripke structures are viable counterexamples for ACTL* .*

Example 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 5. The shaded areas indicate which subformula is disproved. 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 5.

As the examples demonstrate, tree-like counterexamples are indeed easy to understand. The algorithms of [22] can provide annotations to be used by an interactive browser which facilitates manual inspection of tree-like counterexamples. Due to the absence of complicated cycles navigation within the counterexamples is relatively simple. As tree-like counterexamples can be viewed as compositions of trace counterexamples, they tie in well with the existing work on counterexamples. In particular, the refinement techniques necessary for counterexample-guided refinement extend easily to the case of tree-like counterexamples. On a more general note, tree-like graphs have many favorable algorithmic properties: having a treewidth of 2, they share many favorable algorithmic properties of finite trees.

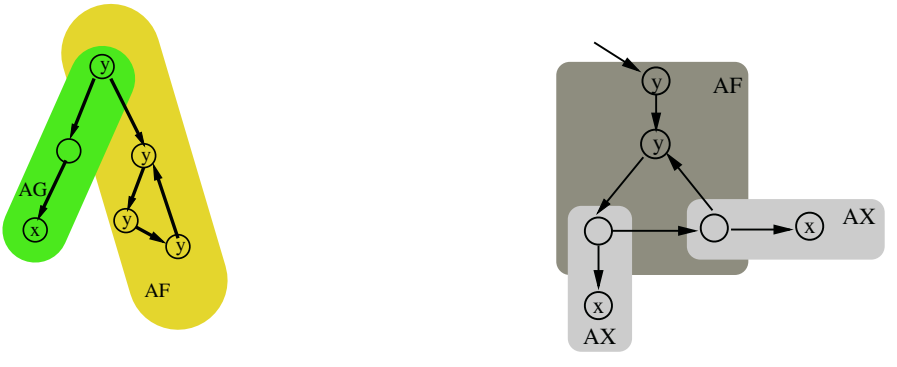

Fig. 5. Tree-like Counterexamples for $\mathbf{AG}\,\neg x \vee \mathbf{AF}\,\neg y$ and $\mathbf{AF}(\neg y \wedge \mathbf{AX}\,\neg x)$.

The result of Theorem 2 indeed was proved for extensions of ACTL by ω-regular linear time temporal operators and by infinitary conjunctions, cf. [22]. In addition, the paper shows how to extend the symbolic counterexample algorithms of [28] to tree-like counterexamples.

(c) Counterexamples for Existential Formulas. It is easy to see that counterexamples for universal formulas are witnesses for the dual existential formulas, and vice versa. Consider for example the existential formula $\mathbf{EF}x$. A

counterexample for this formula has to be a witness for the formula $\mathbf{AG}\neg x$. It is evident that a witness for $\mathbf{AG}\neg x$ has to contain all reachable states, and this is usually infeasible due to state explosion. There may however occur situations where certain existential formulas also can be refuted by counterexamples of a reasonable size.

Loosely speaking, **EF** is *very* existential because both **E** and **F** amount to existential quantification. Consequently, as argued above, it is usually not possible to generate counterexamples for **EF**. The situation is somewhat different for formulas such as $\mathbf{EG}x$ where the **G** operator is defined by a universal property: A counterexample for $\mathbf{EG}x$ may be feasible in certain situations, because witnesses for $\mathbf{AF}\neg x$ may be small even for large Kripke structures. Formulas such as $\mathbf{EX}x$ have witnesses whose size is bounded by the maximum out-degree of the nodes in the Kripke structure. For $\mathbf{E}x\mathbf{U}y$, the counterexample is potentially simple only in case that the counterexample satisfies $\mathbf{A}x \wedge \neg y\mathbf{U}\neg x \wedge \neg y$. Using the methods of [22], it is easy to see that tree-like counterexamples can in principle be generated for all CTL formulas where **EU** or **EF** does not occur; in the latter case, the only reasonable counterexample is given by the Kripke structure, reduced to the set of reachable states.

In recent work, Shankar and Sorea describe counterexamples for full CTL obtained by storing the stages of a fixpoint computation [53], while Shoman and Grumberg use annotations to deal with full CTL [55]. In both cases, however, completeness for CTL is obtained at the cost of compromising the conceptual simplicity of traces and tree-like counterexamples.

