H-0266 (H0901-013) January 18, 2009
Computer Science

IBM Research Report

Explaining a Counterexample Using Causality

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

[1]IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

[2]David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada

Explaining a Counterexample 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`

January 13, 2009

Abstract. When a model does not satisfy a given specification, a counterexample is produced by the model checker to demonstrate the failure. While this service is important and useful, the analysis of the counterexample in order to understand the problem it demonstrates, can be difficult and time-consuming.
In this paper, we address the problem of analyzing a counterexample. Using the notion of *causality*, introduced by Halpern and Pearl, we formally define a set of pairs $\langle$signal, location$\rangle$, which cause the failure of the specification on the counterexample trace. We prove that computation of the exact set of causes is intractable and provide a polynomial-time model checking based algorithm that approximates it. Experimental results, conducted on real-life examples suggest that the approximate algorithm computes causes that match the user's intuition on all examples. Our approach is independent of the tool that produced the counterexample and can be applied as an external layer to any model checking tool.

1 Introduction

Model checking ([CE81,QS81], c.f.[CGP00]) 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* [CGMZ95]: a single 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 a lot of attention. Many works have approached this problem (see [CIW$^+$01,JRS02,DRS03,BNR03,Gro04,GK04,CG05], [SQL05,WYIG06,GSB07,SB07,SFBD08] for a partial list), addressing the question of the cause of the failure in the *model*, and proposing automatic ways to extract more information about the model, to ease the debugging procedure. Naturally, the algorithms proposed by the above mentioned works involved implementation in a specific tool (for example, the BDD procedure of [JRS02] would not work for a SAT based model checker like those of [Gro04,BNR03]).

We address a different, more basic aspect of understanding a counterexample: the task of *matching* the verified specification φ with the trace that contradicts it. When a counterexample is presented to the user, the first phase in the analysis is to check that φ is indeed contradicted by the provided trace, and detect the contradicting places. This task, although it may seem simple, can become challenging and time-consuming when the trace involved is long and when φ is complex. 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-weighted (no size problem involved as only one trace is considered at a time) as well as independent: it can be applied as an external layer to any model checking tool. At the same time, we provide an invaluable debugging aid to the user.

An explanation of a counterexample deals with the question of *which events on the trace cause it to falsify the specification*. Thus, we face the problem of *causality*. The philosophy literature, going back to Hume [Hum39], 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 [HP01] to explanations of counterexamples. The definition of causality used in [HP01], 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, taken from [Hal02], 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 shuttered 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 [HP01] definition of causality to the analysis of a counterexample trace π with respect to a temporal logic formula φ. We view a trace as a set of pairs $\langle$location, signal$\rangle$, and look for the pairs that are causes for the failure of φ according to the definition in [HP01]. To demonstrate our approach, let us consider the following example.

Example 1 *A transaction begins when 'start' is asserted, and ends when 'end' is asserted. Some unbounded number of time units later, the signal 'status-valid' should be asserted. A new transaction must not begin before the 'status-valid' of the previous transaction has arrived. This specification can be written as*

$$G(start \rightarrow (\neg end\, W (end \wedge X(\neg start\, W status\_valid))))$$

A counterexample for the above formula may look like the computation path π shown in Fig. 1.

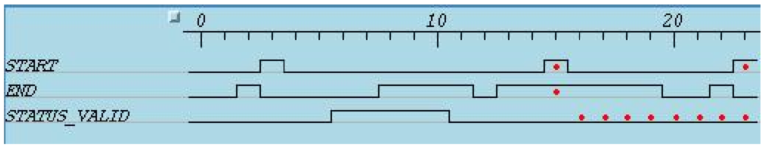


Fig. 1. A counterexample with explanations

Matching the formula with the counterexample is not trivial in this example. Our explanations, displayed as red dots, immediately attracts the user's attention to the relevant places, to ease the matching process. Note that each red dot r (indicating a pair $\langle$location, signal$\rangle$) is a *cause* of the failure of φ on the trace: switching the value of r would, under some contingency, change the value of φ on π.

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 [EL01]). We then present a model-checking-based approximation algorithm whose complexity is linear in the size of the formula and in the length of the trace. Experience show that our algorithm produces an exact causal set for most practical examples.

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 [CHK08], in which causality and its quantitative measure, responsibility (see [CH04] for the definition of responsibility), are viewed as refinement of coverage in model checking. Another work considers responsibility as a refinement of vacuity [CS07]. Finally, causality and responsibility can be used to improve the refinement techniques of symbolic trajectory evaluation (STE) [CGY08].

The rest of the paper is organized as follows. In Section 2 we give the definition of the Linear Temporal Logic LTL, and define causality in binary causal models. 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 approximation algorithm to compute a causal set. In Section 4 we discuss the implementation of our algorithm, on top of the IBM's commercial model checker *RuleBase* [Rul]. We demonstrate the graphical presentation used in practice, and report on experimental results demonstrating the usefulness of the method. Section 5 concludes the paper.

2 Preliminaries

Linear Temporal Logic

Formulas of LTL are built from a set V of Boolean variables using Boolean operators and the temporal operators:

– Every Boolean variable is an LTL formula.

– If φ and ψ are LTL formulas then so are:
  • $\neg\varphi$ • $\varphi \wedge \psi$ • $X\varphi$ • $[\varphi U \psi]$

Additional operators are defined as syntactic sugaring of those above:
• $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$ • $\varphi \rightarrow \psi = \neg\varphi \vee \psi$ • $F\varphi = [true\, U\varphi]$
• $G\varphi = \neg F \neg\varphi$ • $[\varphi W \psi] = [\varphi U \psi] \vee G\varphi$

The semantics of an LTL formula are traditionally defined with respect to an infinite computation path. A computation path is a sequence of states $w = s_0, s_1, s_2, ...,$, where s_i is a subset of the set of Boolean variables V, denoting the set of variables that hold in the state. The suffix of a computation path w, $s_j, s_{j+1}, s_{j+2}, ...$ is denoted by w^j. The finite prefix $s_0, s_1, ..., s_i$ of w is denoted w_i. The concatenation of a finite prefix w with an infinite computation v is denoted $w \cdot v$. We use $w \models \varphi$ to indicate that the LTL formula φ holds on the computation w. The semantics of $\models$ is inductively defined as follows.

