Counterexamples in Model Checking – A Survey

Hichem Debbi
Department of Computer Science, University of Mohamed Boudiaf, M'sila, Algeria
E-mail: hichem.debbi@gmail.com

Overview paper

Keywords: model checking, counterexamples, debugging

Received: December 9, 2016

Model checking is a formal method used for the verification of finite-state systems. Given a system model and such specification, which is a set of formal properties, the model checker verifies whether or not the model meets the specification. One of the major advantages of model checking over other formal methods its ability to generate a counterexample when the model falsifies the specification. Although the main purpose of the counterexample is to help the designer to find the source of the error in complex systems design, the counterexample has been also used for many other purposes, either in the context of model checking itself or in other domains in which model checking is used. In this paper, we will survey algorithms for counterexample generation, from classical algorithms in graph theory to novel algorithms for producing small and indicative counterexamples. We will also show how counterexamples are useful for debugging, and how we can benefit from delivering counterexamples for other purposes.

Povzetek: Pregledni članek se ukvarja s protiprimeri v formalni metodi za preverjanje končnih avtomatov, tj. sistemov manjše računske moči kot Turingovi stroji. Protiprimeri koristijo snovalcem na več načinom, predvsem kot način preverjanja pravilnosti delovanja.

1 Introduction

Model checking is a formal method used for the verification of finite-state systems. Given a system model and such specification, which is a set of formal properties in temporal logics like LTL [109] and CTL [28, 52], the model checker verifies whether or not the model meets the specification. One of the major advantages of model checking over other formal methods its ability to generate a counterexample when the model falsifies such specification. The counterexample is an error trace, by analysing it, the user can locate the source of the error. The original algorithm for counterexample generation was proposed by [31], and was implemented in most symbolic model checkers. This algorithm of generating linear counterexamples for ACTL, which is a fragment of CTL, was later extended to handle arbitrary ACTL properties using the notion of tree-like counterexamples [36]. Since then, many works have addressed this issue in model checking. Counterexample generation has its origins in graph theory through the problem of fair cycle and Strongly Connected Component (SCC) detection, because model checking algorithms of temporal logics employ cycle detection and technically a finite system model is determining a transition graph [32]. The original algorithm for fair cycle detection in LTL and CTL model was proposed by [53]. Since then, many variants of this algorithm and new alternatives were proposed for LTL and CTL model checking. In section 3 we will investigate briefly the problem of fair cycles and SCCs detection.

While the early works introduced by [28, 52] have investigated the problem of generating counterexample so widely, which led to practical implementation within well-known model checkers, the open problem that emerged was the quality of the counterexample generated and how it really serves the purpose. Therefore, in the last decade many papers have considered this issue, earlier in terms of structure[36], by proposing the notion of tree-like counterexamples to handle ACTL properties, and followed later by the works investigating the quality of the counterexample mostly in terms of length to be useful later for debugging. In section 3, we will investigate the methods proposed for generating minimal, small and indicative counterexamples in conventional model checking. Model checking algorithms are classified in two main categories, explicit and symbolic. While explicit algorithms are applied directly on the transition system, symbolic algorithms employ specific data structures. Generally, the explicit algorithms are adopted for LTL model checking, whereas symbolic algorithms are adopted for CTL model checking. In this section, the algorithms for generating small counterexamples are presented with respect to each type of algorithms.

However, generating small and indicative counterexamples only is not enough for understanding the error. Therefore, counterexamples analysis is inevitable. Many works in model checking have addressed the analysis of counterexamples to better understand the error. In section 4, we will investigate the approaches that aim to help the designer to localize the source of the error given counterexamples. In this section, we consider that most of these methods range into two main categories: those that are applied on the counterexample itself without any need to other information, and those that require successful runs or witnesses to be compared with the counterexamples.

Probabilistic model checking has appeared as an extension of model checking for analyzing systems that exhibit stochastic behavior. Several case studies in several domains have been addressed from randomized distributed algorithms and network protocols to biological systems and cloud computing environments. These systems are described usually using Discrete-Time Markov Chains (DTMC), Continuous Time Markov Chains (CTMC) or Markov Decision Processes (MDP), and verified against properties specified in Probabilistic Computation Tree Logic (PCTL)[78] or Continuous Stochastic Logic (CSL) [9, 10]. In probabilistic model checking (PMC) counterexample generation has a quantitative aspect. The counterexample is a set of paths in which a path formula holds, and their accumulative probability mass violates the probability bound. Due to its specific nature, we specify section 5 for counterexample generation in probabilistic model checking. As it was done in conventional model checking, addressing the error explanation in the probabilistic model checking is highly required, especially that probabilistic counterexample consists of multiple paths instead of a single path, and it is probabilistic. So, in this section we will also investigate the counterexample analysis in PMC.

The most important thing about counterexample is that it does not just serve as a debugging tool, but it is also used to refine the model checking process itself, through Counterexample Guided Abstraction Refinement(CEGAR)[37]. CEGAR is an automatic verification method mainly proposed to tackle the problem of state-explosion problem, and it is based on the information obtained from the counterexamples generated. In section 6, we will show how counterexample contributes to this famous method of verification.

Testing is an automated method used to verify the quality of software. When we use model checking to generate test cases, this is called model-based testing. This method has known a great success in the industry through the use of famous model checkers such as SPIN, NuSMV and Java Pathfinder. Model checking is used for testing for two main reasons: first, because model checking is fully automated, and secondly and more importantly because it delivers counterexamples when the property is not satisfied. In section 7, we will show how counterexample serves as a good tool for generating test cases.

Although counterexample generation is in the heart of model checking, not all model checkers deliver counterexamples to the user. In the last section, we will review the famous tools that generate counterexamples. Section 9 concludes the paper, where some brief open problems and future directions are presented.

2 Preliminaries and definitions

Kripke Structure. A Kripke structure is a tuple $M = (AP, S, s_0, R, L)$ that consists of a set AP of atomic propositions, a set S of states, $s_0 \in S$ an initial state, a total transition relation $R \subseteq S \times S$ and a labelling function $L : W \to 2^{AP}$ that labels each state with a set of atomic propositions.

Büchi Automaton. A Büchi automaton is a tuple $B = (S, s_0, E, \sum, F)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $E \subseteq S \times S$ is the transition relation, $\sum$ is a finite alphabet, and $F \subseteq S$ is the set of accepting or final states.

We use Büchi automaton to define a set of infinite words of an alphabet. A path is a sequence of states $(s_0 s_1 ... s_k)$, $k \geq 1$ such that $(s_i, s_{i+1}) \in E$ for all $1 \leq i < k$. A path $(s_0 s_1 ... s_k)$ is a cycle if $s_k = s_1$, the cycle is accepting if it contains a state in F. A path $(s_0 s_1 ... s_k .... s_l)$ where $l > k$ is accepting if $s_k ... s_l$ forms an accepting cycle. We call a path that starts at the initial state and reaches an accepting cycle an accepting path or counterexample (see Figure 1). A minimal counterexample is an accepting path with a minimal number of transitions.

Strongly Connected Component. A graph is a pair $G = (V, E)$, where V is a set of states and $E \subseteq V \times V$ is the set of edges. A path is a sequence of states $(v_1, ..., v_k)$, $k \geq 1$ such that $(v_i, v_{i+1}) \in E$ for all $1 < i \leq k$. Let π be a path, the length of π is defined by the number of transitions and is denoted by $[\pi]$. We say that we can reach a vertex u from a vertex v if there exists a path from v to u. We define a Strongly Connected Component (SCC) as a maximal set of states $C \subseteq V$ such that for every pair of vertices $u, v \in C$, u and v are mutually reachable. A SCC C is trivial if $C = \{v\}$, or otherwise C is non-trivial if for every $u, v \in C$ there is a non-trivial path from u to v.

Discrete-Time Markov Chain (DTMC) A Discrete-Time Markov Chain (DTMC) is a tuple $D = (S, s_{init}, P, L)$, such that S is a finite set of states, $s_{init} \in S$ the initial state,$P : S \times S \to [0, 1]$ represents the transition probability matrix, $L : S \to 2^{AP}$ is a labelling function that assigns to each state $s \in S$ the set $L(s)$ of atomic propositions. An infinite path σ is a sequence of states $s_0 s_1 s_2 ...$, where $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. A finite path is the finite prefix of an infinite path. We define a set of paths starting from a state s_0 by $Paths(s_0)$. The probability of a finite path is calculated as follows:

$$P(\sigma \in Paths(s_0) | s_0 s_1 ... s_n \text{is a prefix of } \sigma) = \prod_{i \leq 0 < n} P(s_i, s_{i+1})$$

Linear Temporal Logic (LTL) The syntax of LTL state formula over the set AP is given as follows :

$$\varphi ::= true | a | \neg\varphi | \varphi_1 \wedge \varphi_2 | \bigcirc \varphi | \varphi_1 U \varphi_2$$

where $a \in AP$ is an atomic proposition. The Other Boolean connectives can be simply derived using the Boolean connectives $\neg$ and $\wedge$. The *eventual* operator F and the

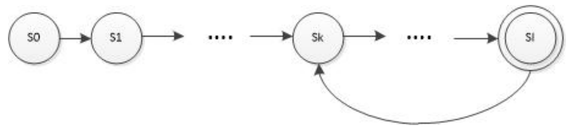


Figure 1: Accepting path (Counterexample).

always operator G can be easily derived using the temporal operator U.

Given a path $\pi = s_0 s_1 ...$ and an integer $j \geq 0$, where $\pi[j] = s_j$, such that $Words(\varphi) = \{\pi \in (2^{AP})^w)\sigma \models \varphi\}$, the semantics of LTL formulas for infinite words over 2^{AP} is given as follows:

$\pi \models true \Leftrightarrow true$
$\pi \models a \Leftrightarrow a \in L(s_0)$
$\pi \models \neg\varphi \Leftrightarrow \pi \not\models \varphi$
$\pi \models \varphi_1 \wedge \varphi_2 \Leftrightarrow s \models \varphi_1 \wedge s \models \varphi_2$
$\pi \models \bigcirc\varphi \Leftrightarrow \pi\,[1..] \models \varphi$
$\pi \models \varphi_1 \mathbf{U} \varphi_2 \Leftrightarrow \exists j \geq 0 . \pi\,[j..] \models \varphi_2 \wedge (\forall 0 \leq k < j . \pi\,[k..] \models \varphi_1)$

The semantics for the derived operators F and G is given as follows :

$\pi \models F\varphi \Leftrightarrow \exists j \geq 0 . \pi\,[j..] \models \varphi$
$\pi \models G\varphi \Leftrightarrow \forall j \geq 0 . \pi\,[j..] \models \varphi$

Verifying whether a finite state system described in Kripke structure A_M satisfies an LTL property φ reduces to the verification that $A = A_M \cap A_{\neg\varphi}$ has no accepting path, where $A_{\neg\varphi}$ refers to the Büchi automaton that violates φ, $L_\omega(A) = Words(\neg\varphi)$. We call this procedure a test of emptiness. So, in case $A_M \cap A_{\neg\varphi} \neq \emptyset$, a counterexample is generated.

Computation Tree Logic (CTL). We use the Computation Tree Logic (CTL) to specify properties of systems described using Kripke Structures. The CTL formulas are evaluated over infinite computations produced by Kripke structure K. A computation of a Kripke structure is an infinite sequence of states $s_0 s_1, ...$ such that $s_i, s_{i+1} \in R$ for all $i \in \mathbb{N}$. We denote by $Paths(s)$ the set of all paths starting at s. The syntax of CTL state formula over the set AP is given as follows:

$$\phi ::= true|a|\neg\phi|\phi_1 \wedge \phi_2|\exists\varphi|\forall\varphi$$

where $a \in AP$ is an atomic proposition and φ is a path formula. The path formulas are formed according to the following grammar:

$$\varphi ::= \bigcirc\phi|\phi_1 U \phi_2$$

We denote by $K, s \models \phi$ the satisfaction of CTL formula at a state s of K. The semantics defined by the satisfaction relation for a state formula is given as follows

$K, s \models true \Leftrightarrow true$
$K, s \models a \Leftrightarrow a \in L(s)$
$K, s \models \neg\phi \Leftrightarrow s \not\models \phi$
$K, s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \wedge s \models \phi_2$
$K, s \models \exists\varphi \Leftrightarrow$ for some $\pi \in Paths(s), \pi \models \varphi$
$K, s \models \forall\varphi \Leftrightarrow$ for all $\pi \in Paths(s), \pi \models \varphi$

Given a path $\pi = s_0 s_1 ...$ and an integer $i \geq 0$, where $\pi[i] = s_i$, the semantics of path formulas is given as follows:

$K, \pi \models \bigcirc\phi \Leftrightarrow \pi\,[1] \models \phi$
$K, \pi \models \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq 0 . \pi\,[j] \models \phi_2 \wedge (\forall 0 \leq k < j . \pi\,[k] \models \phi_1)$

In case the Kripke structure violates the specification $K \not\models \phi$, a counterexample will be generated.