As pointed out by Grädel and Sistla [57,31], the results about tree-like counterexamples tie in nicely with classic results about tree automata [40]: every satisfiable property described by a tree automaton has a back edge model, i.e., a model obtained from a finite tree by adding at most one "back edge" from each leaf of the tree to one of its ancestor nodes. Consequently, back edge models are candidates for viable counterexamples for complicated branching time properties described by tree automata. It is not hard to see that tree-like models are special cases of back edge models, similarly as traces are special cases of tree-like models; back edge models however can be complicated to analyse, as they can have interfering cycles. Thus, the relative simplicity of CTL in comparison with tree automata is mirrored by the simplicity of its counterexamples.

4 Introverted Counterexamples: Counterexamples as Internal Data Structures

In this section we concentrate on "internal" applications of counterexamples, where counterexamples are not only used as output for the verification engineer, but are essential to the verification algorithm.

4.1 Bounded Model Checking

Bounded Model Checking [7,6] is a relatively new approach to LTL model checking which employs highly effective SAT-checkers. From a high-level perspective,

bounded model checking can be viewed as a reduction to Boolean satisfiability. For a given bound k, the SAT instance describes a trace counterexample of size k whose state transitions are parameterized by Boolean variables. The SAT-checker then verifies whether this parameterized counterexample is consistent with the actual transitions in K. The good performance of bounded model checkers such as **BMC** [7] relies both on a subtle construction of the SAT instance, and on the usage of state-of-the-art SAT-checkers such as CHAFF [49].

One of the main advantages of bounded model checking is that it avoids state explosion, i.e., memory usage is kept in reasonable bounds, as the SAT procedures do not generate large additional data structures. The evident disadvantage is the incompleteness of the method, because in each run, only counterexamples of size $\leq k$ are considered. Thus, bounded model checking achieves a trade-off between the high memory requirements of standard LTL model checking (as indicated by PSPACE-completeness) and the possibly large running time of a memory-efficient SAT procedure which needs to be iterated for completeness. A recent thorough analysis of the complexity of bounded model checking can be found in [23].

Practical experiments show that bounded model checking is a very efficient approach to finding minimal counterexamples. In particular, there are many cases where standard methods failed because the construction of a binary decision diagram exceeds the available memory, while the bounded model checker quickly determined a counterexample [7,6]. Bounded model checkers have rapidly found industrial applications [24].

4.2 CEGAR: Counterexample-Guided Abstraction Refinement

The methods for alleviating the state explosion problem in model checking can be classified coarsely into *symbolic methods* and *abstraction methods* [18]. By symbolic methods we understand the use of succinct data structures and symbolic algorithms which help keep state explosion under control by compressing information, using, e.g., binary decision diagrams or efficient SAT procedures.

Abstraction methods in contrast attempt to reduce the size of the state space by employing knowledge about the system and the specification in order to model only relevant features in the Kripke structure. An abstraction function h associates the Kripke structure K with an abstract Kripke structure $\widehat{K}$ such that two properties hold:

- **Feasibility.** $\widehat{K}$ is significantly smaller than K.
- **Preservation.** $\widehat{K}$ preserves all behaviors of K.

Preservation ensures that every universal specification which is true in $\widehat{K}$ is also true in K. The converse implication, however, will not hold in general: a universal property which is false in $\widehat{K}$ may still be true in K. In this case, the counterexample obtained over $\widehat{K}$ cannot be reconstructed for the concrete Kripke structure K, and is called a *spurious counterexample* [17], or a false negative.

An important example of abstraction is *existential abstraction* [19] where the abstract states are essentially taken to be equivalence classes of concrete states;

a transition between two abstract states holds if there was a transition between any two concrete member states in the corresponding equivalence classes.

In certain cases, the user knowledge about the system will be sufficient to allow manual determination of a good abstraction function. In general, however, finding abstraction functions gives rise to the following dichotomy:

- If $\widehat{K}$ is too small, then spurious counterexamples are likely to occur.
- If $\widehat{K}$ is too large, then verification remains infeasible.

Counterexample-Guided Abstraction Refinement is a natural approach to resolve this situation by using an adaptive algorithm which gradually improves an abstraction function by analysing spurious counterexamples.

