Explaining Counterexamples Using Causality*

Ilan Beer[1], Shoham Ben-David[2], Hana Chockler[1], Avigail Orni[1], and Richard Trefler[2]

[1] IBM Research
Mount Carmel, Haifa 31905, Israel
{beer,hanac,ornia}@il.ibm.com
[2] David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
{s3bendav,trefler}@cs.uwaterloo.ca

Abstract. When a model does not satisfy a given specification, a counterexample is produced by the model checker to demonstrate the failure. A user must then examine the counterexample trace, in order to visually identify the failure that it demonstrates. If the trace is long, or the specification is complex, finding the failure in the trace becomes a non-trivial task. In this paper, we address the problem of analyzing a counterexample trace and highlighting the failure that it demonstrates. Using the notion of *causality*, introduced by Halpern and Pearl, we formally define a set of causes for the failure of the specification on the given counterexample trace. These causes are marked as red dots and presented to the user as a visual explanation of the failure. We study the complexity of computing the exact set of causes, and provide a polynomial-time algorithm that approximates it. This algorithm is implemented as a feature in the IBM formal verification platform RuleBase PE, where these visual explanations are an integral part of every counterexample trace. Our approach is independent of the tool that produced the counterexample, and can be applied as a light-weight external layer to any model checking tool, or used to explain simulation traces.

1 Introduction

Model checking [9,27] is a method for verifying that a finite-state concurrent system (a *model*) is correct with respect to a given specification. An important feature of model checking tools is their ability to provide, when the specification does not hold in a model, a *counterexample* [10]: a trace that demonstrates the failure of the specification in the model. This allows the user to analyze the failure, understand its source(s), and fix the specification or model accordingly. In many cases, however, the task of understanding the counterexample is challenging, and may require a significant manual effort.

There are different aspects of *understanding* a counterexample. In recent years, the process of finding the source of a bug has attracted significant attention. Many works have approached this problem (see [12,24,14,3,19,20,6,29,32,18,30,31] for a partial list), addressing the question of finding the root cause of the failure in the *model*, and

* This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 94–108, 2009.

proposing automatic ways to extract more information about the model, to ease the debugging procedure. Naturally, the algorithms proposed in the above mentioned works involve implementation in a specific tool (for example, the BDD procedure of [24] would not work for a SAT based model checker like those of [19,3]).

We address a different, more basic aspect of understanding a counterexample: the task of finding the failure in the trace itself. To motivate our approach, consider a verification engineer, who is formally verifying a hardware design written by a logic designer. The verification engineer writes a specification — a temporal logic formula — and runs a model checker, in order to check the formula on the design. If the formula fails on the design-under-test (DUT), a counterexample trace is produced and displayed in a trace viewer. The verification engineer does not attempt to debug the DUT implementation (since that is the responsibility of the the logic designer who wrote it). Her goal is to look for some basic information about the manner in which the formula fails on the specific trace. For example, if the formula is a safety property, the first question is *when* the formula fails (at what cycle in the trace). If the formula is a complex combination of several conditions, she needs to know which of these conditions has failed. These basic questions are prerequisites to deeper investigations of the failure.

Answering these questions can be done without any knowledge about the inner workings of the DUT, relying only on the given counterexample and the failed formula. Moreover, even in the given trace, only the signals that appear in the formula are relevant for these basic questions, and any other signals may be ignored. If the failed specification is simple enough, the preliminary analysis can be done manually without much effort. For example, if the failed specification is *`ERROR` never occurs*, then the user can visually scan the behavior of the signal `ERROR` in the trace, and find a time point at which `ERROR` holds, i.e., a place where the Boolean invariant $\neg$`ERROR` fails. In practice, however, the Boolean invariant in the specification may be complex, involving multiple signals and Boolean operations between them, in which case the visual scan becomes non-trivial. If the specification involves temporal operators (such as an LTL formula with operators $\mathbf{X}$ or $\mathbf{U}$), then the visual scan becomes even more difficult, since relations between several trace cycles must be considered. This is the point where trace explanations come into play. Explanations show the user the points in the trace that are relevant for the failure, allowing her to focus on these points and ignore other parts of the trace. They are displayed visually in a trace viewer.

We present a method and a tool for explaining the trace itself, without involving the model from which it was extracted. Thus, our approach has the advantage of being light-weight (no size problems are involved, as only one trace is considered at a time) as well as independent: it can be added as an external layer to any model-checking tool, or applied to explanation of simulation traces.

An explanation of a counterexample deals with the question: *what values on the trace cause it to falsify the specification?* Thus, we face the problem of *causality*. The philosophy literature, going back to Hume [23], has long been struggling with the problem of what it means for one event to cause another. We relate the formal definition of causality of Halpern and Pearl [22] to explanations of counterexamples. The definition of causality used in [22], like other definitions of causality in the philosophy literature, is based on *counterfactual dependence*. Event A is said to be a *cause* of event B if, had A not happened (this is the counterfactual condition, since A did in fact happen) then B would not have happened. Unfortunately, this definition does not capture all the subtleties involved with causality. The following story, presented by Hall in [21], demonstrates some of the difficulties in this definition. Suppose that Suzy and Billy both pick up rocks and throw them at a bottle. Suzy's rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy's would have shattered the bottle had it not been preempted by Suzy's throw. Thus, according to the counterfactual condition, Suzy's throw is not a cause for shattering the bottle (because if Suzy wouldn't have thrown her rock, the bottle would have been shattered by Billy's throw). Halpern and Pearl deal with this subtlety by, roughly speaking, taking A to be a cause of B if B counterfactually depends on A under some contingency. For example, Suzy's throw is a cause of the bottle shattering because the bottle shattering counterfactually depends on Suzy's throw, under the contingency that Billy doesn't throw.

