Explaining Hyperproperty Violations

Norine Coenen[1](✉), Raimund Dachselt[2], Bernd Finkbeiner[1], Hadar Frenkel[1], Christopher Hahn[1], Tom Horak[3], Niklas Metzger[1], and Julian Siber[1]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
{norine.coenen,finkbeiner,hadar.frenkel,christopher.hahn, niklas.metzger,julian.siber}@cispa.de
[2] Interactive Media Lab, Technische Universität Dresden, Dresden, Germany
dachselt@acm.org
[3] elevait GmbH & Co. KG, Dresden, Germany
tom.horak@elevait.de

Abstract. Hyperproperties relate multiple computation traces to each other. Model checkers for hyperproperties thus return, in case a system model violates the specification, a set of traces as a counterexample. Fixing the erroneous relations between traces in the system that led to the counterexample is a difficult manual effort that highly benefits from additional explanations. In this paper, we present an explanation method for counterexamples to hyperproperties described in the specification logic HyperLTL. We extend Halpern and Pearl's definition of actual causality to sets of traces witnessing the violation of a HyperLTL formula, which allows us to identify the events that caused the violation. We report on the implementation of our method and show that it significantly improves on previous approaches for analyzing counterexamples returned by HyperLTL model checkers.

1 Introduction

While model checking algorithms and tools (e.g., [12,17,18,26,47,55]) have, in the past, focused on trace properties, recent failures in security-critical systems, such as Heartbleed [28], Meltdown [59], Spectre [52], or Log4j [1], have triggered the development of model checking algorithms for properties that relate multiple computation traces to each other, i.e., *hyperproperties* [21]. Although the counterexample returned by such a model checker for hyperproperties, which takes the shape of a *set* of traces, may aid in the debugging process, understanding and narrowing down which features are actually responsible for the erroneous

This work was funded by DFG grant 389792660 as part of TRR 248 – CPEC, by the DFG as part of the Germany's Excellence Strategy EXC 2050/1 - Project ID 390696704 - Cluster of Excellence "*Centre for Tactile Internet*" (CeTI) of TU Dresden, by the European Research Council (ERC) Grant OSARES (No. 683300), and by the German Israeli Foundation (GIF) Grant No. I-1513-407./2019.

© The Author(s) 2022
S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13371, pp. 407–429, 2022.
https://doi.org/10.1007/978-3-031-13185-1_20

relation between the traces in the counterexample requires significantly more manual effort than for trace properties. In this paper, we develop an explanation technique for these more complex counterexamples that identifies the *actual causes* [44–46] of hyperproperty violations.

Existing hyperproperty model checking approaches (e.g., [33,35,49]), take a HyperLTL formula as an input. HyperLTL is a temporal logic extending LTL with explicit trace quantification [20]. For example, observational determinism, which requires that all traces π, π' agree on their observable outputs lo whenever they agree on their observable inputs li, can be formalized in HyperLTL as $\forall \pi. \forall \pi'. \square(li_\pi \leftrightarrow li_{\pi'}) \rightarrow \square(lo_\pi \leftrightarrow lo_{\pi'})$. In case a system model violates observational determinism, the model checker consequently returns a set of two execution traces witnessing the violation.

A first attempt in explaining model checking results of HyperLTL specifications has been made with HyperVis [48], which visualizes a counterexample returned by the model checker MCHyper [35] in a browser application. While the visualizations are already useful to analyze the counterexample at hand, it fails to identify causes for the violation in several security-critical scenarios. This is because HyperVis identifies important atomic propositions that appear in the HyperLTL formula and highlights these in the trace and the formula. For detecting causes, however, this is insufficient: a cause for a violation of observational determinism, for example, could be a branch on the valuation of a secret input i_s, which is not even part of the formula (see Sect. 3 for a running example).

Defining what constitutes an actual cause for an effect (a violation) in a given scenario is a precious contribution by Halpern and Pearl [44–46], who refined and formalized earlier approaches based on counterfactual reasoning [58]: Causes are sets of events such that, in the counterfactual world where they do not appear, the effect does not occur either. One of the main insights of Halpern and Pearl's work, however, is that naive counterfactuals are too imprecise. If, for instance, our actual cause preempted another potential cause, the mere absence of the actual cause will not be enough to prevent the effect, which will be still produced by the other cause in the new scenario. Halpern and Pearl's definition therefore allows to carefully control for other possible causes through the notion of *contingencies*. In the modified definition [44], contingencies allow to fix certain features of the counterfactual world to be exactly as they are in the actual world, regardless of the system at hand. Such a contingency effectively modifies the dynamics of the underlying model, and one insight of our work is that defining actual causality for reactive systems also needs to modify the system under a contingency. Notably, most works regarding trace causality [13,39] do not consider contingencies but only counterfactuals, and thus are not able to find true actual causes.