– $w \models v$ iff $v \in w^0$
– $w \models \neg\varphi$ iff $w \not\models \varphi$
– $w \models \varphi \wedge \psi$ iff $w \models \varphi$ and $w \models \psi$
– $w \models X\varphi$ iff $w^1 \models \varphi$
– $w \models [\varphi U \psi]$ iff $\exists k \geq 0$ such that $w^k \models \psi$, and for every $0 < j < k, w^j \models \varphi$.

Causality

In this section, we review the definition of causality from [HP01]. As we argue below, models in formal verification are binary, thus we only present the significantly simpler versions of causality and responsibility for binary models (see [EL01] for the simplification of the definition of causality for the binary case). We also omit several other aspects of the general definition including the division of variables to exogenous and endogenous. Readers interested in the general framework of causality are referred to Appendix A.

Definition 2 (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 setting 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* φ is a boolean formula over the set of variables $\mathcal{V}$. A causal formula φ is true or false in a causal model given a context. We write $(M, \vec{u}) \models \varphi$ if φ 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 [CHK08] the definition of *criticality* in binary causal models, which captures the notion of counter-factual causal dependence.

Definition 3 (Critical variable) *[CHK08] Let M be a model, $\vec{u}$ the current context, and φ Boolean formula. Let $(M, \vec{u}) \models \varphi$, 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 φ in $(M, \vec{u})$ iff $(M, \vec{u}) \models (X \leftarrow \neg x)\neg\varphi$. That is, changing the value of X to $\neg x$ falsifies φ in $(M, \vec{u})$.*

With these definitions in hand, we can give the definition of cause in binary causal models from [HP01,EL01].

Definition 4 (Cause) *We say that $X = x$ is a* cause *of φ in $(M, \vec{u})$ if the following conditions hold:*

AC1. $(M, \vec{u}) \models (X = x) \wedge \varphi$.
AC2. *There exist a subset $\vec{W}$ of $\mathcal{V}$ with $X \notin \vec{W}$ and some setting $(x', \vec{w}')$ of the variables in $(X, \vec{W})$ such that setting the variables in $\vec{W}$ to the values $\vec{w}'$ makes $(X = x)$ critical for the satisfaction of φ.*

3 Causality in Counterexamples

In this section we show how thinking in terms of causality is useful for explaining counterexamples. In Section 3.1 we define the causal model for counterexamples and discuss the complexity of computing causality. In Section 3.2 we describe an approximate algorithm for explaining counterexamples that is based on causality and study its complexity.

3.1 Model

A *counterexample* to a formula φ in a model K is a computation path $\pi = s_0, s_1, \ldots$, where s_i is a state of K for all $i \geq 0$, such that $\pi \not\models \varphi$. The labeling function L of K maps each pair $\langle s_i, v\rangle$ of π, for a state s_i and a Boolean signal v, to $\{0, 1\}$ in a natural way: $\langle s_i, v\rangle = 1$ iff s_i is labeled with v, and 0 otherwise.

We view the counterexample trace π and the formula φ as a binary causal model. The set of pairs $\langle s_i, v\rangle$ is the set of variables $\mathcal{V}$ in the causal model M, and $\vec{u}$ is defined by L (since the range of L is binary, our model is indeed binary).

We note that there are two important differences between the definition of causality in Section 2 and our setting:

1. The value of φ. Since π is a counterexample, the value of φ in our model is initially **false**. Thus, in our setting, we are looking for *causes for falsification*, not causes for satisfaction.
2. The formula φ is in LTL, and not a Boolean formula. However, based on the automata-theoretic approach to branching-time model checking [KVW00], model checking can be viewed as evaluating a Boolean circuit with the values of the pairs $\langle s_i, v\rangle$ being the inputs to the circuit, and the output being the value of φ on the given trace. As shown in [CHK08], Boolean circuits are a special case of binary causal models, where each gate of the circuit is a variable of the model, and values

of inner gates are computed based on the values of the inputs to the circuit and the Boolean functions of the gates. A context $\vec{u}$ is a setting to the input variables of the circuit.

There is one subtlety, which we need to take care of before we formally define causes in counterexamples: the value of φ on finite paths. It is important to note that while computation paths are infinite, it is often sufficient to determine that $\pi \not\models \varphi$ after a finite prefix of the path. Thus a counterexample produced by a model checker may be a finite execution path. If π is finite, we use the following definition for the value of φ on π.

Definition 5 *Let π be a finite path and φ an LTL formula. We say that:*

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

Remark 6 *We note that our definition of satisfiability on finite paths coincides with the definition of truncated semantics for linear temporal logic in [EFH$^+$03], specifically the* abort *operator. It was shown in [ABKV03] that translating linear temporal logic formulas with the* abort *operator to B¨uchi automata has nonelementary complexity, in constrast to the* reset *operator, which differs from the* abort *slightly but enjoys the "fast-compilation property": adding* reset *to the linear temporal logic formulas does not increase the (single exponential) complexity of translation to B¨uchi automata. Essentially, the semantics introduced in [ABKV03] states that the value on a truncated path is unknown (or weakly satisfied) if we didn't see a failure yet. However, the only case where the definition in [EFH$^+$03] differs from the definition in [ABKV03] is when the specification in unsatisfiable. Since we assume that our specifications are satisfiable to start with, we do not have the comlexity problem mentioned in [ABKV03].*

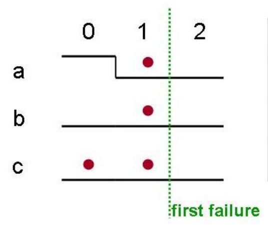
After the current context $\vec{u}$ is changed, the original value **false** of φ can change to either **true** or **unknown**. With these observations in hand, we can give the definition of cause for counterexamples.