We adapt the causality definition of Halpern and Pearl from [22] to the analysis of a counterexample trace π with respect to a temporal logic formula φ. We view a trace as a matrix of values, where an entry (i, j) corresponds to the value of variable i at time j. We look for those entries in the matrix that are causes for the first failure of φ on π, according to the definition in [22]. To demonstrate our approach, let us consider the following example.

Example. A transaction begins when `START` is asserted, and ends when `END` is asserted. Some unbounded number of time units later, the signal `STATUS_VALID` is asserted. Our specification requires that a new transaction must not begin before the `STATUS_VALID` of the previous transaction has arrived and `READY` is indicated[1]. A counterexample for this specification may look like the computation path π shown in Fig. 1.

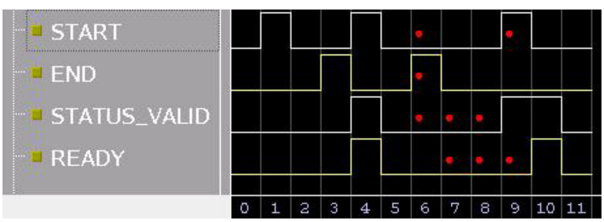

Fig. 1. A counterexample with explanations

In this example, the failure of the specification on the trace is not trivially evident. Our explanations, displayed as *red dots*[2], attract the user's attention to the relevant places, to help in identifying the failure. Note that each dot r is a *cause* of the failure of φ on the trace: switching the value of r would, under some contingency on the

[1] The precise specification is slightly more complex, and can be written in LTL as $\mathbf{G}((\neg\texttt{START} \wedge \neg\texttt{STATUS\_VALID} \wedge \texttt{END}) \rightarrow \mathbf{X}[\neg\texttt{START}\ \mathbf{U}\ (\texttt{STATUS\_VALID} \wedge \texttt{READY})])$.

[2] We refer to these explanations as red dots, since this is their characteristic color in RuleBase PE.

other values, change the value of φ on π. For example, if we switch the value of `START` in state 9 from 1 to 0, φ would not fail on the given trace anymore (in this case, no contingency on the other values is needed). Thus the matrix entry of the variable `START` at time 9 is indicated as a cause.

We show that the complexity of detecting an exact causal set is NP-complete, based on the complexity result for causality in binary models (see [17]). We then present an over-approximation algorithm whose complexity is linear in the size of the formula and in the length of the trace. The implementation of this algorithm is a feature in IBM's formal verification platform *RuleBase PE* [28]. We demonstrate that it produces the exact causal set for practical examples.

The rest of the paper is organized as follows. In Section 2 we give definitions. Section 3 is the main section of the paper, where we define causality in counterexamples, analyze the complexity of its computation and provide an efficient over-approximation algorithm to compute a causal set. In Section 4 we discuss the implementation of our algorithm on top of RuleBase PE. We show the graphical visualization used in practice, and present experimental results, demonstrating the usefulness of the method. Section 5 discusses related work, and Section 6 concludes the paper. Due to the lack of space, proofs are omitted from this version, and appear in the technical report [4].

2 Preliminaries

Kripke Structures

Let V be a set of Boolean variables. A *Kripke structure* K over V is a tuple $K = (S, I, R, L)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$. The labeling function $L : S \to 2^V$ labels each state with the set of variables true in that state. We say that $\pi = s_0, s_1, ...$ is a *path* in K if $s_0 \in I$ and $\forall i$, $(s_i, s_{i+1}) \in R$. We denote $\pi[j..k]$ a sub-path of π, that starts at s_j and ends at state s_k. The concatenation of a finite prefix $\pi[0..k]$ with an infinite path ρ is denoted $(\pi[0..k]) \cdot \rho$. We sometimes use the term "signals" for variables and the term "traces" for paths (as common in the terminology of hardware design).

Linear Temporal Logic (LTL)

Formulas of LTL are built from a set V of Boolean variables and constants **true** and **false** using Boolean operators $\neg$, $\rightarrow$, $\wedge$ and $\vee$, and the temporal operators $\mathbf{X}$, $\mathbf{U}$, $\mathbf{W}$, $\mathbf{G}$, $\mathbf{F}$ (see [11] for the definition of LTL semantics). An occurrence of a sub-formula ψ of φ is said to have a *positive polarity* if it appears in φ under an even number of negations, and a *negative polarity* otherwise (note that an antecedent of an implication is considered to be under negation).

3 Causality in Counterexamples

In this section, we define causality in counterexamples based on the definition of causality by Halpern and Pearl [22]. We demonstrate our definitions on several examples, discuss complexity of computing causality in counterexamples, and present a linear-time over-approximation algorithm for computing the set of causes. The original definition of [22] can be found in the appendix.

3.1 Defining Causality in Counterexamples

A *counterexample* to an LTL formula φ in a Kripke structure K is a computation path $\pi = s_0, s_1, \ldots$ such that $\pi \not\models \varphi$. For a state s_i and a variable v, the labeling function L of K maps the pair $\langle s_i, v\rangle$ to $\{0, 1\}$ in a natural way: $L(\langle s_i, v\rangle) = 1$ if $v \in s_i$, and 0 otherwise. For a pair $\langle s, v\rangle$ in π, we denote by $\langle \hat{s}, v\rangle$ the pair that is derived from $\langle s, v\rangle$ by switching the labeling of v in s. Let π be a path, s a state in π and v a variable in the labeling function. We denote $\pi^{\langle \hat{s}, v\rangle}$ the path derived from π by switching the labeling of v in s on π. This definition can be extended for a set of pairs A: we denote $\hat{A}$ the set $\{\langle \hat{s}, v\rangle | \langle s, v\rangle \in A\}$. The path $\pi^{\hat{A}}$ is then derived from π by switching the values of v in s for all pairs $\langle s, v\rangle \in A$. The definition below is needed in the sequel.