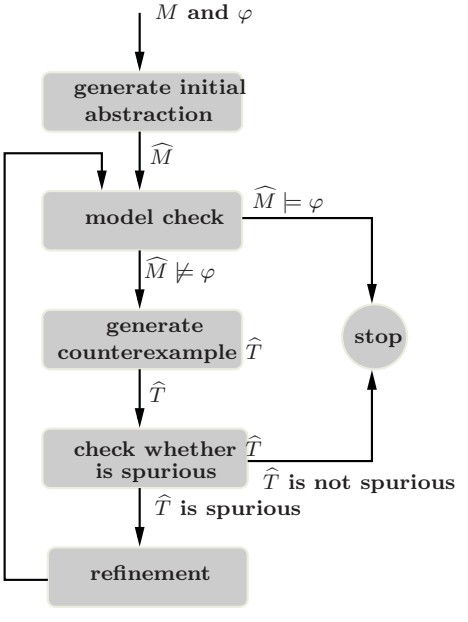

Fig. 6. Counterexample-guided refinement.

(i) **Initialization.** Generate an initial abstraction function.
(ii) **Model Checking.** Verify the abstract model. If verification is successful, the specification is correct, and the algorithm terminates successfully. Otherwise, generate an abstract counterexample $\widehat{T}$ on the abstract model.
(iii) **Sanity Check.** Determine, if the counterexample $\widehat{T}$ is spurious. If a concrete counterexample T can be generated, the algorithm outputs this counterexample and terminates.
(iv) **Refinement.** Refine the abstraction function in such a way that the spurious counterexample is avoided, and return to step 2.

Using counterexamples to refine abstract models has been investigated by several researchers beginning with the *localization reduction* of Kurshan [44] where the model is abstracted/refined by removing/adding variables from the system description. A similar approach has been described by Balarin in [1].

A systematic account of counterexample guided abstraction refinement for CTL model checking was given in [17,22]. Here, the initial abstraction is obtained using predicate abstraction [32] in combination with a simple static analysis of the system description; all other steps use BDD-based techniques. The use of tree-like counterexamples guarantees that the method is complete for ACTL.

During the last few years, the CEGAR paradigm has been adapted to different projects and verification frameworks, both for hardware and software verification [45,30,27,26,4,3,16,21,36,13]. The major improvements to the method include, most notably, the integration of SAT solvers for both verification and refinement, and the use of multiple spurious counterexamples.

It is well-known that most abstraction methodologies can be paraphrased in the framework of abstract interpretation by Cousot and Cousot [25]. Giacobazzi and Quintarelli [29] have shown that, not surprisingly, this holds true for counterexample-guided abstraction refinement as well. The practical and computational significance of such embeddings for verifying real-life systems however remains controversial.

5 Extroverted Counterexamples: Analysis and Debugging Tools

Despite the evident explanatory value of counterexamples as presented in this paper, an error trace may contain much information which is irrelevant for the error, or may be just too large for efficient manual inspection. While for hardware the concept of an error trace is natural for engineers, the strong abstraction methods necessary in software verification incur a perceptional gap between the plain error trace and the original program.

Several recent papers have dealt with the question of analyzing trace counterexamples. Kupferman and Vardi defined a notion of interesting witnesses where the temporal formulas are satisfied in a non-vacuous manner [42]. Jin et al propose an algorithm to generate an error trace divided into fated and free segments [41]. Moreover, several approaches to compute proof-like annotated counterexamples have been presented [35,55,34,58,50].

Failure Diagnosis for software invariants was proposed in the context of NASA's Java PathFinder [33], and Microsoft's SLAM model checker for C [2]. In these approaches, multiple counterexamples are generated in order to extract the real error cause from a systematic comparison of the counterexamples. The question of mapping counterexample traces back to source code has also been studied by the Bandera model checking frontend for Java [51]. Counterexamples have also been used in security for generating attack graphs by Sheyner et al [54].

6 Conclusion

Counterexamples belong to the fundamental concepts of model checking which have been introduced to the method early on [12]. During the last years, the significance of counterexamples has been increasingly recognized, and counterexamples have been studied in their own right. Besides their significance for the user, counterexamples play a crucial algorithmic role in algorithms for bounded model checking and abstraction refinement.

Acknowledgments

The authors extend their gratitude to S. Chaki, P. Chauhan, E. Grädel, A. Groce, S. Jha, A. Sistla, O. Strichman, and D. Wang for valuable discussions.