Definition 7 (Cause in counterexample trace) *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). Then, a pair $\langle s, v \rangle$ is a* cause *of the failure of φ in $\pi[0..k]$ if it is a* cause *(as in Definition 4) of $\neg\varphi$ in the binary causal model that corresponds to $\pi[0..k]$ and φ.*

In other words, $\langle s, v \rangle$ is a *cause* of the failure of φ in $\pi[0..k]$ if there exists a set of pairs $S = \{\langle s_i, r \rangle : i \leq k, r \text{ is a signal in the labeling of } \pi\}$ such that changing the labeling function L for all pairs in S makes $\langle s, v \rangle$ *critical* for the falsification of φ in $\pi[0..k]$ (note that the change in the value of φ can be either to **true** or to **unknown**).

Examples 8 1. As an example of failure of the specification on a finite prefix, consider $\varphi_1 = \mathsf{G}p$ and a path $\pi_1 = s_0, s_1, s_2, s_3, (s_4)^\omega$ labeled as $(p)\cdot(p)\cdot(\neg p)\cdot(\neg p)\cdot(p)^\omega$. Clearly, φ_1 fails on a finite prefix of π_1. The shortest such prefix is $\pi_1[0..2]$. It is easy to see that according to our definition, $\langle s_2, p\rangle$ is a cause of failure of φ_1 in π_1, while $\langle s_3, p\rangle$ is not a cause. It is also easy to see that $\langle s_2, p\rangle$ is the only cause of failure of φ_1 in π_1, which indeed meets out intuition.

2. As an example of failure of the specification on an infinite path only, consider $\varphi_2 = \mathsf{F}p$ and a path $\pi_2 = (s_0)^\omega = (\neg p)^\omega$. Clearly, φ_2 fails in π_2, yet it does not fail on any finite prefix of π_2. According to our definition, the set of pairs $\{\langle s_i, p\rangle : i \in \mathbb{N}\}$ is the set of causes for failure of φ_2 on π_2.
3. A slightly more complicated example is as follows. Consider $\varphi_3 = aU(bUc)$ and a trace $\pi_3 = s_0, s_1, s_2, \ldots$ labeled as $a \cdot (\emptyset)^\omega$ (the prefix of φ_3 of length 3 is pictured in Figure 2). Clearly, φ_3 fails on π_3. Moreover, it is easy to see that the prefix in Figure 2 is the shortest prefix on which φ_3 fails. Indeed, a possible continuation of $\pi_3[0]$ on which φ_3 is satisfied can be a trace in which c is up in the first position, so φ_3 does not fail on $\pi_3[0]$. What is the set of causes for failure of φ_3 on $\pi_3[0..1]$? Clearly, $\langle s_0, a\rangle$ is not a cause, since φ_3 is monotonic increasing in the value of a, and thus changing the value of a from 1 to 0 cannot contribute to the satisfaction of φ_3 on π_3. By checking all possible changes of the current setting we can also see that $\langle s_0, b\rangle$ is not a cause. On the other hand, $\langle s_1, a\rangle$ is a cause for the falsification of φ_3 on $\pi_3[0..1]$ because if we change the value of a in s_1 from 0 to 1, the value of φ_3 on $\pi_3[0..1]$ becomes **unknown**. Changing the value of b in s_1 from 0 to 1 has the same effect, thus $zugs_1, b$ is also a cause. 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_3 to **true** on $\pi[0..1]$. The values of signals in s_2 are not causes because the first failure of φ_3 happens in s_1. The causes are represented graphically as red dots in Figure 2.

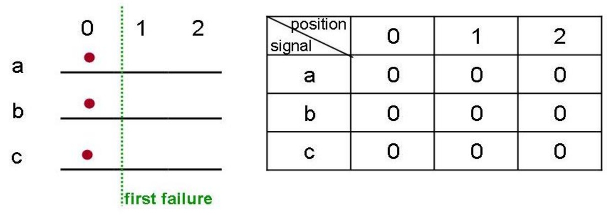


signal \ position	0	1	2
a	1	0	0
b	0	0	0
c	0	0	0

Fig. 2. A counterexample trace for $aU(bUc)$.

4. The following example demonstrates the difference between criticality and causality. Consider $\varphi_4 = G(a \wedge b \wedge c)$ and a trace $\pi_4 = s_0, s_1, s_2, \ldots$ labeled as $(\emptyset)^\omega$ (the prefix of φ_4 of length 3 is pictured in Figure 3). Clearly, φ_4 fails on $\pi_4[0]$, however, changing the value of any signal in one state does not change the value of φ_4. There exists, however, the change of the setting that makes the value of a in s_0 critical for

the value of φ_4 in $\pi_4[0]$: the change of the value of b and the value of x in s_0 from 0 to 1. Similarly, there exists a change of the setting that makes the value of a and the value of c (separately) in s_0 critical for the value of φ_4 in $\pi_4[0]$. Thus, all three of them are causes and are marked by red dots in Figure 3.



position / signal	0	1	2
a	0	0	0
b	0	0	0
c	0	0	0

Fig. 3. A counterexample trace for $G(a \wedge b \wedge c)$.

Why do we need to consider the cases where there is a finite prefix on which the specification fails separately? Note that a trace π may demonstrate more than one point of failure, as in our example above. We state that the first failure is usually the most interesting one to 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.

3.2 Complexity and algorithms

Eiter and Lukasiewicz showed that in binary causal models, computing causality is NP-complete [EL01]. The reduction from binary causal models to Boolean circuits and from Boolean circuits to model-checking, shown in [CHK08], proves that computing causality in model checking of branching time specifications is NP-complete as well. Our setting differs from the setting in [CHK08] in two aspects: our specifications are in linear-time temporal logic, and we compute causes for falsification, rather than satisfaction. Since on a single trace linear-time and branching temporal logics coincide, and since computing the causes for satisfaction is easily reducible to computing the causes for falsification, we have the following lemma.