Both LTL and CTL are considered as sub-logics or fragments of the logic CTL* [28, 52]. CTL is the subset of CTL* where each path operator $\bigcirc$ and U must be immediately preceded by path quantifiers $\forall$ or $\exists$, whereas LTL is the subset of CTL* that consists of formulas that have the form $\forall f$, where f is a path formula in which the only state formulas are just atomic propositions [32]. $ACTL$ is the analogue fragment of CTL and thus of CTL*, where the only quantifier allowed is $\forall$. Using CTL* we can express formulas of the form $\forall(FGp) \vee \forall G(\exists Fp)$, which is a disjunction of LTL and CTL formula.

Probabilistic Computation Tree Logic (PCTL). Probabilistic Computation Tree Logic (PCTL) [78] has appeared as an extension of CTL for the specification of systems that exhibit stochastic behavior. We use the PCTL to define quantitative properties of DTMCs. PCTL state formulas are formed according to the following grammar:

$$\phi ::= true|a|\neg\phi|\phi_1 \wedge \phi_2|\mathbf{P}_{\sim p}(\varphi)$$

Where $a \in AP$ is an atomic proposition, φ is a path formula, $\mathbf{P}$ is a probability threshold operator, $\sim \in \{<, \leq, >, \geq\}$ is a comparison operator, and p is a probability threshold. The path formulas φ are formed according to the following grammar:

$$\varphi ::= \phi_1 \mathbf{U} \phi_2 | \phi_1 \mathbf{W} \phi_2 | \phi_1 \mathbf{U}^{\leq n} \phi_2 | \phi_1 \mathbf{W}^{\leq n} \phi_2$$

Where ϕ_1 and ϕ_2 are state formulas and $n \in \mathbb{N}$. As in CTL, the temporal operators (**U** for strong until, **W** for weak (unless) until and their bounded variants) are required to be immediately preceded by the operator **P**. The PCTL formula is a state formula, where path formulas only occur inside the operator **P**. The operator **P** can be seen as a quantification operator for both the operators $\forall$ (universal quantification) and $\exists$ (existential quantification), since the properties are representing quantitative requirements.

The semantics of a PCTL formula over a state s (or a path σ) in a DTMC model $D = (S, s_{init}, P, L)$ can be defined by a satisfaction relation denoted by $\models$. The satisfaction of $\mathbf{P}_{\sim p}(\varphi)$ on DTMC depends on the probability mass of a set of paths satisfying φ. This set is considered as a countable union of cylinder sets, so that, its measurability is ensured.

The semantics of PCTL state formulas for DTMC is defined as follows:

$s \models true \Leftrightarrow true$
$s \models a \Leftrightarrow a \in L(s)$
$s \models \neg\phi \Leftrightarrow s \not\models \phi$
$s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \wedge s \models \phi_2$
$s \models \mathbf{P}_{\sim p}(\varphi) \Leftrightarrow P(s \models \varphi) \sim p$

Given a path $\sigma = s_0 s_1 ...$ in D and an integer $j \geq 0$, where $\sigma[j] = s_j$, the semantics of PCTL path formulas for DTMC is defined as for CTL as follows:

$\sigma \models \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq 0 . \sigma[j] \models \phi_2 \wedge (\forall 0 \leq k < j . \sigma[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{W} \phi_2 \Leftrightarrow \sigma \models \phi_1 \mathbf{U} \phi_2 \vee (\forall k \geq 0 . \sigma[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{U}^{\leq n} \phi_2 \Leftrightarrow \exists 0 \leq j \leq n . \sigma[j] \models \phi_2 \wedge (\forall 0 \leq k < j . \sigma[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{W}^{\leq n} \phi_2 \Leftrightarrow \sigma \models \phi_1 \mathbf{U}^{\leq n} \phi_2 \vee (\forall 0 \leq k \leq n . \sigma[k] \models \phi_1)$

For specifying properties of CTMC, we use the Continuous Stochastic Logic (CSL). CSL has the same syntax and semantics as PCTL, except that in CSL the time bound in bounded until formula can be presented as an interval of non-negative reals. Before verifying CSL properties over CTMC, the CTMC has to be transformed to its embedded DTMC. Therefore, further description of CTMC model checking is beyond the scope of this paper. We refer to [9, 10] for further details.

Generally, two types of properties can be expressed using temporal logics: *Safety* and *Liveness*. Safety proprieties state that something bad never happens, a simple example of that is the LTL formula $G\neg error$ that means that error never occurs. Liveness properties state that something good eventually happens, a simple example of that is the CTL formula ($\forall G req \rightarrow \forall F grant$) that means that every request is eventually granted.

3 Counterexamples generation

Counterexample generation has its origins in graph theory through the problem of cycle detection. Cycle detection is an important issue in the heart of model checking, either explicit or symbolic model checking. To deal with this issue, various algorithms were proposed for both LTL and CTL model checking. Explicit state model checking is based on Büchi automaton, which is a type of ω-automata. The fairness condition relies on several sets of accepting states, where the acceptance condition is visiting the acceptance set infinitely often. So, a run is accepting if only if it contains a state in every accepting set infinitely often. As a result, the emptiness of the language is based on checking the non-existence of the fair cycle or equivalently the fair non-trivial strongly connected component (SCC) that intersects each accepting set. In the case of non-emptiness, the accepting run is a sign of property failure, and as a result it is rendered as an error trace. We call this error trace a counterexample. So, the counterexample is typically presented by a finite stem followed by a finite cycle. Several algorithms were proposed to find counterexamples in reasonable time, where finding the shortest counterexample has been proved to be a NP-Complete problem [31, 82].

To find fair SCCs, Depth First Search (DFS) and Breadth First Search (BFS) algorithms are used. The main algorithm employing DFS is the Tarjan's algorithm [126] that is based on manipulating the states of the graph explicitly. This algorithm is used to generate linear counterexamples in LTL verification and showed promising results [43, 129]. It is also adopted in probabilistic model checking to generate probabilistic counterexamples for lower-bounded properties, through finding bottom strongly connected components (BSCCs)[5]. BSCC is defined as an SCC B from which no state outside B is reachable from B. Finding the set of BSCCs over the probabilistic models is an important issue for the verification of PCTL and CSL properties. Tarjan's algorithm runs in linear time, but as the number of states grows, it simply becomes infeasible. As a result, the symbolic-based algorithms are proposed as a solution.

In contrast to explicit algorithms, symbolic algorithms [17, 19] employ BFS and can describe large sets in a compact manner using characteristic functions. Several symbolic algorithms were proposed for computing the set of states that contains all the fair SCCs, without enumerating them [32, 84, 128]. We refer to these algorithms as SCC-hull algorithms. Currently, most of the symbolic model checkers are employing Emerson's algorithm due to its high performance, and it was proven by [58] that both of the algorithms [52] and [31] can work in a complementary way. Other works [83, 136] proposed algorithms based on enumerating the SCCs, we refer to these algorithms as symbolic SCC-enumeration algorithms.

Different approaches for generating counterexamples are proposed regarding the two types presented before. Clarke et al. [31] proposed a hull-based approach based on Emerson's algorithm by searching a cycle in a fair SCC close to the initial state. Another approach by Hojati [84] was also employed by other works for generating counterexamples that use isolations techniques of the SCCs [95]. Using Emerson's algorithm in a combinatory way with SCC-Enumeration algorithm is possible, but is still not guaranteed to get a counterexample of short length. Ravi et al. [111] introduced a careful analysis of each type of these algorithms. Since there is no guarantee to find terminal SCCs close to the initial state, finding short counterexamples was still a trade-off and an open problem, and thus it was investigated later by many researchers in both explicit and symbolic model checking.

3.1 Short counterexamples in explicit-state model checking

A counterexample in the Büchi automaton is a path $\sigma = \beta\gamma$ where β is a path without loop from the initial state to an accepting state, and γ is a loop around this accepting state. So that, a minimal counterexample is simply a counterexample with a minimal number of transitions. More for-

```
procedure DFS(s)
    Mark($\langle s, 0\rangle$)
    for each successor $t$ of $s$ do
        if $\langle t, 0\rangle$ not marked then
            DFS(t)
        end if
    end for
    if accepting(s) then $seed := s$; NDFS(s)
    end if
end procedure
procedure NDFS(s)
    Mark($\langle s, 1\rangle$)
    for each successor $t$ of $s$ do
        if $\langle t, 1\rangle$ not marked then
            NDFS(t)
        else
            if (t==seed) then report cycle
            end if
        end if
    end for
end procedure
```

Figure 2: Nested Depth First Search Algorithm[130].

mally, a counterexample $\sigma = \beta\gamma$ is minimal if $(|\beta|+|\gamma|) \leq (|\beta'|+|\gamma'|)$ for any path $\sigma' = \beta'\gamma'$. With respect to this definition, a counterexample has at least one transition. Many algorithms consider the issue of generating counterexamples given Büchi automaton [130, 85, 112]. All these works employ Nested-Depth First Search (NDFS), but they are not capable of finding a minimal counterexample. A basic NDFS algorithm proposed by [130] is depicted in Figure 2. The algorithm is based on computing the accepting states by performing a simple search, once an accepting state is found, another search is performed to find an accepting cycle through it.

Although minimal counterexamples can be computed in polynomial time using minimal paths algorithms, the main drawback, in fact, is the memory, where the resulting Büchi automaton to be checked for emptiness is usually very huge, the thing that makes storing all the minimal paths to be compared so difficult.

Recently, new methods were proposed to compute minimal counterexample in Büchi automaton [77, 64, 63]. Hansen and Kervinen [77] proposed a DFS algorithm that runs in $O(n^2)$ and they showed that $O(n \log n)$ is sufficient, although DFS algorithms are memory consuming in general. This is due to the optimizations added using interleaving. Since the algorithms are based on exploring transitions backwards, adapting this method in practice is very difficult, especially by considering some restrictions. While this method requires more memory than the model checker SPIN does, [64, 63] proposed a method that does not use more memory than SPIN does. While the first one uses DFS and its time complexity is exponential [64], Gastin and Moro proposed a BFS algorithm with some optimizations able of computing the minimal counterexample in polynomial time [63]. Hansen et al. [76] also proposed a method for computing minimal counterexamples based on Dijkstra algorithm for detecting strongly connected components. A novel approach was proposed by [93] for generating short counterexamples based on analyzing the entire model and defining which events have more contribution to the error, these events are called crucial. In addition to generating short counterexamples, the technique helps with reducing the state space. The main drawback of this method is how to determine if such set of events are crucial and really led to the error.

3.2 Short counterexamples in symbolic model checking

The original algorithm for counterexample generation in symbolic model checking was proposed by [31] and was implemented in most symbolic model checkers. This algorithm of generating linear counterexamples for the linear fragment of ACTL was later extended to handle arbitrary ACTL properties using the notion of tree-like counterexamples [36]. The authors realized that linear counterexamples are very weak for ACTL, and thus they proposed to generate tree-like Kripke structure instead, which is proven to be a viable counterexample[36, 38]. Formally, a tree-like counterexample is a a directed tree whose SCCs are either cycles or simple nodes. Figure 3 shows an example of a tree-like counterexample for the ACTL property $\forall G \neg a \vee \forall F \neg b$. As we see in the figure, the counterexample consists of two paths refuting both subformulas. The first path leads to a state that satisfies a, whereas the second path, which is expected to be an infinite one, along which b always holds. The generic algorithm for generating tree-like counterexamples as proposed in [36] is depicted in Figure 4.

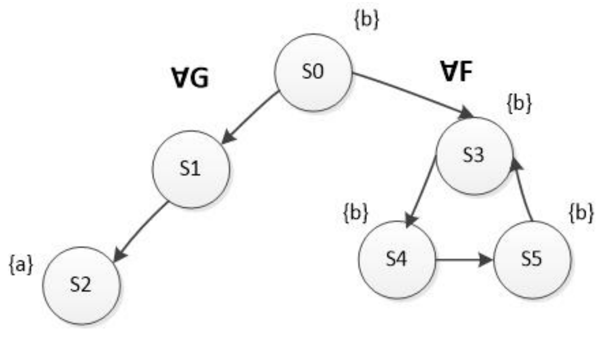


Figure 3: A tree-like counterexample for $\forall G \neg a \vee \forall F \neg b$.