Definition 1 (Bottom value). *For a Kripke structure $K = (S, I, R, L)$, a path π in K, and a formula φ, a pair $\langle s, v\rangle$ is said to have a* bottom value *for φ in π, if, for at least one of the occurrences of v in φ, $L(\langle s, v\rangle) = 0$ and v has a* positive *polarity in φ, or $L(\langle s, v\rangle) = 1$ and v has a* negative *polarity in φ.*

Note that while the value of a pair $\langle s, v\rangle$ depends on the computation path π only, determining whether $\langle s, v\rangle$ has a *bottom value* depends also on the polarity of v in φ. Thus, if v appears in multiple polarities in φ, then $\langle s, v\rangle$ has both a bottom value (with respect to one of the occurrences) and a non-bottom value (with respect to a different occurrence) for any state s.

Before we formally define causes in counterexamples we need to deal with one subtlety: the value of φ on finite paths. While computation paths are infinite, it is often possible to determine that $\pi \not\models \varphi$ after a *finite* prefix of the path. Thus, a counterexample produced by a model checker may demonstrate a finite execution path.

In this paper, we use the semantics of LTL model checking on truncated paths as defined by Eisner et al. in [16]. The main advantage of this semantics is that it preserves the complexity of LTL model checking. Since it is quite complicated to explain, instead we present a simpler definition (due to [16]), which coincides with the definition of Eisner et al. on almost all formulas.

Definition 2. *Let $\pi[0..k]$ be a finite path and φ an LTL formula. We say that:*

1. *The value of φ is* **true** *in $\pi[0..k]$ (denoted $\pi[0..k] \models_f \varphi$, where $\models_f$ stands for "finitely models") iff for all infinite computations ρ, we have $\pi[0..k] \cdot \rho \models \varphi$;*
2. *The value of φ is* **false** *in $\pi[0..k]$ (denoted $\pi[0..k] \not\models_f \varphi$, where $\not\models_f$ stands for "finitely falsifies") iff for all infinite computations ρ, we have $\pi[0..k] \cdot \rho \not\models \varphi$;*
3. *The value of φ in π is* **unknown** *(denoted $\pi[0..k]$? φ) iff there exist two infinite computations ρ_1 and ρ_2 such that $\pi[0..k] \cdot \rho_1 \models \varphi$ and $\pi[0..k] \cdot \rho_2 \not\models \varphi$.*

Let φ be an LTL formula that fails on an infinite path $\pi = s_0, s_1, \ldots$, and let k be the smallest index such that $\pi[0..k] \not\models_f \varphi$. If φ does not fail on any finite prefix of π, we take $k = \infty$ (then $\pi[0..\infty]$ naturally stands for π, and we have $\pi \not\models \varphi$). We can now define the following.

Definition 3 (Criticality in counterexample traces). *A pair $\langle s, v\rangle$ is* critical *for the failure of φ on $\pi[0..k]$ if $\pi[0..k] \not\models_f \varphi$, but either $\pi^{\langle \hat{s}, v\rangle}[0..k] \models_f \varphi$ or $\pi^{\langle \hat{s}, v\rangle}[0..k]\ ?\ \varphi$.*

That is, switching the value of v in s changes the value of φ on $\pi[0..k]$ (to either **true** or **unknown**). As a simple example, consider the formula $\varphi = \mathbf{G}p$, on $\pi = s_0, s_1, s_2$, labeled $p{\cdot}p{\cdot}\neg p$. Then, $\pi[0..2] \not\models_f \varphi$, and $\langle s_2, p\rangle$ is critical for this failure, since switching the value of p in state s_2 changes the value of φ on $\pi^{\langle \hat{s}_2, p\rangle}$ to **unknown**.

Definition 4 (Causality in counterexample traces). *A pair $\langle s, v\rangle$ is a* cause *of the first failure of φ on $\pi[0..k]$ if k is the smallest index such that $\pi[0..k] \not\models_f \varphi$, and there exists a set A of* bottom-valued *pairs, such that $\langle s, v\rangle \notin A$, and the following hold:*

- *$\pi^{\hat{A}}[0..k] \not\models_f \varphi$, and*
- *$\langle s, v\rangle$ is* critical *for the failure of φ on $\pi^{\hat{A}}[0..k]$.*

A pair $\langle s, v\rangle$ is defined to be a *cause* for the first failure of φ on π, if it can be made *critical* for this failure by switching the values of some bottom-valued pairs. Note that according to this definition, only bottom-valued pairs can be causes.

Note that a trace π may have more than one failure, as we demonstrate in the examples below. Our experience shows that the first failure is the most interesting one for the user. Also, focusing on one failure naturally reduces the set of causes, and thus makes it easier for the user to understand the explanation.

Examples

1. Consider $\varphi_1 = \mathsf{G}p$ and a path $\pi_1 = s_0, s_1, s_2, s_3, (s_4)^\omega$ labeled as $(p) \cdot (p) \cdot (\neg p) \cdot (\neg p) \cdot (p)^\omega$. The shortest prefix of π_1 on which φ_1 fails is $\pi_1[0..2]$. $\langle s_2, p\rangle$ is critical for the failure of φ on $\pi_1[0..2]$, because changing its value from 0 to 1 changes the value of φ on $\pi_1[0..2]$ from **false** to **unknown**. Also, there are no other bottom-valued pairs in $\pi_1[0..2]$, thus there are no other causes, which indeed meets our intuition.
2. Consider $\varphi_2 = \mathsf{F}p$ and a path $\pi_2 = (s_0)^\omega = (\neg p)^\omega$. The formula φ_2 fails in π_2, yet it does not fail on any finite prefix of π_2. Note that changing the value of any $\langle s_i, p\rangle$ for $i \geq 0$ results in the satisfaction of φ on π, thus all pairs $\{\langle s_i, p\rangle : i \in \mathbb{N}\}$ are critical and hence are causes for the failure of φ_2 on π_2.
3. The following example demonstrates the difference between criticality and causality. Consider $\varphi{=}\mathbf{G}(a \wedge b \wedge c)$ and a path $\pi_3 = s_0, s_1, s_2, \ldots$ labeled as $(\emptyset)^\omega$ (see Figure 2). The formula φ_3 fails on s_0, however, changing the value of any signal in one state does not change the value of φ_3. There exists, however, a set A of bottom-valued pairs whose change makes the value of a in s_0 critical: $A = \{\langle s_0, b\rangle, \langle s_0, c\rangle\}$. Similarly, $\langle s_0, b\rangle$ and $\langle s_0, c\rangle$ are also causes.