Lemma 9. *Computing the set of causes for falsification of a linear-time temporal specification on a single trace is NP-complete.*

Since traces are usually short and specifications are small, applying the brute-force algorithm (which amounts to checking the effect of changing the value of every subset of literals on the criticality of each pair $\langle$ state, signal $\rangle$) for finding the exact set of causes is not unreasonable. On the other hand, our tool is interactive, and thus is supposed to perform the computation of causes while the user is waiting. Thus, it is important to make the algorithm as efficient as possible.

We start with presenting a reduction of the causality problem to SAT. Since there is currently a plethora of efficient SAT solvers (in particular, IBM's model checker Rule-Base has a very powerful SAT solver Mage built into it), such a translation can lead to an efficient implementation of causality computation. Then, we describe an approximation algorithm for computing causality, with the linear complexity both in the size of the specification and the size of the trace.

Reducing causality to SAT [**sketch**] The path π is a counterexample to φ, thus the values of signals in states of φ represent the satisfying assignment to the SAT formula derived from the product of the whole model with $\neg\varphi$ [BCCZ99]. Since we are looking for modifications of π in orded to compute causality, we do the following: (1) Let $\langle s_i, p\rangle$ be the pair for which we compute whether it is a cause for the falsification of φ in $\pi[0..k]$. We replace all other values of signals in states with variables, and denote the resulting formula by $\Psi_{\langle i,p\rangle}(\neg\varphi, \pi[0..k])$. (2) We flip the value of $\langle s_i, p\rangle$, and denote the formula that is the product of φ and the trace $\pi[0..k]$ with the value of $\langle s_i, p\rangle$ flipped and the other signals replaced with variables by $\Psi_{\neg\langle i,p\rangle}(\varphi, \pi[0..k])$. Then, $\langle i, p\rangle$ is a cause for falsification of φ in $\pi[0..k]$ iff the formula $\Psi_{\langle i,p\rangle}(\neg\varphi, \pi[0..k]) \wedge \Psi_{\neg\langle i,p\rangle}(\varphi, \pi[0..k])$ is satisfiable. Indeed, it is satisfiable iff there exist values of signals in states other than $\langle s_i, p\rangle$ such that the trace with these values and the original value of $\langle s_i, p\rangle$ still falsifies φ, and flipping the value of $\langle s_i, p\rangle$ causes satisfaction of φ in $\pi[0..k]$. [** how to express unknown? also, need to add the truncated semantics to BCCZ99 **]

Approximation algorithm We note that while the definitions provided in previous sections are good for both finite and for infinite computations, the procedure given below considers finite computations only. This is because in practice, the counterexamples we work with are always finite. When representing an infinite path, the counterexample will contain a "loop" indication.

The procedure below produces $C(w, \varphi)$, the approximation of the set of causes for the failure of φ on w. We sometimes use $v(w, \varphi)$ to denote the valuation of φ on w (that is, $v(w, \varphi) = 1$ iff $w \models \varphi$). The function *Time*, used when a proposition is evaluated, returns the time unit we are currently checking. Note that when the recursive procedure gets to the proposition level, the computation is in many cases shorter than the original computation with which it started. This is because the X and U operators recursively apply the algorithm on w^1. Let the length of the top level computation be n, and let w be the computation at the proposition level. Then *Time*(w) = $n - |w|$.

Algorithm 10 (A Recursive Procedure) *A causality set C for φ and π can be computed as follows.*

- $C(\pi, true) = C(\pi, false) = \emptyset$
- $C(\pi, p) = \begin{cases} \{(Time(\pi), p)\} & \textit{if } v(s_0, p) = 1 \\ \emptyset & \textit{otherwise} \end{cases}$
- $C(\pi, \neg p) = \begin{cases} \{(Time(\pi), p)\} & \textit{if } v(s_0, p) = 0 \\ \emptyset & \textit{otherwise} \end{cases}$
- $C(\pi, X\varphi) = \begin{cases} C(\pi^1, \varphi) & \textit{if } |\pi| > 1 \\ \emptyset & \textit{otherwise} \end{cases}$

– $C(\pi, \varphi \wedge \psi) =$
$$\begin{cases} C(\pi,\varphi) \cup C(\pi,\psi) & \textit{if } \pi \not\models \varphi \textit{ and } \pi \not\models \psi \\ C(\pi,\varphi) & \textit{if } \pi \not\models \varphi \textit{ and } \pi \models \psi \\ C(\pi,\psi) & \textit{if } \pi \models \varphi \textit{ and } \pi \not\models \psi \\ \emptyset & \textit{otherwise} \end{cases}$$
– $C(\pi, \varphi \vee \psi) =$
$$\begin{cases} C(\pi,\varphi) \cup C(\pi,\psi) & \textit{if } \pi \not\models \varphi \textit{ and } \pi \not\models \psi \\ \emptyset & \textit{otherwise} \end{cases}$$
– $C(\pi, [\varphi U \psi]) =$
$$\begin{cases} C(\pi,\psi) \cup C(\pi,\varphi) & \textit{if } \pi \not\models \varphi \textit{ and } \pi \not\models \psi \\ C(\pi,\psi) \cup C(\pi^1,[\varphi U \psi]) & \textit{if } \pi \models \varphi \textit{ and } \pi \not\models \psi \\ & \quad \textit{and } \pi \not\models X[\varphi U \psi] \textit{ and } |\pi| > 1 \\ \emptyset & \textit{otherwise} \end{cases}$$

The procedure above recursively performs model checking of the given formula φ on the counterexample π. At the proposition level, p is considered a cause in the current state if and only if it has the erroneous value in the current 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 is falsified at the current state. Note that for conjunctions, if both conjuncts are relevant, only one conjunct is chosen. [** really? then we don't always find the shortest path to faliure! **] We explain in detail the recursive definition of the *Until* operator, since it is the most difficult to follow.