The counterexample is constructed from an indexed Kripke structure K^ω that is obtained by creating isomorphic copy for each state in the original Kriple structure K, whereby no repeated state can be found. This process is called path *unraveling*. The algorithm traverses the specification formula in depth manner, where each subformula is evaluated recursively. The symbol O refers to temporal operator, and C is a global variable that is used in unraveling through denoting index of states.

The algorithm outputs a sequence of *descriptors* of the form $< s_0, .., s_n >$(path descriptor) and $< s_0, .., s_n, s_0 >$ (loop descriptor), where $\bigcup\{desc1, desc2\}$ describes a finite path leading to a cycle. The tree-like counterexample will be then $\bigcup\prod$, where $\prod$ refers to the set of descriptors generated by CEX algorithm. The set of descriptors for the example in Figure 3 would be: $< s_0, s_1, s_2 >$, $< s_0, s_3 >$ and $< s_3, s_4, s_5 >^{\omega}$.

```
1:  procedure CEX($K$, $s_0^i$, $\varphi$)
2:      case $\varphi$ of
3:      $\underline{\varphi_1 \vee \varphi_2}$:
4:      CEX(K, $s_0^i$, $\varphi_1$)
5:      CEX(K, $s_0^i$, $\varphi_2$)
6:      $\underline{\wedge_{i \geq 1}\varphi_i}$:
7:      $\underline{\varphi_1 \wedge \varphi_2}$:
8:      Select $j$ such that $K, s_0 \not\models \varphi j$
9:      CEX(K, $s_0^i$, $\varphi_j$)
10:     $\underline{\forall \mathbf{O}(\psi_1, ..., \psi_k)}$:
11:     Determine $\sigma = s_0, ..., s_N, ..., s_{N+M}$ such that
        $K, \sigma \not\models \mathbf{O}(\psi_1, ..., \psi_k)$
12:     desc1:= $\langle s_0^i, unravel(C, s_1, ..., s_N)\rangle$
13:     desc2:= $\langle unravel(C + N, s_N, ..., s_{N+M})\rangle^{\omega}$
14:     return desc1 and desc2
15:     $C := C + N + M + 1$
16:     for all states $p \in \bigcup\{desc1, desc2\}$ do
17:         for $j \in \{1, ..., k\}$ do
18:             if $K, p \not\models \psi_j$ then
19:                 CEX($K, p, \psi_j$)
20:             end if
21:         end for
22:     end for
23:     end case
24: end procedure
```

Figure 4: The generic counterexample algorithm for ACTL[36].

After these works of Clarke et al., many works have addressed the issue of computing short counterexamples in symbolic model checking [117, 29, 108]. Schuppan et al. [117] proposed some criteria that should be met for the Büchi automaton to accept shortest counterexamples. They proved that these criteria are satisfied in the approach proposed by [29] just for future time LTL specification, and thus they proposed an approach that meets the criteria proposed for LTL specifications with past. The algorithm proposed employs breadth-first reachability check with Binary Decision Diagrams(BDD)-based symbolic model checker.

The authors in [108] proposed a black-box based technique that masks some parts of the system in order to give an understandable counterexample to the designer. So the work does not just tend to produce minimal counterexamples, but also, it delivers small indicative counterexample of good quality to be analyzed in order to get the source of the error. The major drawback of this method is that the generalization of counterexample generation from symbolic model checking to black box model checking, could lead to non-uniform counterexamples that do not meet the behavior of the system intended. While all of these works are applied to unbounded model checking [117, 108], the works [122, 120, 113] consider bounded model checking, through lifting assignments produced by a SAT solver, where the quality of the counterexample generated depends on the SAT solver in use. Other works have investigated the use of heuristics algorithms for generating counterexamples [124, 50]. Although heuristics were not widely used, they gave pretty good results and were also used later for generating probabilistic counterexamples.

4 Counterexamples analysis and debugging

One of the major advantages of model checking over other formal methods is its ability to generate a counterexample when the model falsifies such specification. The counterexample represents an error trace; by analyzing it the user can locate the source of the error, and as Clarke wrote:"The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature" [27].

However, generating small and indicative counterexamples only is not enough for understanding the error. Therefore, counterexamples explanation is inevitable. Error explanation is the task of discovering why the system exhibits this error trace. Many works have addressed the automatic analysis of counterexamples to better understand the failure. Error explanation ranges in two main categories. The first is based on the error trace itself, through considering the small number of changes that have to be made in order to ensure that the given counterexample is no longer exhibited, and thus, these changes represent the sources of the error. The second is based on comparing successful executions with the erroneous one in order to find the differences, and thus those differences are considered as candidate causes for the error. Kumar et al. [97] have introduced a careful analysis of the complexity of each type. For the first type, they showed using three models (Mealy machines, extended finite state machines, and pushdown automaton) that this problem is NP-complete. For the second type, they provided a polynomial algorithm using Mealy machines and pushdown automaton, but solving the problem was difficult with extended finite state machines.

Error explanation methods are successfully integrated into model checkers such as SLAM [12] and Java PathFinder JP [16]. SLAM takes less execution time than JP, and can achieve completeness in finding the causes, but according to Groce [67], this also could be harmful. The error explanation process has many drawbacks; the main one is that the counterexample consists usually of a huge number of states and transitions and involves many variables. The second is that model checker usually floods the designer with multiple counterexamples, without any kind of classification. This makes challenging the task of choosing a helpful counterexample for debugging purposes. Besides, a single counterexample it might not be enough to understand the behavior of the system. Analyzing a set of counterexamples together is an option but the problem is that it requires much effort, and even though, the set of counterexamples to be analyzed could contain the same diagnostic information, which may make analyzing this set of counterexamples a waste of time. The last and the most important problem in error explanation is that not all the events that occur on the error trace are of importance for the designer, so locating critical events is the goal behind error explanation. In this section, we survey some works with respect to the two categories.

4.1 Computing the minimal number of changes

Jin et al. [92] proposed an algorithm for analyzing the counterexamples based on the local information, by segmenting the events of the counterexamples in two main segments, fated and free. The fated segments refer to the events that obviously have to occur in the executions, and the free segments refer to the events that should be avoided for the error not to occur, and thus they are candidate to be causes. Fated and free segments are computed with respect to input variables in the system, where they are classified into controlling and non-controlling. While controlling variables are considered to be critical, and have more control on the environment, the non-controlling variables have less importance. So that, fated segments are determined with respect to controlling variables, whereas free segments are determined with respect to non-controlling ones.

Wang et al. [132] also proposed a method that works just on the failed execution path without considering successful ones. The idea is about looking at the predicates candidate for causing the failure in the error trace. To do so, they use weakest pre-condition computation, the technique that is widely used in predicate abstraction. This computation aims to find the minimal number of conditions that should be met in order to not let the program violate the assertion. This results in a set of predicates that contradict with each other. By comparing how these predicates contradict to each other, we can find the cause for the assertion failed and map it back to the real code. Many similar works also provided error explanation methods in the context of C programs [137, 127, 138].

Using the notion of causality by Halpern and Pearl [74], Beer et al. [88] introduced a polynomial-time algorithm for explaining LTL counterexamples that was implemented as a feature in the IBM formal verification platform RuleBase PE. Given the error trace, the causes for the violation are highlighted visually as red dots on the error trace itself. The question asked was: what values of signals on the trace cause it to falsify the specification? Following the definition of Halpern and Pearl, they refer to such a set of pairs of state-variable as bottom-valued pairs whose values should be switched to make such state-variable pair critical. The pair is said to be critical if changing the value of the variable in this state no longer produces a counterexample. This pair represents the cause for the first failure of the LTL formula given the error trace, where they argue that the first failure is the most relevant to the user. Nevertheless, the algorithm computes an over-approximation of the set of causes not just the first cause that occurred.

Let φ be an LTL formula in Negation Normal Form (NNF) and $\pi = s0, s1, ..., sk$ a counterexample for it. The algorithm for computing the approximate set of causes given φ and π is depicted in Figure 5. The procedure invokes each time a function *val* for evaluating sub-formulas of φ on the path. The procedure is executed recursively given the formula φ until it reaches the proposition level, where the cause is finally rendered as a pair $\langle s_i, p\rangle$/$\langle s_i, \neg p\rangle$, where s_i refers to the current state.

Let us consider the formula : $G((\neg START \wedge \neg STATUS\_VALID \wedge END) \rightarrow [\neg START \mathbf{U} STATUS\_VALID])$. The result of executing the RuleBase PE implementation of the algorithm on this formula is shown in Figure 6. The red dots refer to the relevant causes for the error. Where some variables are not critical for the failure, others can be critical, which means that switching their values alone could result in mitigating the violation. For instance, in state 9, $START$ precedes $STATUS\_VALID$, by switching the value of $START$ from 1 to 0 in state 9, the formula would not fail anymore given this counterexample.

4.2 Comparing counterexamples with successful runs

This is the most adopted method for error explanation that is successfully featured in many model checkers such as SLAM and Java PathFinder. Groce et al. [70] have proposed an approach for counterexamples explanation based on computing a set of faulty runs called negatives, in which the counterexample is included, and comparing it to a set of correct runs called positives. Analyzing the common features and differences could lead to getting a useful diagnostic information. Their algorithms were implemented in JAVA pathfinder. Based on Lewis counterfactual theory of causality [105] and distance metrics, Groce [68] has proposed a semi-automated approach for isolating errors in ANSI C programs, by considering the alternative worlds as programs executions and the events as propositions about those executions. The approach relies on finding causal dependencies between predicates of a program. A predicate a is causally dependent on b given the faulty execution, if only if the executions in which the removal of a cause a also removes the effect b are more likely than the executions where a and b do not appear together. For finding these traces, which are as close as possible to the faulty one, the authors employed distance metric. A description of their

```
1:  procedure CAUSES($\varphi$, $\pi^i$)
2:      Case $\varphi$ of
3:      $p$:
4:      if $p \notin s_i$ then
5:          return $\langle s_i, p\rangle$
6:      end if
7:      $\neg p$:
8:      if $p \in s_i$ then
9:          return $\langle s_i, p\rangle$
10:     end if
11:     $\mathbf{X}\varphi$:
12:     if $i < k$ then return Causes($\varphi$, $\pi^{i+1}$)
13:     end if
14:     $\varphi \wedge \psi$:
15:     return Causes($\varphi$, $\pi^i$) $\cup$ Causes($\psi$, $\pi^i$)
16:     $\varphi \vee \psi$:
17:     if $val(\varphi, \pi^i) = 0$ and $val(\varphi, \psi^i) = 0$ then
18:         return Causes($\varphi$, $\pi^i$) $\cup$ Causes($\psi$, $\pi^i$)
19:     end if
20:     $\mathbf{G}\varphi$:
21:     if $val(\varphi, \pi^i) = 0$ then
22:         return Causes($\varphi$, $\pi^i$)
23:     else
24:         if $val(\varphi, \pi^i) = 1$ and $i < k$ and $val(\mathbf{XG}\varphi, \pi^i) = 1$ then
25:             return Causes($\mathbf{G}\varphi$, $\pi^{i+1}$)
26:         end if
27:     end if
28:     $\phi\mathbf{U}\psi$:
29:     if $val(\varphi, \pi^i) = 0$ and $val(\psi, \pi^i) = 0$ then
30:         return Causes($\varphi$, $\pi^i$) $\cup$ Causes($\psi$, $\pi^i$)
31:         if $val(\varphi, \pi^i) = 1$ and $val(\psi, \pi^i) = 0$ and $i = k$ then
32:             return Causes($\psi$, $\pi^i$)
33:         end if
34:         if $val(\varphi, \pi^i) = 1$ and $val(\psi, \pi^i) = 0$ and $i < k$ and $val(\mathbf{X}[\varphi\mathbf{U}\psi], \pi^i) = 0$ then
35:             return Causes($\psi$, $\pi^i$) $\cup$ Causes($\pi^{i+1}$, $[\varphi\mathbf{U}\psi]$)
36:         end if
37:     end if
38: end procedure
```

Figure 5: Causes generation algorithm given a counterexample[88].

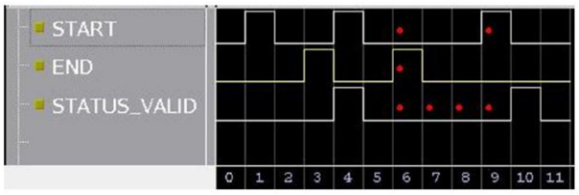


Figure 6: Explanations on counterexample as red dots[88].

approach is depicted in Figure 7.