3.2 Complexity of Computing Causality in Counterexamples

The complexity of computing causes for counterexamples follows from the complexity of computing causality in binary causal models defined in [22] (see Section 2).

Lemma 5. *Computing the set of causes for falsification of a linear-time temporal specification on a single trace is NP-complete.*

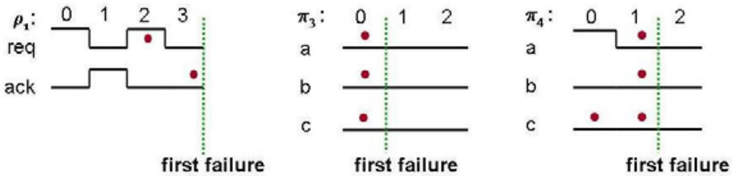

Fig. 2. Counterexample traces

Proof Sketch. Informally, the proof of NP-hardness is based on the reduction from computing causality in binary causal models to computing causality in counterexamples. The problem of computing causality in binary causal models is NP-complete [17]. The reduction from binary causal models to Boolean circuits and from Boolean circuits to model-checking, shown in [8], is based on the automata-theoretic approach to branching-time model checking ([25]), and proves that computing causality in model checking of branching time specifications is NP-complete. On a single trace linear-time and branching temporal logics coincide, and computing the causes for satisfaction is easily reducible to computing the causes for falsification.

The proof of membership in NP is straightforward: given a path π and a formula φ that is falsified on π, the number of pairs $\langle s, v \rangle$ is $|\varphi| \cdot |\pi|$; for a pair $\langle s, v \rangle$, we can non-deterministically choose a set A of bottom-valued pairs; checking whether changing L on S makes $\langle s, v \rangle$ critical for the falsification of φ requires model-checking φ on the modified π twice, and thus can be done in linear time.[3] $\square$

The problem of computing the exact set of causes can be translated into a SAT problem using the reduction from [17], with small changes due to the added semantics of satisfiability on finite paths. However, the natural usage of explanation of a counterexample is in an interactive tool (and indeed this is the approach we use in our implementation), and therefore the computation of causes should be done while the user is waiting. Thus, it is important to make the algorithm as efficient as possible. We describe an efficient approximation algorithm for computing the set of causes in Section 3.3, and its implementation results are shown in Section 4.

3.3 An Approximation Algorithm

The counterexamples we work with have a finite number of states. When representing an infinite path, the counterexample will contain a loop indication, i.e., an indication that the last state in the counterexample is equal to one of the earlier states.

Let φ be a formula, given in negation normal form, and let $\pi[0..k] = s_0, s_1, ..., s_k$ be a non-empty counterexample for it, consisting of a finite number of states and a possible loop indication.

We denote by $\pi[i..k]$ the suffix of $\pi[0..k]$ that starts at s_i. The procedure C below produces $C(\pi[i..k], \psi)$, the approximation of the set of causes for the failure of a

[3] Note that the proof of membership in NP relies on model-checking of truncated paths having the same complexity as model-checking of infinite paths [16].

sub-formula ψ on the suffix of $\pi[0..k]$ that starts with s_i. We invoke the procedure C with the arguments $(\pi[0..k], \varphi)$ to produce the set of causes for the failure of φ on $\pi[0..k]$.

If the path $\pi[0..k]$ has a loop indication, the invocation of C is preceded by a pre-processing stage, in which we unwrap the loop $|\varphi| + 1$ times. This gives us a different finite representation, $\pi'[0..k']$ (with $k' \geq k$), for the same infinite path[4]. We then execute $C(\pi'[i..k'], \varphi)$, which returns a set of causes that are pairs in $\pi'[0..k']$. These pairs can then be (optionally) mapped back to pairs in the original representation $\pi[0..k]$, by shifting each index to its corresponding index in the first copy of the loop[5].

During the computation of $C(\pi[i..k], \varphi)$, we use the auxiliary function val, that evaluates sub-formulas of φ on the given path. It returns 0 if the sub-formula fails on the path and 1 otherwise. The computation of val is done in parallel with the computation of the causality set, and relies on recursively computed causality sets for sub-formulas of φ. The value of val is computed as follows:

- $val(\pi[i..k], true) = 1$
- $val(\pi[i..k], false) = 0$
- For any formula $\varphi \notin \{true, false\}$, $val(\pi[i..k], \varphi) = 1$ iff $C(\pi[i..k], \varphi) = \emptyset$

Algorithm 6 (Causality Set). *An over-approximated causality set C for $\pi[i..k]$ and ψ is computed as follows*