The formula $\eta = [\varphi U \psi]$ is relevant on π in two cases.

1. If $\pi \models \eta$.
   The trace π can satisfy η in one of the following ways.
   (a) $\pi \models \psi$. The relevance set of η would be the relevance set of ψ.
   (b) $\pi \models \varphi \wedge w \models X[\varphi U \psi]$. The relevance set for this case would be the union of the relevance set for φ and the relevance set for $X[\varphi U \psi]$.
2. If $\pi \not\models \eta$, there can be one of the three reasons to that:
   (a) $\pi \not\models \varphi \wedge \pi \not\models \psi$, in which case, the relevance set would be the union of the relevance sets for φ and for ψ.
   (b) $\pi \models \varphi \wedge \pi \not\models \psi \wedge |\pi| = 1$, in which case the relevance set would be the set for ψ.
   (c) $\pi \models \varphi \wedge \pi \not\models \psi \wedge \pi \not\models X[\varphi U \psi]$. The relevance set would be the union of the sets for ψ and for $X[\varphi U \psi]$.

It is easy to see that the computation of the approximate set of causes and the first point of failure of φ on π is done in one path on π, and the satisfaction of φ is checked locally at each state. Thus, we have the following lemmas.

Lemma 11. *The Algorithm 10 returns the shortest prefix of π on which φ fails.*

Lemma 12. *The complexity of Algorithm 10 is linear in k and in $|\varphi|$.*

4 Implementation and Experimental Results

The idea of explaining counterexamples arose from a genuine need encountered by RuleBase PE users. A RuleBase PE user is typically a verification engineer, who is formally verifiying a hardware design written by a logic designer. The verification engineer writes temporal formulas and runs them in RuleBase PE, selecting one or more model checking engines with which to check each formula. Each engine uses a different model checking algorithm, as described in [Rul]. If a formula fails on the design-under-test (DUT), the model checking engine produces a counterexample trace, which the verification engineer views in RuleBase PE's built-in trace viewer.

When the user views a counterexample trace for the first time, her purpose is not trying to debug the hardware design. Although the verification engineer understands the specification of the DUT, she is probably not familiar with the detailed implementation, which is owned by the logic designer. What the user is looking for is some very 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.

For answering these questions, the user restricts her view of the trace to the signals that appear in the formula itself. For example, if the formula is $G\ \neg\texttt{ERROR}$, then the user only views the behavior of the signal `ERROR` in the trace viewer. For this particular formula, it is then relatively easy to visually scan the trace and find a time point at which `ERROR` holds (although even this simple task may be difficult if the trace is very long). However, the boolean invariant in the formula may be more complex, involving multiple signals and boolean operations between them, in which case the visual scan becomes non-trivial (see Example 13 below). Adding other temporal operators to the formula, such as X or U, makes the visual scan even more difficult, since relations between several trace cycles must be considered.

This is the point at which visual trace explanation becomes helpful. In the RuleBase PE trace viewer, the trace is displayed as usual, but with the addition of small red dots at several $\langle signal, cycle\rangle$ locations on the trace. This shows the user which points in the trace are relevant for the fail, allowing her to focus on these points and ignore other parts of the trace.

These red dots will also accompany the trace when it becomes part of a bug report sent to the logic designer. The designer will also use them as a starting point for actually debugging the fail.

Implementation in RuleBase PE

The algorithm used in practice by RuleBase PE for computing the displayed set of "red dots" is a variant of the recursive procedure presented in Algorithm 10. This procedure is applied to every counterexample trace produced by the model checking engines. The procedure receives the trace and formula as input, and is oblivious to the method by which the trace was obtained. The output of the procedure is a set of $\langle signal, cycle\rangle$ pairs, which is passed on to the trace viewer for displaying the red dots at the designated locations. The execution time of the red dots computation is negligible, and is not felt by the user.

In addition to counterexample traces for fails, RuleBase PE provides *witness traces* for formulas that pass [XXX citation needed]. Red dots are displayed on these traces as well. In practice, a witness trace is simply a counterexample trace for a witness formula, and therefore the procedure for computing red dots on a witness trace is the same as for a counterexample trace.

Examples

We implemented Algorithm 10 in RuleBase PE, replacing the similar algorithm that is normally used in the tool. We used this implementation to obtain the red dots shown in the following examples.

Example 13 (A boolean invariant) *As an example of a complex boolean invariant, consider the formula*

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

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 failure at cycle 12.

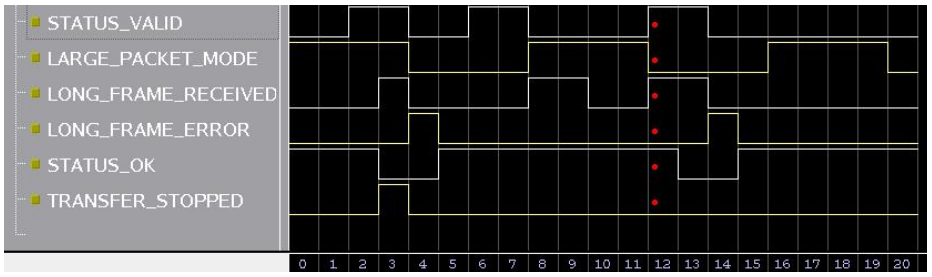


Fig. 4. Counterexample for a boolean invariant

Example 14 (A liveness property) *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. The point at which the loop begins is marked with a red dot (in addition to the red dots computed by Algorithm 10). For example, the trace in Figure 5 is a counterexample for the formula*