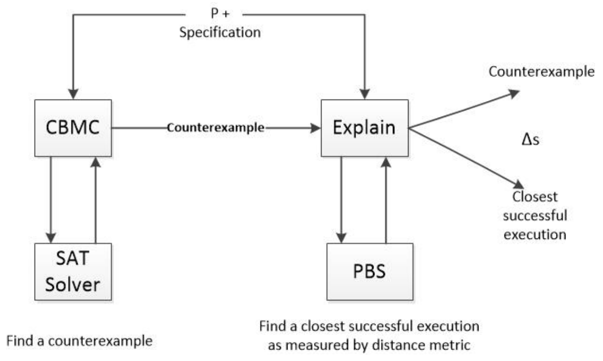
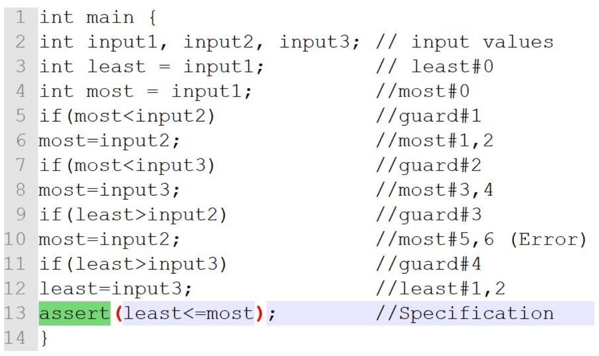

P +
Specification
CBMC
Counterexample
Explain
Counterexample
Δs
Closest successful execution
SAT Solver
PBS
Find a counterexample
Find a closest successful execution as measured by distance metric

Figure 7: Explanation using distance metric[68].

Given a program P and its specification, the model checker CBMC is used to generate a counterexample through using SAT solver, where the counterexample represents a finite execution of P. The explain tool[69] gets the counterexample generated from CBMC together with the P and its specification. It generates first a set of executions that do not violate the specification, and then using PBS solver [7], it tries to find the closest execution to the counterexample. Finally, the distance metric is computed, and a dynamic slicing technique is applied in order to point out to the most relevant assignments in the program that had contributed the most to the error.



```
int main {
int input1, input2, input3; // input values
int least = input1;         // least#0
int most = input1;          //most#0
if(most<input2)             //guard#1
most=input2;                //most#1,2
if(most<input3)             //guard#2
most=input3;                //most#3,4
if(least>input2)            //guard#3
most=input2;                //most#5,6 (Error)
if(least>input3)            //guard#4
least=input3;               //least#1,2
assert(least<=most);        //Specification
}
```

Figure 8: An example of C program.

We introduce here a brief explanation of this approach through a running example on a C code as indicated in [68]. The C program is depicted in Figure 8. For input values (1,0,1), a counterexample is rendered in a set of assignments form named Static Single Assignment(SSA),

```
input1#0=1          most#3=1
input2#0=0          most#4=1
input3#0=1          \guard#3=True
least#0=1           most#5=0
most#0=0            most#6=0
\guard#1=False      \guard#4=False
most#1=0            least#1=1
most#2=1            least#2=1
\guard#2=False
```

Figure 9: Counterexample values.

```
input1#0=1          most#3=1
input2#0=1          most#4=1
input3#0=1          \guard#3=False
least#0=1           most#5=1
most#0=1            most#6=1
\guard#1=False      \guard#4=False
most#1=1            least#1=1
most#2=1            least#2=1
\guard#2=False
```

Figure 10: Successful execution values.

which is a representation used by CBMC (See Figure 9). A successful execution so close to the counterexample can be found for the input values (1,1,1) (See Figure 10). The change in the value of $input2$ results in the assertion $least <= most$ being true. The differences between the two executions are presented in Figure 11. The first change is in the value of $input2$, which results in the change of $most\#1$ from 0 to 1. These two changes have of course lower importance to the following change, which concerns the non execution of $guard\#3$ that was executed in the counterexample, since $least\#0$ is no longer greater than $input2\#0$, and thus the value of $\#most6$ has changed to 1, which is considered the last change. The explanation that can be given for this counterexample, is that not executing the instruction at line 10 leads to the satisfaction of the assertion (no error occurring). This shows clearly the causal dependency of the satisfaction/violation of the assertion on executing this line of code. As a result, line 10 will be highlighted by *Explain* tool as an indication for the source of the error. The user will then understand that the this line of code should be corrected as $least = input2$.

In [22], Chaki and Groce extended the original approach for comparing a counterexample with the closest successful run through combining distance metric with predicate abstraction in order to generate explanations for abstract counterexamples. They argue that even for abstract counterexample, abstract state-space makes the explanation more informative. Renieris and Reiss [114] also introduced a method based on distance metric to select the closest correct runs to the faulty one and they provided a quantitative method for evaluating their methods.

```
Value changed:  input2#0 from 0 to 1
Value changed:  most#1 from 0 to 1
Guard changed:  least#0 > input2#0 (\guard#3) was True
Value changed:  most#5 from 0 to 1
Value changed:  most#6 from 0 to 1
```

Figure 11: Differences between the counterexample and the successful execution.

Ball et al. [11] proposed an effective approach that is currently featured in SLAM model checker. Their method is based on the same principle of finding successful runs to be compared with the counterexample. The interesting difference here is that it generates error trace per error cause, which makes the diagnostic easier, since there will not be causal dependencies in the traces generated. It is clear that this method will require the invocation of the model checker each time a cause for the error is found. Finally, the causes are reported as erroneous transitions that do not occur in any correct trace. Copty et al. [41] proposed a framework for debugging counterexamples as they refer to it as counterexample Wizard in the context of symbolic LTL. The technique employs three main capabilities: multi-value counterexample annotation, constraint-based debugging and multiple counterexample generation. But in contrast to the work by Ball et all, the model checker is not invoked each time an error cause is found, but instead, it gets all the data needed together to start the analysis.

Leue and Tabaei Befrouei [104, 103] proposed a novel approach based on computing two datasets, the *bad dataset* that represents the set of counterexamples, and the *good dataset* that represents the successful runs. Both datasets are produced using SPIN model checker. The idea is always about computing the differences between good and bad traces, but this time with the help of data mining technique called sequential pattern mining [49]. The aim behind using this technique is to extract a set of sequences of actions that are mostly to appear in the bad dataset. In concurrent systems, which are usually modeled using interleaving semantics, the unforeseen interleavings resulted from such a set of actions stand as a good indicator for the source of the error.

While all of the previous works addressed safety properties, Kumazawa and Tamai [98] attended to explain errors for liveness properties that involve more computational complexity. For that reason, the counterexample is represented as an infinite trace and not a finite one, and the witnesses to be compared with this counterexample are infinite as well. The method also employs shortest paths algorithms. Many similar works for counterexamples analysis have been done [121, 73, 40, 110, 119, 118, 56, 45].

5 Probabilistic counterexamples

Unlike the previous methods proposed for conventional model checking that generate the counterexample as a single path ending with a bad state representing the failure, the task in PMC is quite different. The counterexample in PMC is a set of evidences or diagnostic paths that satisfy path formula and their probability mass violates the probability threshold. The probabilistic counterexample is generated when a PCTL/CSL property is not satisfied. The probabilistic property $\phi = \mathbf{P}_{\leq p}(\varphi)$ is refuted when the probability mass of the paths satisfying φ exceeds the bound p. Therefore, a probabilistic counterexample for the property ϕ is formed by a set of paths starting at a state s and satisfying the path formula φ. We denote these paths by $Paths(s_0 \models \phi)$. The counterexample can be formed of a set of finite paths where each path $\sigma = s_0 s_1 ... s_n$ is a prefix of an infinite path from $Paths(s_0 \models \phi)$ satisfying the formula φ. We denote these paths by $FinitePaths(s_0 \models \phi)$.

We can get a set of probabilistic counterexamples, noted $PCX(s_0 \models \phi)$, which is a set of any combination from $FinitePaths(s_0 \models \phi)$ that their probability mass exceeds the bound p. Among all these probabilistic counterexamples, we are interested by the most indicative one. The most indicative counterexample is minimal counterexample (has the least number of paths from $FinitePaths(s_0 \models \phi)$) and its probability mass is the highest among all other minimal counterexamples. We denote the most indicative probabilistic counterexample by $MIPCX(s_0 \models \phi)$. We should note that the most indicative probabilistic counterexample may not be unique.

For the counterexample to have a high probability, it should consist of paths that carry high probabilities from $FinitePaths(s_0 \models \phi)$. The path σ having the highest probability over all these paths is called strongest path and is defined as follows: for every path $\sigma' \in FinitePaths(s_0 \models \phi)$: $P(\sigma) \geq P(\sigma')$. The strongest path also may not be unique.

Example Let us consider the example of DTMC shown in Figure 12 and the property $P_{\leq 0.5}(\varphi)$, where $\varphi = (a \vee b) \cup (c \wedge d)$. The property above is violated in this model ($s_0 \not\models \mathbf{P}_{\leq 0.5}(\varphi)$), since there exists a set of paths satisfying φ whose probability mass is higher than the probability bound (0.5). Any combination from $FinitePaths(s_0 \models \phi)$ having probability mass higher than 0.5, is a valid counterexample including the whole set. For instance, we can find three counterexamples:

$$P(CX_1) = P(\{s_0s_1, s_0s_2s_3, s_0s_2s_4s_3, s_0s_2s_4s_5, s_0s_4s_5\})$$

$$= 0.25 + 0.2 + 0.09 + 0.15 + 0.12 = 0.81$$

$$P(CX_2) = P(\{s_0s_1, s_0s_2s_4s_5, s_0s_4s_5\})$$

$$= 0.25 + 0.15 + 0.12 = 0.52$$

$$P(CX_3) = P(\{s_0s_1, s_0s_2s_3, s_0s_2s_4s_5\})$$

$$= 0.25 + 0.2 + 0.15 = 0.6$$

The last probabilistic counterexample is the most indicative since it is minimal and its probability is higher than

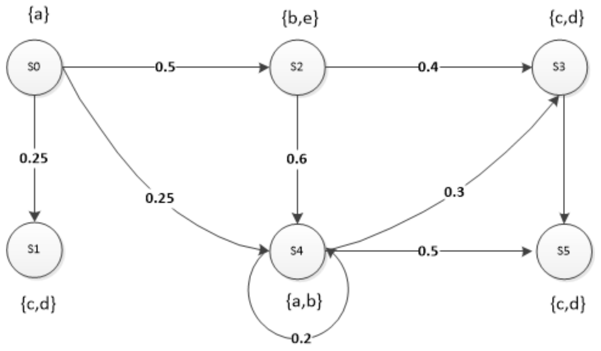


Figure 12: A DTMC.

the other minimal counterexample CX_2,$P(CX_3) = 0.6 > P(CX_2)$. The strongest path is s_0s_1, which is included in the most indicative probabilistic counterexample.

5.1 Probabilistic counterexample generation

Various approaches for probabilistic counterexamples generation have been proposed. Aljazzar et al. [1, 3] introduced an approach for counterexample generation for DTMC and CTMC against timed reachability proprieties using heuristics and directed explicit state space search. Since resolving nondeterminism in an MDP results in a DTMC, in complementary work [4], Aljazzar and Leue proposed an approach for counterexample generation for MDPs based on existing methods for DTMC. Aljazzar and Leue introduced a complete work in [5] for generating counterexamples for DTMCs and CTMSs as what they refer to as diagnostic sub-graphs. All these works on generating indicative counterexamples have led to the development of the K* algorithm [6], an on-the-fly heuristics guided algorithm for the K shortest path problem. Comparing to classical k-shortest-paths algorithms, K* has two main advantages, it woks on-the- fly in way it avoids exploring the entire graph, and it can be guided using heuristic functions. Based on all the previous works, they built a tool DiPro [2] for generating indicative counterexamples for DTMCs, CTMCs and MDPs. This tool can be jointly used with the model checkers PRISM [81] and MRMC [94], and can render the counterexamples in text format as well as in graphical mode. These heuristic-based algorithms showed a great efficiency in terms of counterexample quality. Nevertheless, with large models, DiPro tool that implements these algorithms takes usually a long time to produce a counterexample. By running DiPro on a PIRSM model of the DTMC presented in Figure 12 against the same property, we obtain the most indicative counterexample $CX3$. The graphical representation of $CX3$ as rendered by DiPro is depicted in Figure 13. The diamonds refer to the final or end states (s_1,s_3,s_5), whereas the circles represent simple nodes s_2 and s_4. The user can navigate through the counterexample and inspect all values.