- $C(\pi[i..k], true) = C(\pi[i..k], false) = \emptyset$
- $C(\pi[i..k], p) = \begin{cases} \{\langle s_i, p\rangle\} & \text{if } p \notin L(s_i) \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \neg p) = \begin{cases} \{\langle s_i, p\rangle\} & \text{if } p \in L(s_i) \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \mathbf{X}\varphi) = \begin{cases} C(\pi[i+1..k], \varphi) & \text{if } i < k \\ \emptyset & \text{otherwise} \end{cases}$
- $C(\pi[i..k], \varphi \wedge \psi) = C(\pi[i..k], \varphi) \cup C(\pi[i..k], \psi)$
- $C(\pi[i..k], \varphi \vee \psi) =$
$$\begin{cases} C(\pi[i..k], \varphi) \cup C(\pi[i..k], \psi) & \text{if } val(\pi[i..k], \varphi) = 0 \text{ and } val(\pi[i..k], \psi) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$
- $C(\pi[i..k], \mathbf{G}\varphi) =$
$$\begin{cases} C(\pi[i..k], \varphi) & \text{if } val(\pi[i..k], \varphi) = 0 \\ C(\pi[i+1..k], \mathbf{G}\varphi) & \text{if } val(\pi[i..k], \varphi) = 1 \text{ and } i < k \text{ and } val(\pi[i..k], \mathbf{XG}\varphi) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$
- $C(\pi[i..k], [\varphi \mathbf{U} \psi]) =$
$$\begin{cases} C(\pi[i..k], \psi) \cup C(\pi[i..k], \varphi) & \text{if } val(\pi[i..k], \varphi) = 0 \text{ and } val(\pi[i..k], \psi) = 0 \\ C(\pi[i..k], \psi) & \text{if } val(\pi[i..k], \varphi) = 1 \text{ and } val(\pi[i..k], \psi) = 0 \\ & \quad \text{and } i = k \\ C(\pi[i..k], \psi) \cup C(\pi[i+1..k], [\varphi \mathbf{U} \psi]) & \text{if } val(\pi[i..k], \varphi) = 1 \text{ and } val(\pi[i..k], \psi) = 0 \\ & \quad \text{and } i < k \text{ and } val(\pi[i..k], \mathbf{X}[\varphi \mathbf{U} \psi]) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

---

[4] It can be shown, by straightforward induction, that φ fails on the finite path with $|\varphi| + 1$ repetitions of the loop iff it fails on the infinite path.

[5] This mapping is necessary for presenting the set of causes graphically on the original path $\pi[0..k]$.

The procedure above recursively computes a set of causes for the given formula φ on the suffix of a counterexample $\pi[i..k]$. At the proposition level, p is considered a cause in the current state if and only if it has a bottom-value in the state. At every level of the recursion, a sub-formula is considered relevant (that is, its exploration can produce causes for falsification of the whole specification) if it has a value of **false** at the current state.

Lemma 7. *The complexity of Algorithm 6 is linear in k and in $|\varphi|$.*

Proof Sketch. The complexity follows from the fact that each subformula ψ of φ is evaluated at most once at each state s_i of the counterexample π. Similarly to global CTL model checking, the decision at each state is made based on the local information.

For paths with a loop, unwrapping the loop $|\varphi| + 1$ times does not add to this complexity, since each subformula ψ is evaluated on a single copy of the loop, based on its depth in φ. □

Theorem 8. *The set of pairs produced by Algorithm 6 for a formula φ on a path π is an over-approximation of the set of causes for φ on π according to Definition 4.*

Proof Sketch. The proof uses the automata-theoretic approach to branching-time model checking[6] introduced in [25], where model checking is reduced to evaluating an AND-OR graph with nodes labeled with pairs $\langle$ state s of π, subformula ψ of $\varphi\rangle$. The leaves are labeled with $\langle s, l\rangle$, with l being a literal (variable or its negation), and the root is labeled with $\langle s_0, \varphi\rangle$. The values of leaves are determined by the labeling function L, and the value of each inner node is 0 or 1 according to the evaluation of its successors. Since φ fails on π, the value of the root is 0. A path that starts from a bottom-valued leaf, and on the way up visits only nodes with value 0, is called a **failure path**. The proof is based on the following fact: if $\langle s, v\rangle$ is a cause of failure of φ on π according to Definition 4, then it is a bottom-valued pair and there exists a failure path from the leaf labeled with $\langle s, v\rangle$ to the root. By examining Algorithm 6, it can be proved that the set of pairs that is the result of the algorithm is exactly the set of bottom-valued leaves from which there exists a failure path to the root. Thus, it contains the exact set of causes. □

We note that not all bottom-valued leaves that have a failure path to the root are causes (otherwise Algorithm 6 would always give accurate results!). In our experience, Algorithm 6 gives accurate results for the majority of real-life examples. As an example of a formula on which Algorithm 6 does not give an accurate result, consider $\varphi_4 = a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$ and a trace $\pi_4 = s_0, s_1, s_2, \ldots$ labeled as $a \cdot (\emptyset)^\omega$ (see Figure 2). The formula φ_4 fails on π_4, and $\pi_4[0..1]$ is the shortest prefix on which it fails. What is the set of causes for failure of φ_4 on $\pi_4[0..1]$? The pair $\langle s_0, a\rangle$ is not a cause, since it is not bottom-valued. Checking all possible changes of sets of bottom-valued pairs shows that $\langle s_0, b\rangle$ is not a cause. On the other hand, $\langle s_1, a\rangle$ and $\langle s_1, b\rangle$ are causes because changing the value of a in s_1 from 0 to 1 makes φ_4 **unknown** on $\pi_4[0..1]$, and similarly for $\langle s_1, b\rangle$. The pairs $\langle s_0, c\rangle$ and $\langle s_1, c\rangle$ are causes because changing the value of c in either s_0 or s_1 from 0 to 1 changes the value of φ_4 to **true** on $\pi_4[0..1]$. The values of

[6] Recall, that on a single trace LTL and CTL coincide.