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

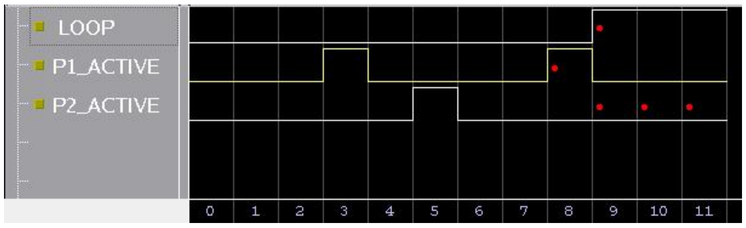


Fig. 5. Counterexample for a liveness property

Example 15 *We revisit the formula $aU(bUc)$ from Example 8, to compare the causality set computed by Algorithm 10 with the causality set derived from Definition 7. We know that the algorithm cannot compute the exact set in all cases, and indeed in this case we see a difference. Figure 6 shows the red dots computed by Algorithm 10. The algorithm places a red dot on signal b at cycle 0, which is not a cause of the failure according to Definition 7.*

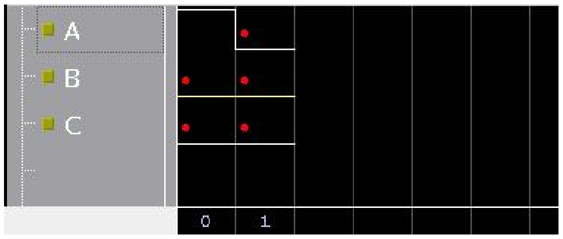


Fig. 6. Counterexample for $aU(bUc)$

5 Conclusion and Future Directions

We have shown how the causality definition of Halpern and Pearl [HP01] can be applied to the task of explaining a counterexample. Our method is implemented as part of the IBM model checking tool-set RuleBase [Rul], 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 at a time, no size issues are involved. 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 [BDEGW03], 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, it is interesting to see whether there exist subsets of LTL for which the computation of the exact set of causes is polynomial. One of the natural candidates for being such an "easy" sublogic of LTL is the PSL simple subset defined in [EF06]. Another natural candidate could be the common fragment of LTL and ACTL, called LTL$^{\mathsf{d}et}$ (see [Mai00]), however, we were not able to come up with a polynomial-time algorithm for computing the set of causes for this logic yet. The main reason for that is that, roughly speaking, it seems that we need determinism in falsification rather than in satisfaction of the specification.

Another possible direction for a future research is to try to come up with efficient implementations of the brute-force algorithm for computing the exact set of causes. A promising direction is to reduce this problem to a satisfiability problem, due to the abundance of efficient SAT solvers.

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 the full PSL without significantly increasing its complexity.

References

[ABKV03] R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi. Resets vs. aborts in linear temporal logic. In *TACAS*, pages 65–80, 2003.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[BDEGW03] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22(2):101–108, 2003.

[BNR03] T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages*, pages 97–105, 2003.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CG05] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. In *Proceedings of FASE'05*, pages 217–233, 2005.

[CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automation Conference*, pages 427–432. IEEE Computer Society, 1995.

[CGP00] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[CGY08] Hana Chockler, Orna Grumberg, and Avi Yadgar. Efficient automatic ste refinement using responsibility. In *TACAS*, pages 233–248, 2008.

[CH04] H. Chockler and J.Y. Halpern. Responsibility and blame: a structural-model approach. *Journal of Artificial Intelligence Research (JAIR)*, 22:93–115, 2004.

[CHK08] H. Chockler, J. Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification&quest;. *ACM Trans. Comput. Log.*, 9(3), 2008.

[CIW$^+$01] F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In *In Proceedings of CHARME'01*, pages 275–292, 2001.

[CS07] H. Chockler and O. Strichman. Easier and more informative vacuity checks. In *MEMOCODE*, pages 189–198. IEEE, 2007.

[DRS03] Y. Dong, C. R. Ramakrishnan, and S. A. Smolka. Model checking and evidence exploration. In *IEEE Conference and Workshops on Engineering Computer Based Systems*, pages 214–223, Huntsville, Alabama, 2003. IEEE.

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL.* Series on Integrated Circuits and Systems. Springer-Verlag, 2006.

[EFH$^+$03] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, pages 27–39, 2003.

[EL01] T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. In *Proc. 7th International Joint Conference on Artificial Intelligence*, pages 35–40, 2001.

[GK04] A. Groce and D. Kroening. Making the most of bmc counterexamples. In *SGSH*, July 2004.

[Gro04] A. Groce. Error explanation with distance metrics. In *TACAS*, 2004.

[GSB07] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for c programs. *Electr. Notes Theor. Comput. Sci.*, 174(4):95–111, 2007.

[Hal02] N. Hall. Two concepts of causation. In J. Collins, N. Hall, and L. A. Paul, editors, *Causation and Counterfactuals*. MIT Press, Cambridge, Mass., 2002.

[HP01] J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach — part 1: Causes. In *Uncertainty in Artificial Intelligence: Proceedings of the Seventeenth Conference (UAI-2001)*, pages 194–202, San Francisco, CA, 2001. Morgan Kaufmann Publishers.

[Hum39] D. Hume. *A treatise of human nature*. John Noon, London, 1939.

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS02*, pages 445–458, 2002.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[Mai00] Monika Maidl. The common fragment of CTL and LTL. In *IEEE Symposium on Foundations of Computer Science*, pages 643–652, 2000.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rul] Rulebase homepage. $http://www.haifa.il.ibm.com/projects/verification/RB\_Homepage.$

[SB07] S. Staber and R. Bloem. Fault localization and correction with qbf. In *SAT*, pages 355–368, 2007.

[SFBD08] A. Sülflow, G. Fey, R. Bloem, and R. Drechsler. Using unsatisfiable cores to debug multiple design errors. In *ACM Great Lakes Symposium on VLSI*, pages 77–82, 2008.

[SQL05] S. Shen, Y. Qin, and S. Li. A faster counterexample minization algorithm based on refutation analysis. In *DATE05*, pages 672–677, 2005.