Similar to the previous works, [75] has proposed the notion of smallest most indicative counterexample that redu-

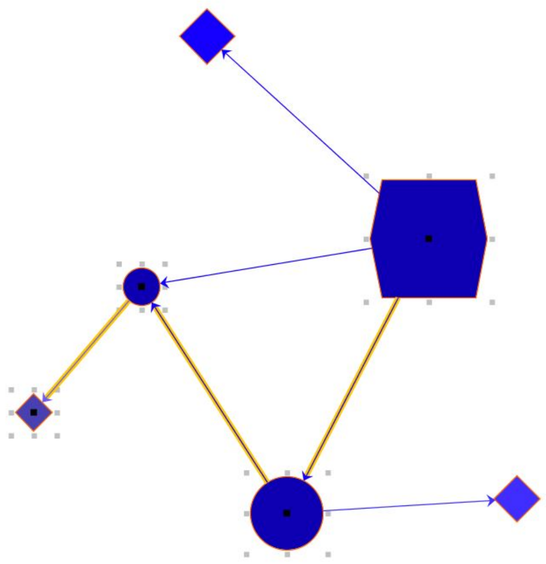


Figure 13: A counterexample generated by DiPro.

ces to the problem of finding K shortest paths. In a weighted digraph transformed from the DTMC model, and given initial state and the target states, the strongest evidences that form the counterexample are selected using extensions of K-shortest paths algorithms for an arbitrary number k. Instead of generating path-based counterexamples, [134] have proposed a novel approach for DTMCs and MDPs based on critical subsystems using SMT solvers and mixed integer linear programming. Critical subsystem is simply a part of the model (states and transitions) that are considered relevant because of its contribution to exceeding the probability bound. The problem has been shown that is NP-Complete. Another work always based on the notion of critical subsystem is proposed to deliver abstract counterexamples with less number of states and transitions using hierarchical refinement method. Based on all of these works, Jansen et al. proposed the COMICS tool for generating the critical subsystems that induce the counterexamples [90].

There are also many other works that addressed special cases for generating counterexamples in PMC. the authors of [8], proposed an approach for finding sets of evidences for bounded probabilistic LTL properties on MDP that behave differently from each other giving significant diagnostic information. While their method is also based on K-shortest path, the main contribution is about selecting the evidences or the witnesses with respect to main five criteria in addition to the high probability. While all of the previous works for counterexample generation are explicit-based, the authors in [133] proposed a symbolic method using bounded model checking. In contrast to the previous methods, this method lacks the selection of the strongest evidences first, since the selection is performed in arbitrary order. Another approach for counterexample generation that uses bounded model checking has been proposed [15]. Unlike the previous work that uses conventional SAT solvers, the authors used a SMT-solving approach in order to put some constraints on the paths selected, in order to get more abstract counterexample that consists of strongest paths. Counterexample generation for probabilistic LTL model checking has been addressed in [116] and probabilistic CEGAR has been also addressed [80]. A comprehensive representation of the counterexamples using regular expressions has been addressed in [44]. Since regular expressions deliver compact representations, they can help to deliver short counterexamples. Besides, they are widely known and easily understandable, so that they will give more benefits as a tool for error explanation.

5.2 Probabilistic counterexample analysis

Instead of relying on the state space search resulted from the parallel composition of the modules, [135] suggests to rely directly on the guarded command language used by the model checker, which is more likely and helpful for debugging purpose. To do so, the authors employ the critical subsystem technique [134] to identify the smallest set of guarded commands contributing to the error.

To analyze probabilistic counterexamples, Debbi and Bourahla [48, 47] proposed a diagnostic method based on the definition of causality by Halpern and Pearl [74] and responsibility [25]. The method proposed takes the probabilistic counterexample generated by DiPro tool and the probabilistic formula as input, and returns a set of pairs (state-variable) as candidate causes for the violation ordered with respect to their contribution to the error. So, in contrast to the previous methods, this method does not tend to generate indicative counterexamples, but it acts directly on indicative counterexamples already generated. Another similar approach for debugging probabilistic counterexamples has been introduced by [46]. It adopts the same definition of causality by Halpern and Pearl to reason formally about the causes, and then transforms the causality model into regression model using Structural Equation Modeling (SEM). SEM is a comprehensive analytical method used for testing and estimating causal relationships between variables embedded in theoretical causal model. This method helps to understand the behavior of the model through quantifying the causal effect of the variables on the violation, and the causal dependencies between them.

The same definition of causality has also been adopted to event orders for generating fault trees from probabilistic counterexamples, where the selection of traces forming the fault tree are restricted to some minimality condition [102]. To do so, Leitner-Fischer and Leue proposed the event order logic to reason about Boolean conditions on the occurrence of events, where the cause of the hazard in their context is presented as an Event Order Logic (EOL) formula, which is a conjunction of events. In [57], they extended their approach by integrating causality in explicit-state model checking algorithm to give a causal interpretation for sub- and super-sets of execution traces, the thing that could help the designer to get a better insight on the behavior of the system. They proved the applicability of their approach to many industrial size PROMELA models. They extended the causality checking approach to probabilistic counterexamples by computing the probabilities of events combination [101], but they still consider the use of causality checking of qualitative PROMELA models.

6 Counterexample guided abstraction refinement (CEGAR)

The main challenge in model checking is the state explosion problem. Dealing with this issue is in the heart of model checking, it was addressed at the beginning of model checking and not finished. Many methods were proposed to tackle this issue, the most famous are: symbolic algorithms, Partial Order Reduction (POR), Bounded Model Checking (BMC) and abstraction. Among these techniques, abstraction is considered as the most general and flexible for handling the state explosion problem [30]. Abstraction is about hiding or simplifying some details about the system to be verified, even removing some parts from it that are considered irrelevant for the property under consideration. The central idea is that verifying a simplified or an abstract model is more efficient than the entire model. Evidently, this abstraction has a price, which is losing some information, and the best abstraction methods are those that control this loss of information. Over-approximation and under-approximation are two main key concepts for this problem. Many abstraction methods have been proposed [42, 65, 106], the last one had the most attention and was adopted in the symbolic model checker NuSMV.

Abstraction can be defined by a set of abstract states $\widehat{S}$, an abstraction mapping function h that maps the states in the concrete model to $\widehat{S}$, and the set of atomic propositions AP labeling these states. Regarding the choice on $\widehat{S}$, h and AP, we distinguish three main types of abstraction : predicate abstraction [66, 115], localization reduction [99] and data abstraction [39]. Predicate abstraction is based on eliminating some variables from the program to be replaced by predicates that still serve the information about these variables. Each predicate has a Boolean variable corresponding to it, where the abstract states $\widehat{S}$ resulted are valuations of these variables. Both the abstraction mapping h between the concrete and abstract states, and the set of atomic propositions AP, are determined with respect to the truth values of these predicates. The entire abstract model can then be defined through existential abstraction. To this end, we can use BDDs, SAT solvers or theorem provers depending on the size of the program. Localization reduction and data abstraction are actually just extensions of predicate abstraction. Localization reduction aims to define a small set of variables that are considered relevant to the property in hand to be verified, these variables are called *visible*, the rest of variables that have no importance with respect to the property to be verified are called *invisible*. We should mention that this technique does not apply any abstraction on the domain of visible variables. Data abstraction deals mainly with the domains of variables by making an abstract domain for each variable. So the abstract model will be built with respect to the abstract values. For more detail on abstraction techniques, we refer to [71].

Given the possible loss of information caused by the abstraction, inventing some refinement methods of the abstract model is necessary. The most known method for abstraction refinement is Counterexample-Guided Abstraction Refinement (CEGAR) that has been proposed by [30] as a generalization of the localization reduction approach. A prototype implementation of this method in NuSMV has also been presented. In this approach, the counterexample plays the crucial role for finding the right abstract model. The process of CEGAR consists of three main steps: the first is to generate an abstract model using one of the abstractions techniques [30, 23, 33] given a formula φ. The second step is about checking the satisfaction of φ, if it is satisfied then the model checker stops and returns that the concrete or the original model satisfies the formula, if it is not satisfied, a counterexample will be generated. The counterexample generated is in the abstract model, so we have to check if it is also a valid counterexample in the concrete model, because the abstract model has different behavior comparing to the concrete one. Otherwise, the counterexample is called spurious and the abstraction must be carried out based on this counterexample. So, a spurious counterexample is an erroneous counterexample that exists only in the abstract model, not the concrete model. The final step is to refine the model until no spurious counterexample is found (see Figure 14). This is how the technique gets its name, refining the abstract model using the spurious counterexample. Refinement is an important task of CEGAR that can make the process faster and gives the appropriate results. To refine the abstract model, different partitioning algorithms are applied to abstract states. Like abstraction, partitioning the abstract states in order to eliminate the spurious counterexample can be carried out in many other ways than BDDs [30]. SAT solvers [24] or linear programming and machine learning [34] can be used to define the most relevant variables to be considered for the next abstraction.

In the literature, we find many extensions for CEGAR depending on the type of predicates and application domains: large program executions [96], non-Disjunctive abstractions [107] and propositional circumscription [89]. The CEGAR technique itself has been used to find bugs in complex and large systems [14]. The idea is based on gathering and saving information during the abstract model checking process in order to generate short counterexamples in the case of failure. This could be helpful for large models that make generating counterexamples using standard BMS intractable. CEGAR currently is implemented in many tools such as NuSMV[26], SLAM and BLAST[13].

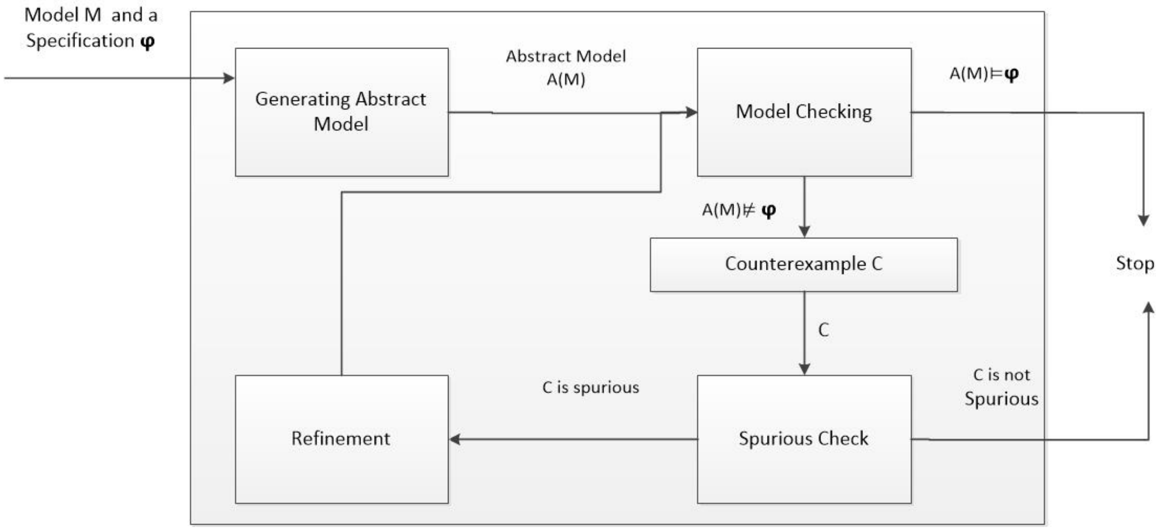


Figure 14: Counterexample Guided Abstraction Refinement Process.

7 Counterexamples for test cases generation

Counterexample generation gives the opportunity for model checking to be adopted and used in different domains, one of the domains in which the model checking has been adopted is test cases generation. Roughly speaking, testing is an automated method used to verify the quality of software. When we use model checking to generate test cases, this is called model-based testing. The use of model checking for testing is mainly subjected to the size of the software to be tested, because a suitable model must be guaranteed. The central idea of using model checking for testing [20, 55] is about interpreting counterexamples generated by the model checkers as test cases, and then test data and some expected results are extracted from these tests using such execution framework. Counterexamples are mainly used to help the designer to find the source of the error. However, they are very useful as test cases. [60].

A test describes the behavior of the test case intended: the final state, the states that should be traversed to reach the final state and so forth. In practice, it might not be possible to execute all test cases, since the software to be tested has usually a large number of behaviors. Nevertheless, there exist some techniques to help us to measure the reliability of testing. These techniques range in two main categories: first, deterministic methods (given initial state and some input, we will be certainty aware about the output), most famous methods for this category are coverage analysis and mutation analysis. Second, statistical analysis, where the reliability of the test is measured with respect to some probability distribution.