In this paper, we develop the notion of actual causality for effects described by HyperLTL formulas and use the generated causes as explanations for counterexamples returned by a model checker. We show that an implementation of our algorithm is practically feasible and significantly increases the state-of-the-art in explaining and analyzing HyperLTL model checking results.

2 Preliminaries

We model a system as a *Moore machine* [62] $T = (S, s_0, AP, \delta, l)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $AP = I \cup O$ is the set of atomic propositions consisting of inputs I and outputs O, $\delta : S \times 2^I \to S$ is the transition function determining the successor state for a given state and set of inputs, and $l : S \to 2^O$ is the labeling function mapping each state to a set of outputs. A *trace* $t = t_0 t_1 t_2 \ldots \in (2^{AP})^\omega$ of T is an infinite sequence of sets of atomic propositions with $t_i = A \cup l(s_i)$, where $A \subseteq I$ and $\delta(s_i, A) = s_{i+1}$ for all $i \geq 0$. We usually write $t[n]$ to refer to the set t_n at the $(n+1)$-th position of t. With $\mathit{traces}(T)$, we denote the set of all traces of T. For some sequence of inputs $a = a_0 a_1 a_2 \ldots \in (2^I)^\omega$, the trace $T(a)$ is defined by $T(a)_i = a_i \cup l(s_i)$ and $\delta(s_i, a_i) = s_{i+1}$ for all $i \geq 0$. A trace property $P \subseteq T$ is a set of traces. A hyperproperty H is a lifting of a trace property, i.e., a *set of sets of traces*. A model T satisfies a hyperproperty H if the set of traces of T is an element of the hyperproperty, i.e., $\mathit{traces}(T) \in H$.

2.1 HyperLTL

HyperLTL is a recently introduced logic for expressing temporal hyperproperties, extending linear-time temporal logic (LTL) [64] with trace quantification:

$$\varphi ::= \forall \pi.\, \varphi \mid \exists \pi.\, \varphi \mid \ \psi$$
$$\psi ::= a_\pi \mid \neg \psi \mid \psi \wedge \psi \mid \bigcirc \psi \mid \psi \mathcal{U} \psi$$

We also consider the usual derived Boolean ($\vee$, $\to$, $\leftrightarrow$) and temporal operators ($\varphi \mathcal{R} \psi \equiv \neg(\neg\varphi \mathcal{U} \neg\psi)$, $\Diamond \varphi \equiv \mathit{true}\, \mathcal{U} \varphi$, $\Box \varphi \equiv \mathit{false} \mathcal{R} \varphi$). The semantics of HyperLTL formulas is defined with respect to a set of traces Tr and a trace assignment $\Pi : \mathcal{V} \to Tr$ that maps trace variables to traces. To update the trace assignment so that it maps trace variable π to trace t, we write $\Pi[\pi \mapsto t]$.

$\Pi, i \vDash_{Tr} a_\pi$	iff	$a \in \Pi(\pi)[i]$
$\Pi, i \vDash_{Tr} \neg\varphi$	iff	$\Pi, i \nvDash_{Tr} \varphi$
$\Pi, i \vDash_{Tr} \varphi \wedge \psi$	iff	$\Pi, i \vDash_{Tr} \varphi$ and $\Pi, i \vDash_{Tr} \psi$
$\Pi, i \vDash_{Tr} \bigcirc \varphi$	iff	$\Pi, i+1 \vDash_{Tr} \varphi$
$\Pi, i \vDash_{Tr} \varphi \mathcal{U} \psi$	iff	$\exists j \geq i.\, \Pi, j \vDash_{Tr} \psi \wedge \forall i \leq k < j.\, \Pi, k \vDash_{Tr} \varphi$
$\Pi, i \vDash_{Tr} \exists \pi.\, \varphi$	iff	there is some $t \in Tr$ such that $\Pi[\pi \mapsto t], i \vDash_{Tr} \varphi$
$\Pi, i \vDash_{Tr} \forall \pi.\, \varphi$	iff	for all $t \in Tr$ it holds that $\Pi[\pi \mapsto t], i \vDash_{Tr} \varphi$