References

1. F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, 1993.
2. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing erros in counterexample traces. In *Annual ACM Symposium on Principles of Programming Languages*, 2003.
3. T. Ball and S. K. Rajamani. Getting abstract explanations of spurious counterexamples in C programs, 2002. Microsoft Technical Report MSR-TR-2002-09.
4. S. Barner, D. Geist, and A. Gringauze. Symbolic localization reducation with reconstruction layering and backtracking. In *CAV 2002*, volume 2404 of *LNCS*, pages 65–77, 2002.
5. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Inf.*, 20:207–226, 1983. Full version of POPL'81 paper.
6. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference*, pages 317–320, 1999.
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, number 1579 in LNCS. Springer-Verlag, 1999.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, Aug. 1986.
9. 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.
10. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
11. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.
12. E. Clarke, S. Bose, M. C. Browne, and O. Grumberg. The design and verification of finite state hardware controllers. In *Int. Symposium on VLSI Technology, Systems, and Applications*, pages 53–61, 1987.
13. E. Clarke, S. Chaki, S. Jha, and H. Veith. Strategy-guided abstraction refinement, 2003.

14. E. Clarke and I. A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354, pages 428–437. Springer-Verlag: Lecture Notes in Computer Science, 1988.
15. E. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, LNCS, 1981.
16. E. Clarke, A. Fehnker, Z. Han, B. H. Krogh, O. Stursberg, and M. Theobald. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *TACAS'03*, pages 192–207, 2003.
17. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000. Extended version to appear in J.ACM.
18. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics, 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 176–194, 2001.
19. E. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
20. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
21. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction - refinement using ILP and machine learning techniques. In E. Brinksma and K. Larsen, editors, *Computer-Aided Verification*, volume 2404 of *LNCS*, pages 265–279, Copenhagen, Denmark, July 2002. Springer.
22. E. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *Proc. Logic in Computer Science (LICS)*, 2002.
23. E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. 2003.
24. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Computer-Aided Verification*, pages 436–453, 2001.
25. P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *ACM Symposium of Programming Language*, pages 238–252, 1977.
26. S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS*, pages 51–60, 2001.
27. S. Das and D. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*, pages 19–32, 2002.
28. 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.
29. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *SAS'01*, pages 356–373, 2001.
30. M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *TACAS'03*, pages 176–191, 2003.
31. E. Grädel. Private Communication, 2002.
32. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, June 1997.
33. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop*, 2003.

34. A. Gurfinkel and M. Chechik. Generating counterexamples for multi-valued model checking. In *FME*, 2003.
35. A. Gurfinkel and M. Chechik. Proof-like counterexamples. In *TACAS*, 2003.
36. T. A. Henzinger, R. Jhala, and R. Majumdar. Counterexample guided control. In *ICALP*, 2003.
37. 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.
38. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
39. G. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.
40. R. Hossley and C. Rackoff. The emptiness problem for automata on infinite trees. In *FOCS'72*, pages 121–124, 1972.
41. H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS*, volume 2280 of *LNCS*, pages 445–459, 2002.
42. O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. In *Formal Aspect of System Design*, pages 82–96, 1999.
43. O. Kupferman and M. Y. Vardi. Module checking. In *Proc. 8th Int'l. Conf. on Computer-Aided Verification*, Lecture Notes in Computer Science 1102, pages 75–86. Springer-Verlag, 1996.
44. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
45. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS'01*, pages 98–112, 2001.
46. M. Maidl. The common fragment of CTL and LTL. In *Proc. 41th Symp. on Foundations of Computer Science (FOCS)*, pages 643–652, 2000.
47. Z. Manna and A. Pnueli. *Temporal Verifications of Reactive Systems - Safety*. Springer-Verlag, 1995.
48. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
49. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: engineering an efficient SAT solver. In *DAC*, 2001.
50. K. S. Namjoshi. Certifying model checkers. In *Computer-Aided Verification*, pages 2–13, 2001.
51. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *TACAS'01*, pages 284–298, 2001.
52. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
53. N. Shankar and M. Sorea. Counterexample-driven model checking, 2003.
54. O. Sheyner, S. Jha, and J. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284, 2002.
55. S. Shoman and O. Grumberg. A game-based framework for CTL counter-examples and 3-valued abstraction refinement. In *CAV 2003*, 2003.
56. A. S. Sistla and E. Clarke. The complexity of propositional linear temporal logics. *J. Assoc. Comput. Mach.*, 32(3):733–749, 1985.
57. P. Sistla. Private Communication, 2002.
58. L. Tan and R. Cleaveland. Evidence-based model checking. In *Computer-Aided Verification*, pages 455–470, 2002.
59. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.