[WYIG06] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *ATVA*, pages 82–95, 2006.

A The General Framework of Causality

In this section, we review the details of the definitions of causality for general recursive causal models from [HP01].

A *signature* is a tuple $\mathcal{S} = \langle \mathcal{U}, \mathcal{V}, \mathcal{R} \rangle$, where $\mathcal{U}$ is a finite set of *exogenous* variables, $\mathcal{V}$ is a set of *endogenous* variables, and the function $\mathcal{R} : \mathcal{U} \cup \mathcal{V} \to \mathcal{D}$ associates with every variable $Y \in \mathcal{U} \cup \mathcal{V}$ a nonempty set $\mathcal{R}(Y)$ of possible values for Y from the range $\mathcal{D}$. Intuitively, the exogenous variables are ones whose values are determined by factors outside the model, while the endogenous variables are ones whose values are ultimately determined by the exogenous variables. A *causal model* over signature $\mathcal{S}$ is a tuple $M = \langle \mathcal{S}, \mathcal{F} \rangle$, where $\mathcal{F}$ associates with every endogenous variable $X \in \mathcal{V}$ a function F_X such that $F_X : (\times_{U \in \mathcal{U}} \mathcal{R}(U)) \times (\times_{Y \in \mathcal{V} \setminus \{X\}} \mathcal{R}(Y)) \to \mathcal{R}(X)$. That is, F_X describes how the value of the endogenous variable X is determined by the values of all other variables in $\mathcal{U} \cup \mathcal{V}$. If the range $\mathcal{D}$ contains only two values, we say that M is a *binary causal model*.

We can describe (some salient features of) a causal model M using a *causal network*. This is a graph with nodes corresponding to the random 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. Intuitively, variables can have a causal effect only on their descendants in the causal network; if Y is not a descendant of X, then a change in the value of X has no affect on the value of Y. For ease of exposition, we restrict attention to what are called *recursive* models. These are ones whose associated causal network is a directed acyclic graph (that is, a graph that has no cycle of edges). It should be clear that if M is a recursive causal model, then there is always a unique solution to the equations in M, given a *context*, that is, a setting $\vec{u}$ for the variables in $\mathcal{U}$.

The equations determined by $\{F_X : X \in \mathcal{V}\}$ can be thought of as representing processes (or mechanisms) by which values are assigned to variables. For example, if $F_X(Y, Z, U) = Y + U$ (which we usually write as $X = Y + U$), then if $Y = 3$ and $U = 2$, then $X = 5$, regardless of how Z is set. This equation also gives counterfactual information. It says that, in the context $U = 4$, if Y were 4, then X would be $u + 4$, regardless of what value X, Y, and Z actually take in the real world.

While the equations for a given problem are typically obvious, the choice of variables may not be. For example, consider the rock-throwing example from the introduction. In this case, a naive model might have an exogenous variable U that encapsulates whatever background factors cause Suzy and Billy to decide to throw the rock (the details of U do not matter, since we are interested only in the context where U's value is such that both Suzy and Billy throw), a variable *ST* for Suzy throws (*ST* $= 1$ if Suzy throws, and *ST* $= 0$ if she doesn't), a variable *BT* for Billy throws, and a variable BS for bottle shatters. In the naive model, BS is 1 if one of *ST* and *BT* is 1.

This causal model does not distinguish between Suzy and Billy's rocks hitting the bottle simultaneously and Suzy's rock hitting first. A more sophisticated model is the one that takes into account the fact that Suzy throws first. It might also include variables *SH* and *BH*, for Suzy's rock hits the bottle and Billy's rock hits the bottle. Clearly BS is 1 iff one of *BH* and *BT* is 1. However, now, *SH* is 1 if *ST* is 1, and *BH* $= 1$ if *BT* $= 1$ and *SH* $= 0$. Thus, Billy's throw hits if Billy throws *and* Suzy's rock doesn't hit.

Given a causal model $M = (\mathcal{S}, \mathcal{F})$, a (possibly empty) vector $\vec{X}$ of variables in $\mathcal{V}$, and vectors $\vec{x}$ and $\vec{u}$ of values for the variables in $\vec{X}$ and $\mathcal{U}$, respectively, we can define a new causal model denoted $M_{\vec{X} \leftarrow \vec{x}}$ over the signature $\mathcal{S}_{\vec{X}} = (\mathcal{U}, \mathcal{V} - \vec{X}, \mathcal{R}|_{\mathcal{V}-\vec{X}})$. Formally, $M_{\vec{X} \leftarrow \vec{x}} = (\mathcal{S}_{\vec{X}}, \mathcal{F}^{\vec{X} \leftarrow \vec{x}})$, where $F_Y^{\vec{X} \leftarrow \vec{x}}$ is obtained from F_Y by setting the values of the variables in $\vec{X}$ to $\vec{x}$. Intuitively, this is the causal model that results when the variables in $\vec{X}$ are set to $\vec{x}$ by some external action that affects only the variables in $\vec{X}$; we do not model the action or its causes explicitly. For example, if M is the more sophisticated model for the rock-throwing example, then $M_{ST \leftarrow 0}$ is the model where Suzy doesn't throw.

Given a signature $\mathcal{S} = (\mathcal{U}, \mathcal{V}, \mathcal{R})$, a formula of the form $X = x$, for $X \in V$ and $x \in \mathcal{R}(X)$, is called a *primitive event*. A *basic causal formula* is one of the form $[Y_1 \leftarrow y_1, \ldots, Y_k \leftarrow y_k]\phi$, where ϕ is a Boolean combination of primitive events; $Y_1, \ldots, Y_k$ are distinct variables in $\mathcal{V}$; and $y_i \in \mathcal{R}(Y_i)$. Such a formula is abbreviated as $[\vec{Y} \leftarrow \vec{y}]\phi$. The special case where $k = 0$ is abbreviated as ϕ. Intuitively, $[Y_1 \leftarrow y_1, \ldots, Y_k \leftarrow y_k]\phi$ says that ϕ holds in the counterfactual world that would arise if Y_i is set to y_i, $i = 1, \ldots, k$. A *causal formula* is a Boolean combination of basic causal formulas.