In coverage-based testing, the test purpose is specified in temporal logic and then converted to what is called a never-claim by negation; to assert that the test purpose never becomes true. So, the counterexample generated after the verification process will describe how the never-claim is violated, which is a description of how test purpose is fulfilled. Many approaches for creating never-claims based on coverage criterion (called "trap properties") [61] are proposed. Coverage criteria aim to find how such a system is exercised given a specification in order to get the states that were not traversed during the test; in this context, we call this specification a test suit. So, a full coverage is achieved if all the states of the system are covered. To create a test suite that covers all states, we need a trap property for each possible state. For example, claiming that the value of a variable is never 0: $G\neg(a = 0)$. A counterexample to such a trap property is any trace that reaches a state where $(a = 0)$.

With regard to *trap properties*, we find many variations. Gargantini and Heitmeyer addressed the coverage of software cost reduction (SCR) specifications [61]. SCR specifications are defined by tables over the events that represent the change of a value in state and lead to a new state, and conditions defined on the states. Formally, a SCR model is defined as quadruple (S, S_0, E^m, T) where S is the set of states, S_0 is the initial state set, E^m is the set of input events, and T is the transform function that maps an input event and the current state to a new one. SCR requirement properties can be used as never-claims, first by converting SCR into model checkers languages such as SPIN language (PROMELA), or SMV language, and then transform SCR tables into if-else construct in the case of using SPIN, or a case statement in the case of SMV. Another approach by Heimdahl et al. addressed the coverage of transition systems globally [79], where they consider the use of $RSML^{-e}$ as the specification language. A simple example of transition coverage criteria is of the form $G(A \wedge C \rightarrow \neg B)$, where A represents a system's state s, B represents the next state, and C is the condition that guards the transition A to B. So a counterexample for this property could be a trace that reaches a state B when C

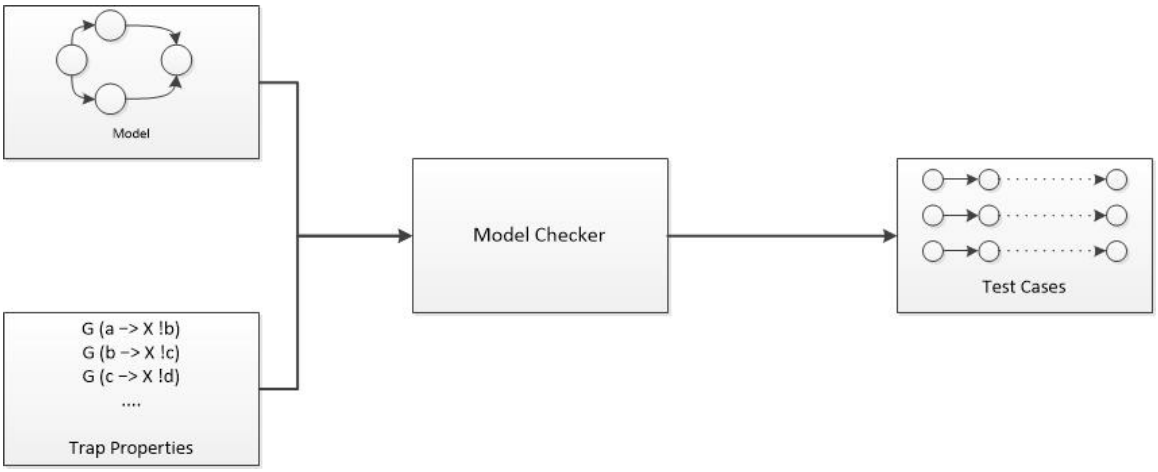


Figure 15: Coverage based test case generation [60].

evaluates to true, or a trace that reaches another state than B when C evaluates to false. Hong and Lee [87] proposed an approach based on control and data flow, where they use SMV model checker to generate counterexamples during the model checking of state-charts. The counterexample generated can be mapped into test sequence that induces information about which initial and stable states are considered. Another approach based on abstract state machines has been introduced [62]. The trap properties here will be defined over a set of rules for guarded function updates. We can see that all coverage-based approaches deal with the same thing, which is trap properties, and defer from each other in the formalism adopted.

Another approach for using requirement properties as test cases has been introduced by [54]. In this approach, each requirement has a set of tests. Trap properties can be easily derived from requirement properties under property-coverage criteria [125]. Another method that is completely different from coverage-based analysis is mutation-based analysis [18]. Mutation analysis consists of creating a set of mutants, which can be obtained by making small modifications on the original program in way these mutants lead to realistic faults. We differ between each mutant by its score, the mutant with the high score indicates high fault sensitivity. It is evident that deriving such mutant that is equivalent to the original program will have a high computational cost [91], because we have to apply all the test cases to each mutant, and all mutants should be considered. And for each mutant the model checker must be invoked. Fraser et al. [59] reviewed in detail most of these techniques and proposed several effective techniques to improve the quality of the test cases generated in model checking-based testing, especially requirements based testing, and apply them on different types of properties in many industrial case studies.

8 Counterexamples generation tools

Practically, all successful model checkers are able to output counterexamples in varying formats [38]. In this section, we will try to survey the tools supporting counterexample generation and study their effectiveness. A set of model checkers with their features are presented in Table 1.

Berkeley Lazy Abstraction Software Verification Tool (BLAST) [13] is a software model checking tool for C programs. BLAST has the ability to generate counterexamples, and furthermore, it employs CEGAR. BLAST is not just a CEGAR-based model checker, but it can be also used for generating test cases. BLAST shows promising results with safety properties of programs with a medium size.

CBMC [35] is a well-known Bounded Model Checker for AINCI C and C++ programs. CBMC performs symbolic execution on the programs and employs a SAT solver in the verification procedure, when the specification is falsified, a counterexample in the form of states with variables valuation leading to these states is rendered to the user.

JavaPathfinder(JPF) [131] is a famous software model checking tool for Java programs. JavaPathfinder is an effective virtual machine-based tool that verifies the program along all the possible executions. Due to its ability to deal with most of JAVA language features, because it runs on byte-code level, JavaPathfinder can generate a detailed report on the error in case that the property is violated. Besides, the tool gives the ability to generate test cases.

SPIN [86] is a model checker mostly known for the verification of systems that exhibit a high interaction between processes. The systems are described using Process Meta Language (PROMELA) language, and verified against properties specified in LTL. By applying a Depth-First Search algorithm on the intersection product of the model and the Büchi automaton representing the LTL formula, a counterexample is generated in case an accepted cycle is detected. SPIN offers an interactive simulator that helps to understand the cause of the failure by showing the processes and their interactions in order.

Table 1: Model checkers with their features.

Name	Model Checking				Counterexample generation	
	Programs, systems	Algorithms, methods	Modeling language	Specification language	Visualization	Form
BLAST	C programs	Predicate lazy abstraction, CEGAR	C	BLAST	No	Set of predicates
CBMC	C programs	BMC, SAT solving	C/C++	Assertions	No	Variables and valuations
JPF	Java programs	Explicit state, POR	Java	No	No	Error report
SPIN	Concurrent, distributed, asynchronous	Nested Depth First Search, POR	PROMELA	LTL	Yes	Execution path
NuSMV	Synchronous, asynchronous	BDD-based, SAT-based BMC	SMV	LTL, CTL	No	States and valuations
UPPAAL	Ral-time	On-the-fly, Symbolic	Timed automata	TCTL	Yes	Sequence of states
PRISM	Probabilistic, real-time	Graph-based, numerical	PRISM	PCTL,CSL, PTCTL	No-By DiPro	Graph
MRMC	Probabilistic	Numerical	PRISM, PEPA	PCTL,CSL, PRCTL,CSRL	No-By DiPro or COMIC	Graph

NuSMV [26] is a symbolic model checker that appeared as an extension of the Binary Decision Diagrams(BDD)-based model checker SMV. NuSMV includes both LTL and CTL for specification analysis, and combines SAT and BDD techniques for the verification. NuSMV can deliver a counterexample in XML format by indicating the states of the trace and the variables with their new values that cause the transitions.

UPPAAL [100] is a verification framework for real-time systems. The systems can be modeled as networks of timed automata extended with data types and synchronization channels, and the properties are specified using a Timed CTL(TCTL). UPPAAL can find and generate counterexamples in graphical mode as message sequence charts that indicate the events with respect to their order.

PRISM [81] is a probabilistic model checker used for the analysis of systems that exhibit stochastic behavior. The systems are described as DTMCs, CTMCs or MDPs, using guarded command language, and verified against probabilistic properties expressed in PCTL and CSL, and can be extended with rewards. Another successful probabilistic model checker extended with rewards is the Markov Reward Model Checker (MRMC) [94]. MRMC is mainly used for performance and dependability analysis. It takes the models as input files in two formats, in PRISM language or Performance Evaluation Process Algebra (PEPA). Although both model checkers have shown high effectiveness, they lack a mechanism for generating probabilistic counterexamples. Nevertheless, they have been used by recent tools (DiPro [2] and COMICS [90]) for generating and visualizing probabilistic counterexamples.

9 Conclusions and future directions

In this paper we surveyed counterexamples in model checking from different aspects. At the beginning of using model checking, counterexamples have not been treated as a particular subject, but they have been treated as a related problem to fair cycle detection algorithms, as presented in section 3. But recently, the quality of the counterexamples generated has been treated as a standalone and a fundamental problem. Many works tried to deliver short and indicative counterexamples to be used for debugging purpose. Concerning their structure, tree-like counterexamples have been proposed for the fragment of ACTL as an alternative for linear counterexamples, however, we see that this approach has not been adopted in model checkers, but instead model checkers are still based on generating simple non-branching counterexamples .

For debugging, we can conclude that approaches that require other successful runs might have some advantages over other methods based on single trace, in way that they compare many good traces to restrict the set of candidate causes. However, these methods take usually much execution time in order to select the appropriate set of traces, and besides, such traces could contain the same diagnostic information. Regardless of the debugging method in use, the challenge of visualizing the error traces and the quality of diagnoses generated to facilitate debugging is still an open issue.

For the case of counterexample generation in PMC, we have seen that the principle of counterexample generation is completely different than conventional model checking, where the presentation of counterexample is different from a work to another, from smallest and indicative set of paths to most critical sub-systems. Despite the notable advancement in generating probabilistic counterexamples that led to inventing important tools like DiPro and COMICS, unfortunately this advancement is still insufficient for debugging. Actually, it is more than important to see the techniques for counterexample generation and counterexample analysis integrated in probabilistic model checkers to get their benefit. All these techniques act on verification results of probabilistic model checkers like PRISM, so making the approaches of counterexample generation and counterexamples analysis to be performed during the model checking process itself is still an open problem. This could really have a great impact on probabilistic model checking.

We have also seen the usefulness of counterexamples for other purposes than debugging, like CEGAR and test cases generation. For CEGAR, we have seen different approaches for both abstraction and refinement. We have seen that we can benefit from using SAT solvers and theorem provers on the both sides, abstraction and refinement, thus they are very useful for CEGAR. Fast elimination of spurious counterexamples is still an active research topic. We also expect to see more works on CEGAR in PMC.

For testing, we have seen that the most useful approaches using model checking are those based on coverage and trap properties. Other approaches for testing like requirement-based analysis and mutation-based analysis have received smaller attention due to the limitations presented. Currently, coverage-based techniques are widely used in the industry. In the future, we expect to see the proposition of new approaches to enable us to test new emerging systems, which require new transformation mechanisms for enabling trap properties to be verified by model checkers to generate the counterexamples.

We should mention that such techniques can benefit from other techniques. For instance, new efficient CEGAR techniques will not only have an impact on conventional model checking, but on probabilistic model checking as well. We can also see in the future the use of probabilistic model checkers like PRISM for testing probabilistic systems. Since PRISM does not generate counterexamples, any advancement in generating indicative counterexamples could be of benefit for testing probabilistic systems. We can also see that techniques based on counterexamples like CEGAR can directly benefit from any advancement in generating small and indicative counterexamples in a considerable time.

In addition to all of this, we expect to see more works in other domains that adapt model checking techniques just for the seek of getting counterexamples. In previous works we have seen for instance that counterexamples can be mapped to UML sequence diagrams, describing states and events in the original model [51], they can be used to generate attack graphs in networks security [123], in fragmentation of services in Service-Based Applications (SBAs) [21], and they have been also used to enforce synchronizability and realizability in distributed services integration [72].

References

[1] Aljazzar, H., Hermanns, H., and Leue, S. Counterexamples for timed probabilistic reachability. In *FORMATS* (2005), LNCS, vol. 3829, Springer, Berlin, Heidelberg, pp. 177–195.