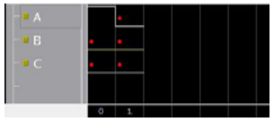

Fig. 3. Counterexample for $a\,\mathbf{U}\,(b\,\mathbf{U}\,c)$

signals in s_2 are not causes because the first failure of φ_4 happens in s_1. The causes are represented graphically as red dots in Figure 2. By examining the algorithm, we can see that on φ_4 and π_4 it outputs the set of pairs that contains, in addition to the exact set of causes, the pair $\langle s_0, b\rangle$. The results of running the implementation of Algorithm 6 on φ_4 are presented in Figure 3.

4 Implementation and Experimental Results

Visual counterexample explanation is an existing feature in RuleBase PE, IBM's formal verification platform. When the formal verification tool displays a counterexample in its trace viewer, it shows trace explanations as red dots at several $\langle position, signal\rangle$ locations on the trace.

The approximation algorithm presented in Section 3.3 is applied to every counterexample produced by the model checker[7]. The algorithm receives the counterexample trace and formula as input, and outputs a set of $\langle position, signal\rangle$ pairs. This set of pairs is passed on to the tool's built-in trace viewer, which displays the red dots at the designated locations.

Users of the formal verification tool may select one or more model-checking engines with which to check each formula. Each engine uses a different model-checking algorithm, as described in [5]. When a formula fails, the engine that finds the failure is also responsible for producing a counterexample trace. Thus, a counterexample viewed by a user may have been generated by any one of the various algorithms. The red dots computation procedure runs independently on any counterexample, and is oblivious to the method by which the trace was obtained.

The red dots are a routine part of the workflow for users. The red dots computation is performed automatically by the tool, behind the scenes, with no user intervention. Displaying the red dots on the trace is also done automatically, every time a user activates the trace viewer. User feedback indicates that red dots are seen as helpful and important, and are relied on as an integral part of trace visualization.

Experimental Results

We ran the RuleBase PE implementation of Algorithm 6 on the following pairs of formulas and counterexamples. These formulas are based on real-world specifications from [1]. In all cases presented below, the output of the approximation algorithm is the exact causality set, according to Definition 4.

[7] Commercially available versions of RuleBase PE implement an older version of this algorithm.

1. We consider a formula with a complex Boolean invariant

$$\mathbf{G}\,((\mathtt{STATUS\_VALID} \wedge \neg\mathtt{LARGE\_PACKET\_MODE} \wedge \mathtt{LONG\_FRAME\_RECEIVED}) \rightarrow ((\mathtt{LONG\_FRAME\_ERROR} \wedge \neg\mathtt{STATUS\_OK}) \vee \mathtt{TRANSFER\_STOPPED}))$$

The trace in Figure 4 was produced by a SAT-based BMC engine, which was configured to increase its bound by increments of 20. As a result, the trace is longer than necessary, and the failure does not occur on the last cycle of the trace. The red dots point us to the first failure, at cycle 12. Note that the Boolean invariant also fails at cycle 15, but this later failure is not highlighted. The execution time for the red dots computation on the trace in Figure 4 is less than 1 second. Running with the same formula, but with traces of up to 5000 cycles, gives execution times that are still under 2 seconds.

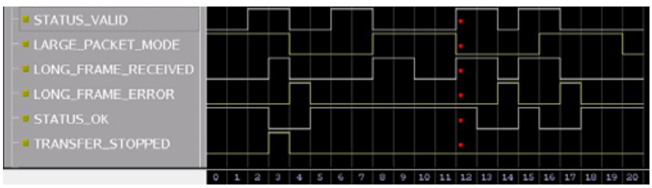

Fig. 4. Counterexample for a Boolean invariant

2. For a liveness property, the model checking engine gives a counterexample with a loop. This is marked by the signal LOOP in the trace viewer, as seen in the trace in Figure 5, which is a counterexample for the formula

$$\mathbf{G}\,(\mathtt{P1\_ACTIVE} \rightarrow \mathbf{F}\ \mathtt{P2\_ACTIVE})$$

The LOOP signal rises at cycle 9, indicating that the loop starts at that cycle, and that the last cycle (cycle 11) is equal to cycle 9. The point at which the loop begins is marked with a red dot, in addition to the red dots computed by Algorithm 6. The execution time, when running on the trace in Figure 5, is less than 1 second. On a counterexample of 1000 cycles for the same formula, the execution time is less than 2 seconds. For 5000 cycles, the execution time reaches 20 seconds.

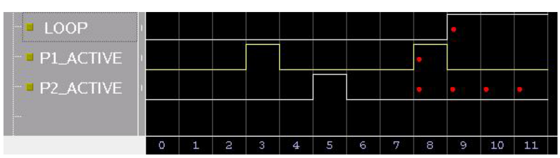

Fig. 5. Counterexample for a liveness property

3. In the introduction, we demonstrated red dot explanations for the formula

$$\mathbf{G}((\neg\mathtt{START} \wedge \neg\mathtt{STATUS\_VALID} \wedge \mathtt{END}) \rightarrow \mathbf{X}[\neg\mathtt{START}\ \mathbf{U}\ (\mathtt{STATUS\_VALID} \wedge \mathtt{READY})])$$

This formula contains a combination of temporal operators and Boolean conditions, which makes it difficult to visually identify a failure on the counterexample. The red dots produced by the approximation algorithm are shown in Figure 1. The execution time on the trace in Figure 1 is less than 1 second. For the same formula and a counterexample of 1000 cycles, the execution time is 70 seconds.

5 Related Work

There are several works that tie the definition of causality by Halpern and Pearl to formal verification. Most closely related to our work is the paper by Chockler et. al [8], in which causality and its quantitative measure, responsibility, are viewed as a refinement of coverage in model checking. Causality and responsibility have also been used to improve the refinement techniques of symbolic trajectory evaluation (STE) [7].