A causal formula ϕ is true or false in a causal model, given a *context*. We write $(M, \vec{u}) \models \phi$ if ϕ is true in causal model M given context $\vec{u}$. $(M, \vec{u}) \models [\vec{Y} \leftarrow \vec{y}](X = x)$ if the variable X has value x in the unique (since we are dealing with recursive models) solution to the equations in $M_{\vec{Y} \leftarrow \vec{y}}$ in context $\vec{u}$ (that is, the unique vector of values for the exogenous variables that simultaneously satisfies all equations $F_Z^{\vec{Y} \leftarrow \vec{y}}$, $Z \in \mathcal{V} - \vec{Y}$, with the variables in $\mathcal{U}$ set to $\vec{u}$). We extend the definition to arbitrary causal formulas in the obvious way.

With these definitions in hand, we can give the definition of cause from [HP01].

Definition 1. *We say that $\vec{X} = \vec{x}$ is a* cause *of φ in $(M, \vec{u})$ if the following three conditions hold:*

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

AC2. *There exist a partition $(\vec{Z}, \vec{W})$ of $\mathcal{V}$ with $\vec{X} \subseteq \vec{Z}$ and some setting $(\vec{x}', \vec{w}')$ of the variables in $(\vec{X}, \vec{W})$ such that if $(M, \vec{u}) \models Z = z^*$ for $Z \in \vec{Z}$, then*

(a) $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}']\neg\varphi$. That is, changing $(\vec{X}, \vec{W})$ from $(\vec{x}, \vec{w})$ to $(\vec{x}', \vec{w}')$ changes φ from **true** *to* **false**.

(b) $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}, \vec{W} \leftarrow \vec{w}', \vec{Z}' \leftarrow \vec{z}^*]\varphi$ for all subsets $\vec{Z}'$ of $\vec{Z}$. That is, setting $\vec{W}$ to $\vec{w}'$ should have no effect on φ as long as $\vec{X}$ has the value $\vec{x}$, even if all the variables in an arbitrary subset of $\vec{Z}$ are set to their original values in the context $\vec{u}$.

AC3. *$(\vec{X} = \vec{x})$ is minimal, that is, no subset of $\vec{X}$ satisfies AC2.*

AC1 just says that A cannot be a cause of B unless both A and B are true, while AC3 is a minimality condition to prevent, for example, Suzy throwing the rock and sneezing from being a cause of the bottle shattering. Eiter and Lukasiewicz [EL01] showed that one consequence of AC3 is that causes can always be taken to be single conjuncts. The core of this definition lies in AC2. Informally, the variables in $\vec{Z}$ should be thought of as describing the "active causal process" from $\vec{X}$ to ϕ. These are the variables that mediate between $\vec{X}$ and ϕ. AC2(a) is reminiscent of the traditional counterfactual criterion. However, AC2(a) is more permissive than the traditional criterion; it allows the dependence of ϕ on $\vec{X}$ to be tested under special *structural contingencies*, in which the variables $\vec{W}$ are held constant at some setting $\vec{w}'$. AC2(b) is an attempt to counteract the "permissiveness" of AC2(a) with regard to structural contingencies. Essentially, it ensures that $\vec{X}$ alone suffices to bring about the change from ϕ to $\neg\phi$; setting $\vec{W}$ to $\vec{w}'$ merely eliminates spurious side effects that tend to mask the action of $\vec{X}$.

To understand the role of AC2(b), consider the rock-throwing example again. Looking at the simple model, it is easy to see that both Suzy and Billy are causes of the bottle shattering. Taking $\vec{Z} = \{\mathit{ST}, BS\}$, consider the structural contingency where Billy doesn't throw ($\mathit{BT} = 0$). Clearly $[\mathit{ST} \leftarrow 0, \mathit{BT} \leftarrow 0]BS = 0$ and $[\mathit{ST} \leftarrow 1, \mathit{BT} \leftarrow 0]BS = 1$ both hold, so Suzy is a cause of the bottle shattering. A symmetric argument shows that Billy is also the cause.

But now consider the model that takes into account that Suzy throws first. It is still the case that Suzy is a cause in this model. We can take $\vec{Z} = \{\mathit{ST}, \mathit{SH}, BS\}$ and again consider the contingency where Billy doesn't throw. However, Billy is *not* a cause of the bottle shattering. For suppose that we now take $\vec{Z} = \{\mathit{BT}, \mathit{BH}, BS\}$ and consider the contingency where Suzy doesn't throw. Clearly AC2(a) holds, since if Billy doesn't throw (under this contingency), then the bottle doesn't shatter. However, AC2(b) does not hold. Since $\mathit{BH} \in \vec{Z}$, if we set *BH* to 0 (it's original value), then AC2(b) requires that $[\mathit{BT} \leftarrow 1, \mathit{ST} \leftarrow 0, \mathit{BH} \leftarrow 0](BS = 1)$ hold, but it does not. Similar arguments show that no other choice of $(\vec{Z}, \vec{W})$ makes Billy's throw a cause.

The causal model is represented graphically in Figure 7. Clearly BS is 1 iff one of *BH* and *BT* is 1. However, now, *SH* is 1 if *ST* is 1, and $\mathit{BH} = 1$ if $\mathit{BT} = 1$ and $\mathit{SH} = 0$. Thus, Billy's throw hits if Billy throws *and* Suzy's rock doesn't hit. In Figure 7 there is an arrow from variable X to variable Y if the value of Y depends on the value of X.

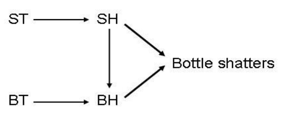


Fig. 7. The rock-throwing example.