[2] Aljazzar, H., Leitner-Fischer, F., Leue, S., and Simeonov, D. Dipro - a tool for probabilistic counterexample generation. In *18th International SPIN Workshop* (2011), LNCS, vol. 6823, Springer, Berlin, Heidelberg, pp. 183–187.

[3] Aljazzar, H., and Leue, S. Extended directed search for probabilistic timed reachability. In *FORMATS* (2006), LNCS, vol. 4202, Springer, Berlin, Heidelberg, pp. 33–51.

[4] Aljazzar, H., and Leue, S. Generation of counterexamples for model checking of markov decision processes. In *International Conference on Quantitative Evaluation of Systems (QEST)* (2009), pp. 197–206.

[5] Aljazzar, H., and Leue, S. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. on Software Engineering 36*, 1 (2010), 37–60.

[6] Aljazzar, H., and Leue, S. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence 175*, 18 (2011), 2129 – 2154.

[7] Aloul, F., Ramani, A., Markov, I., and Sakallah, K. Pbs: A backtrack search pseudo boolean solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)* (2002), pp. 346–353.

[8] Andres, M. E., DArgenio, P., and van Rossum, P. Significant diagnostic counterexamples in probabilistic model checking. In *Haifa Verification Conference* (2008), pp. 129–148.

[9] Aziz, A., Sanwal, K., Singhal, V., and Brayton, R. Model-checking continuous-time markov chains. *ACM Transactions on Computational Logic 1*, 1 (2000), 162–170.

[10] Baier, C., Haverkort, B., Hermanns, H., and Katoen, J.-P. Model checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering 29*, 7 (2003), 524–541.

[11] Ball, T., Naik, M., and Rajamani, S. From symptom to cause: Localizing errors in counterexample traces. In *ACM Symposium on the Principles of Programming Languages* (2003), pp. 97–105.

[12] Ball, T., and Rajamani, S. The slam project: Debugging system software via static analysis. In *ACM Symposium on the Principles of Programming Languages* (2002), pp. 1–3.

[13] Beyer, D., Henzinger, T., Jhala, R., and Majumdar, R. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT) 9*, 5 (2007), 505–525.

[14] Bjesse, P., and Kukula, J. Using counterexample guided abstraction refinement to find complex bugs. In *Design, Automation and Test in European Conference and Exhibition* (2004), pp. 156–161.

[15] Braitling, B., and Wimmer, R. Counterexample generation for markov chains using smt-based bounded model checking. In *Formal Techniques for Distributed Systems* (2011), LNCS, vol. 6722, Springer, Berlin, Heidelberg, pp. 75–89.

[16] Brat, G., Havelund, K., Park, S., and Visser, W. Java pathfinder a second generation of a java model checker. In *Workshop on Advances in Verification* (2000).

[17] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput 35*, 8 (1986), 677–691.

[18] Budd, T., and Gopal, A. Program testing by specification mutation. *Journal Computer Languages 10*, 1 (1985), 63–73.

[19] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. Symbolic model checking: 1020 states and beyond. *Information and Computation 98*, 2 (1992), 142–170.

[20] Callahan, J., Schneider, F., and Easterbrook, S. Automated software testing using model checking. In *SPIN Workshop* (1996).

[21] Chabane, Y., Hantry, F., and Hacid, M. Querying and splitting techniques for sba: A model checking based approach. In *Emerging Intelligent Technologies in Industry* (2011), SCI 369, Springer-Verlag, Berlin, Heidelberg, pp. 105–122.

[22] Chaki, S., and Groce, A. Explaining abstract counterexamples. In *SIGSOFT04/FSE* (2004), pp. 73–82.

[23] Chauhan, P., Clarke, E., Kukula, J., Sapra, S., Veith, H., and D.Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer Aided Design(FMCAD)* (2002), LNCS, vol. 2517, Springer, Berlin, Heidelberg, pp. 33–51.

[24] Chauhan, P., Clarke, E., Kukula, J., Sapra, S., Veith, H., and D.Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *FMCAD 2002* (2002), LNCS, vol. 2517, Springer-Verlag, Berlin, Heidelberg, pp. 33–51.

[25] Chockler, H., and Halpern, J. Y. Responsibility and blame: a structural model approach. *Journal of Artificial Intelligence Research (JAIR) 22*, 1 (2004), 93–115.

[26] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri., M. Nusmv: a new symbolic model verifier. In *Proceedings Eleventh Conference on Computer-Aided Verification (CAV 99)* (1999), LNCS, vol. 1633, Springer, Berlin, Heidelberg, pp. 495–499.

[27] Clarke, E. The birth of model checking. In *Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking* (2008), LNCS, Springer, Berlin, Heidelberg, pp. 1–26.

[28] Clarke, E., and Emerson, A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs* (1982), Springer-Verlag, pp. 52–71.

[29] Clarke, E., Grumberg, O., and Hamaguchi, K. Another look at ltl model checking. *Formal Methods in System Design 10*, 1 (1997), 47–71.

[30] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM) 50*, 5 (2003), 752–794.

[31] Clarke, E., Grumberg, O., McMillan, K., and Zhao, X. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. of the Design Automation Conference* (1995), ACM Press, pp. 427–432.

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

[33] Clarke, E., Gupta, A., Kukula, J., and Strichman, O. Sat based abstraction refinement using ilp and machine leraning techniques. In *Computer-Aided Verification (CAV)* (2002), LNCS, vol. 2404, Springer, Berlin, Heidelberg, pp. 137–150.

[34] Clarke, E., Gupta, A., Kukula, J., and Strichman, O. Sat based abstraction refinement using ilp and machine leraning techniques. In *CAV 2002* (2002), LNCS, vol. 2404, Springer-Verlag, Berlin, Heidelberg, pp. 265–279.

[35] Clarke, E., Kroening, D., and Lerda, F. A tool for checking ansi-c programs. In *TACAS 2004* (2004), LNCS, vol. 2988, Springer, Berlin, Heidelberg, pp. 168–176.

[36] Clarke, E., Lu, Y., s. Jha, and Veith, H. Tree-like counterexamples in model checking. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science* (2002), pp. 19–29.

[37] Clarke, E., O.Grumberg, Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement. In *CAV* (1986), pp. 154–169.

[38] Clarke, E., and Veith, H. Counterexamples revisited: Principles, algorithms and applications. In *In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking* (2008), LNCS, Springer, Berlin, Heidelberg, pp. 1–26.

[39] Clarke, E. M., Grumberg, O., and Andlong, D. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16*, 5 (1994), 1512–1542.

[40] Cleve, H., and Zeller, A. Locating causes of program failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2005), pp. 342–351.

[41] Copty, F., Irron, A., Weissberg, O., Kropp, N., and Gila, K. Effcient debugging in a formal verification environment. *Int J Softw Tools Technol Transfer 4* (2003), 335–348.

[42] COUSOT, P., and COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium of Programming Language* (2003), pp. 238–252.

[43] Couvreur, J. On-the-fly verification of linear temporal logic. In *FM* (1999), LNCS, vol. 1708, Springer, Heidelberg, pp. 253–271.

[44] Damman, B., Han, T., and Katoen, J. Regular expressions for pctl counterexamples. In *Quantitative Evaluation of Systems(QEST)* (2008), pp. 179–188.

[45] de Alfaro, L., Henzinger, T., and Mang, F. Detecting errors before reaching them. In *CAV* (2000), LNCS, vol. 2725, Springer, Berlin, Heidelberg, pp. 186–201.

[46] Debbi, H. Diagnosis of probabilistic models using causality and regression. In *in Proceedings of the 8th International Workshop on Verification and Evaluation of Computer and Communication Systems* (2014), pp. 33–44.

[47] Debbi, H. *Systems Analysis using Model Checking with Causality*. PhD thesis, University of M'sila, 2015.

[48] Debbi, H., and Bourahla, M. Causal analysis of probabilistic counterexamples. In *Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign (Memocode)* (2008), pp. 77–86.

[49] Dong, G., and Pei, J. *Sequence Data Mining*. Springer, 2007.

[50] Edelkamp, S., Leue, S., and Lluch-Lafuente, A. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer 5*, 2 (2004), 247–267.

[51] Elamkulam, J., Z. Glazberg, I. R., Kowlali, G., Chandra, S., Kohli, S., and Dattathrani, S. Detecting design flaws in uml state charts for embedded software. In *HVC 2006* (2006), LNCS, vol. 4383, Springer-Verlag, Berlin, Heidelberg, pp. 109–121.

[52] Emerson, E., and Halpern, J. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC 82: Proceedings of the fourteenth annual ACM symposium on Theory of computing* (1982), ACM Press, pp. 169–180.

[53] Emerson, E. A., and Lei, C.-L. Efficient model checking in fragments of the propositional mucalculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science* (1986), pp. 267–278.

[54] Engels, A., Feijs, L., and Mauw, S. Test generation for intelligent networks using model checking. In *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS97)* (1997), LNCS, vol. 1217, Springer, Berlin, Heidelberg, pp. 384–398.

[55] Engels, A., Feijs, L., and Mauw, S. Test generation for intelligent networks using model checking. In *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems(TACAS)* (2010), LNCS, vol. 1217, Springer, Berlin, Heidelberg, pp. 384–398.

[56] Fey, G., and Drechsler, R. Finding good counterexamples to aid design verification. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE03)* (2003), pp. 51–52.

[57] Fischer, F., and Leue, S. Causality checking for complex system models. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2013), LNCS, vol. 7737, Springer, Berlin, Heidelberg, pp. 248–276.

[58] Fisler, K., Fraer, R., Kamhi, G., Vardi, M., and Yang, Z. Is there a best symbolic cycle-detection algorithm. In *TACAS 2001* (2001), LNCS, vol. 2031, Springer, Berlin, Heidelberg, pp. 420–434.

[59] Fraser, G. *Automated Software Testing with Model Checkers*. PhD thesis, IST - Institute for Software Technology, 2007.

[60] Fraser, G., Wotawa, F., and Ammann, P. E. Testing with model checkers. *Journal of Software Testing, Verification and Reliability 19*, 3 (2009), 215–261.

[61] Gargantini, A., and Heitmeyer, C. Using model checking to generate tests from requirements specifications. In *ESEC/FSE99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (1999), LNCS, vol. 1687, Springer, Berlin, Heidelberg, pp. 146–162.

[62] Gargantini, A., Riccobene, E., and Rinzivillo, S. Using spin to generate tests from asm specifications. In *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM* (2003), LNCS, vol. 2589, Springer, Berlin, Heidelberg, pp. 263–277.

[63] Gastin, P., and Moro, P. Minimal counterexample generation for spin. In *14th International SPIN Workshop 2007* (2007), LNCS, vol. 4595, Springer, Berlin, Heidelberg, pp. 24–38.

[64] Gastin, P., Moro, P., and Zeitoun, M. Minimization of counterexample in spin. In *SPIN 2004* (2004), LNCS, vol. 2989, Springer, Berlin, Heidelberg, pp. 92–108.

[65] GRAF, S., and ANDSADI, H. Construction of abstract state graphs with pvs. In *CAV* (1997), LNCS, vol. 1254, Springer, Berlin, Heidelberg, pp. 72–83.

[66] Graf, S., and Saidi, H. Construction of abstract state graphs with pvs. In *CAV 97* (1997), LNCS, vol. 1254, Springer-Verlag, Berlin, Heidelberg, pp. 72–83.

[67] Groce, A. *Error Explanation and Fault Localization with Distance Metrics*. PhD thesis, School of Computer Science Carnegie Mellon University, 2005.

[68] Groce, A., Chaki, S., Kroening, D., and Strichman, O. Error explanation with distance metrics. *International Journal on Software Tools for Technology 4*, 3 (2006), 229–247.

[69] Groce, A., Kroening, D., and Lerda, F. Understanding counterexamples with explain. In *Alur R., Peled D.A. (eds) Computer Aided Verification. CAV 2004* (2004), vol. 3114 of *Lecture Notes in Computer Science*, Springer, pp. 453–456.

[70] Groce, A., and Visser, W. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software* (2003), pp. 121–135.

[71] Grumberg, O. Abstraction and refinement in model checking. In *FMCO 2005* (2006), LNCS, vol. 4111, Springer-Verlag, Berlin, Heidelberg, pp. 219–242.

[72] Gudemann, M., Salaun, G., and Ouederni, M. Counterexample guided synthesis of monitors for realizability enforcement. In *ATVA 2012* (2012), LNCS, vol. 7561, Springer-Verlag, Berlin, Heidelberg, pp. 238–253.

[73] Guo, L., Roychoudhury, A., and Wang, T. Accurately choosing execution runs for software fault localization. In *15th international conference on Compiler Construction* (2006), LNCS, vol. 3923, Springer, Berlin, Heidelberg, pp. 80–95.