As discussed in the introduction, there has been much previous work addressing various aspects of understanding a counterexample. The most relevant to our work is a method that uses *minimal unsatisfiable cores* [13,31][8] to aid in debugging.

We note that minimal unsatisfiable cores differ from our notion of causality in two main aspects. First, a single unsatisfiable core does not give all relevant information for understanding the failure. For example, for the path π_3 in Figure 2 and the specification $\varphi_3 = \mathbf{G}(a \wedge b \wedge c)$, there are three minimal unsatisfiable cores: $\{\langle s_0, a\rangle\}$, $\{\langle s_0, b\rangle\}$, and $\{\langle s_0, c\rangle\}$. None of them gives the full information needed to explain the failure. In contrast, the set of causes is $\{\langle s_0, a\rangle, \langle s_0, b\rangle, \langle s_0, c\rangle\}$, consisting of all pairs that can be made critical for the failure of φ_3 on π_3.

The second difference is that our definition of causality refers to the *first* failure of the formula on the given path. Consider, for example, a specification $\varphi = \mathbf{G}(req \rightarrow \mathbf{X}ack)$ and a trace $\pi = (req) \cdot (ack) \cdot (req) \cdot (req) \cdot (\emptyset)^{\omega}$ (that is, a trace that is labeled with req in s_0, s_2, and s_3, and with ack in s_1). The minimal unsatisfiable cores in this example are $\{\langle s_2, req\rangle, \langle s_3, ack\rangle\}$ and $\{\langle s_3, req\rangle, \langle s_4, ack\rangle\}$. In contrast, the set of causes for the first failure of φ on π is $\{\langle s_2, req\rangle, \langle s_3, ack\rangle\}$. Unlike the previous example, here the set of causes differs from the union of minimal unsatisfiable cores.

We conjecture that if a given counterexample trace is of minimal length needed to demonstrate the failure, then the *union* of the minimal unsatisfiable cores is equal to the set of causes as defined in this paper. We note though, that computing the union of minimal unsatisfiable cores is a considerably harder task, both in the worst case complexity and in practice, than computing the set of causes directly. Indeed, it is easy to see that the problem of counting the number of all unsatisfiable cores for a given specification and a counterexample trace is complete in $\#P$, similarly to the problem of counting all satisfying assignments to a SAT problem. In practice, extracting several minimal unsatisfiable cores is done sequentially, each time backtracking from the current solution provided by the SAT solver and forcing it to look for a different solution (see, for example, [31]). In contrast, the problem of computing the set of causes is "only" NP-complete, making even the brute-force algorithm feasible for small specifications and

[8] An unsatisfiable core of an unsatisfiable CNF formula is a subset of clauses that is in itself unsatisfiable. A minimal unsatisfiable core is an unsatisfiable core such that removing any one of its clauses makes it satisfiable.

short traces. Essentially, computing the union of minimal unsatisfiable cores by computing all unsatisfiable cores separately is an "overkill": it gives us a set of sets, whereas we are only interested in one set - the union.

6 Conclusion and Future Directions

We have shown how the causality definition of Halpern and Pearl [22] can be applied to the task of explaining a counterexample. Our method is implemented as part of the IBM formal verification platform RuleBase PE [28], and it is applied to every counterexample presented by the tool. Experience shows that when visually presented as described in Section 4, the causality information substantially speeds up the time needed for the understanding of a counterexample. Since the causality algorithm is applied to a single counterexample and ignores the model from which it was extracted, no size issues are involved, and the execution time is negligible. An important advantage of our method is the fact that it is independent of the tool that produced the counterexample. When more than one model checking "engine" is invoked to verify a formula, as described in [5], the independence of the causality algorithm is especially important.

The approach presented in this paper defines and (approximately) detects a set of causes for the *first* failure of a formula on a trace. While we believe that this information is the most beneficial for the user, there can be circumstances where the sets of causes for other failures are also desirable. A very small and straightforward enhancement of the algorithm will allow to compute the (approximate) sets of causes of all or a subset of failures of the given counterexample.

As a future work, the relation between causes and unsatisfiable cores should be investigated further. Specifically, a conclusive answer should be provided for the conjecture mentioned in Section 5, about the union of unsatisfiable cores on a trace of minimal length.

In a different direction, it is interesting to see whether there exist subsets of LTL for which the computation of the exact set of causes is polynomial. Natural candidates for being such "easy" sublogics of LTL are the PSL simple subset defined in [15], and the common fragment of LTL and ACTL, called LTL^{det} (see [26]). Finally, we note that our approach, though demonstrated here for LTL specifications, can be applied to other linear temporal logics as well, with slight modifications. This is because our definition of cause holds for any monotonic temporal logic. It will be interesting to see whether it can also be extended to full PSL without significantly increasing its complexity.

Acknowledgments. We thank Cindy Eisner for helpful discussions on the truncated semantics. We thank the anonymous reviewers for important comments.