[74] Halpern, J., and Pearl, J. Causes and explanations: A structural-model approach part i: Causes. In *17th UAI* (2001), pp. 194–202.

[75] Han, T., and Katoen, J. Counterexamples generation in probabilistic model checking. *IEEE Trans. on Software Engineering 35*, 2 (2009), 72–86.

[76] Hansen, H., and Geldenhuys, J. Cheap and small counterexamples. In *Software Engineering and Formal Methods, SEFM '08* (2008), IEEE Computer Society Press, pp. 53–62.

[77] Hansen, H., and Kervinen, A. Minimal counterexamples in o(n log n) memory and o(n 2) time. In *ACDC 2006* (2006), IEEE Computer Society Press, pp. 131–142.

[78] Hansson, H., and Jonsson, B. logic for reasoning about time and reliability. *Formal aspects of Computing 6*, 5 (1994), 512–535.

[79] Heimdahl, M., Rayadurgam, S., and Visser, W. Specification centered testing. In *Second International Workshop on Automates Program Analysis, Testing and Verification* (2000).

[80] Hermanns, H., Wachter, B., and Zhang, L. Probabilistic cegar. In *Computer Aided Verification (CAV)* (2008), LNCS, vol. 5123, Springer, Berlin, Heidelberg, pp. 162–175.

[81] Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. Prism: A tool for automatic verification of probabilistic systems. In *TACAS* (2006), LNCS, vol. 3920, Springer, Berlin, Heidelberg, pp. 441–444.

[82] Hojati, R., Brayton, R. K., and Kurshan, R. P. Bdd-based debugging of designs using language containment and fair ctl. In *Fifth Conference on Computer Aided Verification (CAV 93)* (1993), LNCS, vol. 697, Springer, Berlin, Heidelberg, pp. 41–58.

[83] Hojati, R., Brayton, R. K., and Kurshan, R. P. Bdd-based debugging of designs using language containment and fair ctl. In *CAV 93* (1993), LNCS, vol. 697, Springer, Berlin, Heidelberg, pp. 41–58.

[84] Hojati, R., Touati, H., Kurshan, R. P., and Brayton, R. K. Effcient -regular language containment. In *Computer Aided Verification* (1992), LNCS, vol. 1708, Springer, Berlin, Heidelberg, pp. 371–382.

[85] Holzmann, G., Peled, D., and Yannakakis, M. On nested depth

first search. In *SPIN'96* (1996).

[86] Holzmann, G. J. The model checker spin. *IEEE Transactions on Software Engineering 23*, 5 (1997), 1–17.

[87] Hong, H. S., and Lee, I. Automatic test generation from specifications for controlflow and data-flow coverage criteria. In *International Conference on Software Engineering (ICSE)* (2003).

[88] I.Beer, Ben-David, S., Chockler, H., Orni, A., and Treer, R. Explaining counterexamples using causality. *Formal Methods Systems Design 40*, 1 (2012), 20–40.

[89] Janota, M., Grigore, R., and Marques-Silva, J. Counterexample guided abstraction refinement algorithm for propositional circumscription. In *JELIA'10 Proceedings of the 12th European conference on Logics in artificial intelligence* (2010), LNCS, vol. 6341, Springer, Berlin, Heidelberg, pp. 195–207.

[90] Jansen, N., Abraham, E., Volk, M., Wilmer, R., Katoen, J., and Becker, B. The comics tool - computing minimal counterexamples for dtmcs. In *ATVA* (2012), LNCS, vol. 7561, Springer, Berlin, Heidelberg, pp. 249–253.

[91] Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. *IEEE Transactions ON Software Engineering 37*, 05 (2011), 649 – 678.

[92] Jin, H., Ravi, K., and F.Somenzi. Fate and free will in error traces. *International Journal on Software Tools for Technology Transfer 6*, 2 (2004), 102–116.

[93] Kashyap, S., and Garg, V. Producing short counterexamples using crucial events. In *CAV 2008* (2008), LNCS, vol. 5123, Springer, Berlin, Heidelberg, pp. 491–503.

[94] Katoen, J.-P., Khattri, M., and Zapreev, I. S. A markov reward model checker. In *QEST* (2005), pp. 243–244.

[95] Kesten, Y., Pnueli, A., and o. Raviv, L. Algorithmic verification of linear temporal logic specifications. In *International Colloquium on Automata, Languages, and Programming (ICALP-98),* (1998), LNCS, vol. 1443, Springer, Berlin, Heidelberg, pp. 1–16.

[96] Kroening, D., Groce, A., and Clarke, E. Counterexample guided abstraction refinement via program execution. In *6th International Conference on Formal Engineering Methods (ICFEM)* (2004), LNCS, vol. 3308, Springer, Berlin, Heidelberg, pp. 224–238.

[97] Kuma, N., Kumar, V., and Viswanathan, M. On the complexity of error explanation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)* (2005), LNCS, vol. 3385, Springer, Berlin, Heidelberg, pp. 448–464.

[98] Kumazawa, T., and Tamai, T. Counterexample-based error localization of behavior models. In *NASA Formal Methods* (2011), pp. 222–236.

[99] Kurshan, R. P. *Computer-Aided Verification of coordinating processes - the automata theoretic approach.* Princeton University Press, 1994.

[100] Larsen, K. G., Pettersson, P., and Wang, Y. Uppaal in a nutshell. *Int. J. Software Tools for Technology Transfer 1*, 1 (1997), 134–152.

[101] Leitner-Fischer, F., and Leue, S. On the synergy of probabilistic causality computation and causality checking. In *SPIN 2013* (2013), LNCS, vol. 7976, Springer-Verlag, Berlin, Heidelberg, pp. 246–263.

[102] Leitner-Fischer, F., and Leue, S. Probabilistic fault tree synthesis using causality computation. *IJCCBS 4*, 2 (2013), 119–143.

[103] Leue, S., and Befrouei, M. T. Counterexample explanation by anomaly detection. In *SPIN* (2012), vol. 7385 of *Lecture Notes in Computer Science*, Springer, pp. 24–42.

[104] Leue, S., and Befrouei, M. T. Mining sequential patterns to explain concurrent counterexamples. In *SPIN* (2013), vol. 7976 of *Lecture Notes in Computer Science*, Springer, pp. 264–281.

[105] Lewis, D. Causation. *Journal of Philosophy 70* (1973), 556–567.

[106] LONG, D. *Model checking, abstraction and compositional verification.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2005.

[107] McMillan, K., and Zuck, L. Abstract counterexamples for non-disjunctive abstractions. In *Reachability Problems* (2009), LNCS, vol. 5797, Springer, Berlin, Heidelberg, pp. 176–188.

[108] Nopper, T., Scholl, C., and Becker., B. Computation of minimal counterexamples by using black box techniques and symbolic methods. In *Computer-Aided Design (ICCAD)* (2007), IEEE Computer Society Press, pp. 273–280.

[109] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science* (1977), IEEE, pp. 46–57.

[110] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. P. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging* (2003), pp. 273–276.

[111] Ravi, K., Bloem, R., and Somenzi, F. A comparative study of symbolic algorithms for the computation of fair cycles. In *Third International Conference, FMCAD 2000* (2000), LNCS, vol. 1954, Springer, Berlin, Heidelberg, pp. 162–179.

[112] Ravi, K., Bloem, R., and Somenzi, F. A note on on-the-fly verification algorithms. In *TACAS 2005* (2005), LNCS, vol. 3440, Springer, Berlin, Heidelberg, pp. 174–190.

[113] Ravi, K., and Somenzi, F. Minimal assignments for bounded model checking. In *TACAS* (2004), LNCS, vol. 2988, Springer, Berlin, Heidelberg, pp. 31–45.

[114] Renieris, M., and Reiss, S. Fault localization with nearest neighbor queries. In *ASE* (2003), IEEE Computer Society, pp. 30–39.

[115] Saidi, H., and Shankar, N. Abstract and model check while you prove. In *CAV 99* (1999), LNCS, vol. 4111, Springer-Verlag, Berlin, Heidelberg, pp. 219–242.

[116] Schmalz, M., Varacca, D., and Volzer, H. Counterexamples in probabilistic ltl model checking for markov chains. In *International Conference on Concurrency Theory (CONCUR)* (2009), LNCS, vol. 5710, Springer, Berlin, Heidelberg, pp. 787–602.

[117] Schuppan, V., and Biere, A. Shortest counterexamples for symbolic model checking of ltl with past. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2005), LNCS, vol. 3440, Springer, Berlin, Heidelberg, pp. 493–509.

[118] Shen, S., Qin, Y., and Li, S. Bug localization of hardware system with control flow distance minimization. In *13th IEEE International Workshop on Logic and Synthesis (IWLS 2004)* (2004).

[119] Shen, S., Qin, Y., and Li, S. Localizing errors in counterexample with iteratively witness searching. In *ATVA* (2004), LNCS, vol. 3299, Springer, Berlin, Heidelberg, pp. 456–469.

[120] Shen, S., Qin, Y., and Li, S. Minimizing counterexample with unit core extraction and incremental sat. In *Verification, Model Checking, and Abstract Interpretation* (2005), LNCS, vol. 3385, Springer, Berlin, Heidelberg, pp. 298–312.

[121] Shen, S., and Y. Qin, S. L. Localizing errors in counterexample with iteratively witness searching. In *ATVA 2004* (2004), LNCS, vol. 3299, Springer, Berlin, Heidelberg, pp. 459–464.

[122] Shen, S.-Y., Qin, Y., and Li, S. A fast counterexample minimization approach with refutation analysis and incremental sat. In *Conference on Asia South Pacific Design Automation* (2005), pp. 451–454.

[123] Sheyner, O., Haines, J., , Jha, S., Lippmann, R., and Wing, J. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privecy 2002* (2002), pp. 273–284.

[124] Tan, J., Avrunin, G., and Leue, S. Heuristic-guided counterexample search in flavers. In *12th ACM SIGSOFT international symposium on Foundations of software engineering* (2004), pp. 201–210.

[125] Tan, L., Sokolsky, O., and Lee, I. Specification-based testing with linear temporal logic. In *Proceedings of IEEE International Conference on Information Reuse and Integration* (2004), pp. 493–498.

[126] Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM Journal of Computing 1*, 2 (1972), 146–160.

[127] Tip, F., and Dinesh, T. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology1 10*, 1 (2001), 5–55.

[128] Touati, H. J., Brayton, R. K., and Kurshan, R. P. Testing language containment for ω automata using bdds. In *International Workshop on Formal Methods in VLSI Design* (1991), pp. 371–382.

[129] Valmari, A., and Geldenhuys, J. Tarjans algorithm makes on-the-fly ltl verification more efficient. In *Jensen, K., Podelski, A. (eds.) TACAS* (2004), LNCS, vol. 2988, Springer, Berlin, Heidelberg, pp. 205–219.

[130] Vardi, M., Wolper, P., and Yannakakis, M. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design 1*, 2 (1992), 275–288.

[131] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. *Automated Software Engineering Journal 10*, 2 (2003), 203–222.

[132] Wang, C., Yang, Z., Ivancic, F., and Gupta, A. Whodunit? causal analysis for counterexamples. In *4th International Symposium, ATVA* (2006), LNCS, vol. 4218, Springer, Berlin, Heidelberg, pp. 82–95.

[133] Wimmer, R., Braitling, B., and Becker, B. Counterexample generation for discrete-time markov

chains using bounded model checking. In *Verification, Model Checking, and Abstract Interpretation* (2009), LNCS, vol. 5403, Springer, Berlin, Heidelberg, pp. 366–380.

[134] Wimmer, R., Jansen, N., Abraham, E., Becker, B., and Katoen, J. Minimal critical subsystems for discrete-time markov models. In *TACAS* (2012), LNCS, vol. 7214, Springer, Berlin, Heidelberg, pp. 299–314.

[135] Wimmer, R., Jansen, N., and Vorpahl, A. High-level counterexamples for probabilistic automata. In *Quantitative Evaluation of Systems (QEST)* (2013), LNCS, vol. 8054, Springer, Berlin, Heidelberg, pp. 39–54.

[136] Xie, A., and Beerel, P. A. Implicit enumeration of strongly connected components. In *International Conference on ComputerAided Design* (1999), pp. 37–40.

[137] Zeller, A. Yesterday, my program worked. today, is does not. why? In *ACM Symposium on the Foundations of Software Engineering* (1999), pp. 253–267.

[138] Zeller, A. Isolating cause-effect chains for computer programs. In *ACM Symposium on the Foundations of Software Engineering* (2002), pp. 1–10.