References

1. Prosyd: Property-Based System Design (2005), `http://www.prosyd.org/`
2. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.Y.: Resets vs. aborts in linear temporal logic. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 65–80. Springer, Heidelberg (2003)
3. Ball, T., Naik, M., Rajamani, S.: From symptom to cause: Localizing errors in counterexample traces. In: Proc. of POPL, pp. 97–105 (2003)

4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining Counterexamples Using Causality. IBM technical report number H-0266, `http://domino.watson.ibm.com/library/cyberdig.nsf/Home`
5. Ben-David, S., Eisner, C., Geist, D., Wolfsthal, Y.: Model checking at IBM. FMSD 22(2), 101–108 (2003)
6. Chechik, M., Gurfinkel, A.: A framework for counterexample generation and exploration. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 217–233. Springer, Heidelberg (2005)
7. Chockler, H., Grumberg, O., Yadgar, A.: Efficient automatic STE refinement using responsibility. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 233–248. Springer, Heidelberg (2008)
8. Chockler, H., Halpern, J.Y., Kupferman, O.: What causes a system to satisfy a specification? ACM TOCL 9(3) (2008)
9. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
10. Clarke, E., Grumberg, O., McMillan, K., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proc. 32nd DAC, pp. 427–432 (1995)
11. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Copty, F., Irron, A., Weissberg, O., Kropp, N., Kamhi, G.: Efficient debugging in a formal verification environment. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 275–292. Springer, Heidelberg (2001)
13. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
14. Dong, Y., Ramakrishnan, C.R., Smolka, S.A.: Model checking and evidence exploration. In: Proc. of ECBS, pp. 214–223 (2003)
15. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Series on Integrated Circuits and Systems (2006)
16. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
17. Eiter, T., Lukasiewicz, T.: Complexity results for structure-based causality. In: Proc. 7th IJCAI, pp. 35–40 (2001)
18. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for c programs. ENTCS 174(4), 95–111 (2007)
19. Groce, A.: Error explanation with distance metrics. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 108–122. Springer, Heidelberg (2004)
20. Groce, A., Kroening, D.: Making the most of BMC counterexamples. In: SGSH (July 2004)
21. Hall, N.: Two concepts of causation. Causation and Counterfactuals. MIT Press, Cambridge (2002)
22. Halpern, J., Pearl, J.: Causes and explanations: A structural-model approach — part I: Causes. In: Proc. of 17th UAI, pp. 194–202. Morgan Kaufmann Publishers, San Francisco (2001)
23. Hume, D.: A treatise of human nature. John Noon, London (1939)
24. Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 445–458. Springer, Heidelberg (2002)
25. Kupferman, O., Vardi, M., Wolper, P.: An automata-theoretic approach to branching-time model checking. JACM 47(2), 312–360 (2000)
26. Maidl, M.: The common fragment of CTL and LTL. In: Proc. of FOCS, pp. 643–652 (2000)
27. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in Cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1981)

28. RuleBase PE homepage,
http://www.haifa.il.ibm.com/projects/verification/RB_Homepage
29. Shen, S., Qin, Y., Li, S.: A faster counterexample minization algorithm based on refutation analysis. In: Proc. of DATE, pp. 672–677 (2005)
30. Staber, S., Bloem, R.: Fault localization and correction with QBF. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 355–368. Springer, Heidelberg (2007)
31. Sülflow, A., Fey, G., Bloem, R., Drechsler, R.: Using unsatisfiable cores to debug multiple design errors. In: Proc. of Symp. on VLSI, pp. 77–82 (2008)
32. Wang, C., Yang, Z., Ivancic, F., Gupta, A.: Whodunit? causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006)

A Causality Background

The formal definition of causality used in this paper was first introduced in the Artificial Intelligence community in [22]. Our definitions in Section 3 are based on a restricted (and much simpler) version that applies to *binary* causal models, where the range of all variables is binary [17]. We present the definition of a binary causal model below.

Definition 9 (Binary causal model). *A* binary causal model M *is a tuple* $\langle \mathcal{V}, \mathcal{F} \rangle$*, where* $\mathcal{V}$ *is the set of Boolean variables and* $\mathcal{F}$ *associates with every variable* $X \in \mathcal{V}$ *a function* F_X *that describes how the value of* X *is determined by the values of all other variables in* $\mathcal{V}$*. A* context $\vec{u}$ *is a legal assignment for the variables in* $\mathcal{V}$*.*

A causal model M is conveniently described by a *causal network*, which is a graph with nodes corresponding to the variables in $\mathcal{V}$ and an edge from a node labeled X to one labeled Y if F_Y depends on the value of X. We restrict our attention to what are called *recursive models*. These are ones whose associated causal network is a directed acyclic graph.

A *causal formula* η is a Boolean formula over the set of variables $\mathcal{V}$. A causal formula η is true or false in a causal model given a context. We write $(M, \vec{u}) \models \eta$ if η is true in M given a context $\vec{u}$. We write $(M, \vec{u}) \models [\vec{Y} \leftarrow \vec{y}](X = x)$ if the variable X has value x in the model M given the context $\vec{u}$ and the assignment $\vec{y}$ to the variables in the set $\vec{Y} \subset \mathcal{V}$.

For the ease of presentation, we borrow from [8] the definition of *criticality* in binary causal models, that captures the notion of counterfactual causal dependence.

Definition 10 (Critical variable [8]). *Let* M *be a model,* $\vec{u}$ *the current context, and* η *a Boolean formula. Let* $(M, \vec{u}) \models \eta$*, and* X *a Boolean variable in* M *that has the value* x *in the context* $\vec{u}$*, and* $\bar{x}$ *the other possible value (0 or 1). We say that* $(X = x)$ *is* critical *for* η *in* $(M, \vec{u})$ *iff* $(M, \vec{u}) \models (X \leftarrow \neg x)\neg\eta$*. That is, changing the value of* X *to* $\neg x$ *falsifies* η *in* $(M, \vec{u})$*.*

We can now give the definition of a cause in binary causal models from [22,17].

Definition 11 (Cause). *We say that* $X = x$ *is a* cause *of* η *in* $(M, \vec{u})$ *if the following conditions hold:*

AC1. $(M, \vec{u}) \models (X = x) \wedge \eta$.

AC2. *There exists a subset* $\vec{W}$ *of* $\mathcal{V}$ *with* $X \notin \vec{W}$ *and some setting* $\vec{w}'$ *of the variables in* $\vec{W}$ *such that setting the variables in* $\vec{W}$ *to the values* $\vec{w}'$ *makes* $(X = x)$ *critical for the satisfaction of* η*.*