We explain counterexamples found by MCHyper [24,35], which is a model checker for HyperLTL formulas, building on ABC [12]. MCHyper takes as inputs a hardware circuit, specified in the Aiger format [8], and a HyperLTL formula. MCHyper solves the model checking problem by computing the self-composition [6] of the system. If the system violates the HyperLTL formula, MCHyper returns a counterexample. This counterexample is a set of traces through the original system that together violate the HyperLTL formula. Depending on the type of violation, this counterexample can then be used to debug the circuit or refine the specification iteratively.

2.2 Actual Causality

A formal definition of what actually causes an observed effect in a given context has been proposed by Halpern and Pearl [45]. Here, we outline the version later modified by Halpern [44]. Causality is defined with respect to a *causal model* $\mathcal{M} = (\mathcal{S}, \mathcal{F})$, given by a *signature* $\mathcal{S}$ and set of *structural equations* $\mathcal{F}$, which define the dynamics of the system. A signature $\mathcal{S}$ is a tuple $(\mathcal{U}, \mathcal{V}, \mathcal{D})$, where $\mathcal{U}$ and $\mathcal{V}$ are disjoint sets of variables, termed *exogenous* and *endogenous* variables, respectively; and $\mathcal{D}$ defines the *range* of possible values $\mathcal{D}(Y)$ for all variables $Y \in \mathcal{U} \cup \mathcal{V}$. A *context* $\vec{u}$ is an assignment to the variables in $\mathcal{U} \cup \mathcal{V}$ such that the values of the exogenous variables are determined by factors outside of the model, while the value of some endogenous variable X is defined by the associated structural equation $f_X \in \mathcal{F}$. An *effect* φ in a causal model is a Boolean formula over assignments to endogenous variables. We say that a context $\vec{u}$ of a model $\mathcal{M}$ satisfies a partial variable assignment $\vec{X} = \vec{x}$ for $\vec{X} \subseteq \mathcal{U} \cup \mathcal{V}$ if the assignments in $\vec{u}$ and in $\vec{x}$ coincide for every variable $X \in \vec{X}$. The extension for Boolean formulas over variable assignments is as expected. For a context $\vec{u}$ and a partial variable assignment $\vec{X} = \vec{x}$, we denote by $(\mathcal{M}, \vec{u})[\vec{X} \leftarrow \vec{x}]$ the context $\vec{u}'$ in which the values of the variables in $\vec{X}$ are set according to $\vec{x}$, and all other values are computed according to the structural equations.

The actual causality framework of Halpern and Pearl aims at defining what events (given as variable assignments) are the cause for the occurrence of an effect in a specific given context. We now provide the formal definition.

Definition 1 ([44,45]). *A partial variable assignment $\vec{X} = \vec{x}$ is an* actual cause *of the effect φ in $(\mathcal{M}, \vec{u})$ if the following three conditions hold.*

AC1: $(\mathcal{M}, \vec{u}) \vDash \vec{X} = \vec{x}$ and $(\mathcal{M}, \vec{u}) \vDash \varphi$, i.e., both cause and effect are true in the actual world.

AC2: There is a set $\vec{W} \subseteq V$ of endogenous variables and an assignment $\vec{x}'$ to the variables in $\vec{X}$ s.t. if $(\mathcal{M}, \vec{u}) \vDash \vec{W} = \vec{w}$, then $(\mathcal{M}, \vec{u})[\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}] \vDash \neg\varphi$.

AC3: $\vec{X}$ is minimal, i.e. no subset of $\vec{X}$ satisfies AC1 and AC2.

Intuitively, AC2 states that in the counterfactual world obtained by intervening on the cause $\vec{X} = \vec{x}$ in the actual world (that is, setting the variables in $\vec{X}$ to $\vec{x}'$), the effect does not appear either. However, intervening on the possible cause might not be enough, for example when that cause preempted another. After intervention, this other cause may produce the effect again, therefore clouding the effect of the intervention. To address this problem, AC2 allows to reset values through the notion of *contingencies*, i.e., the set of variables $\vec{W}$ can be reset to $\vec{w}$, which is (implicitly) universally quantified. However, since the actual world has to model $\vec{W} = \vec{w}$, it is in fact uniquely determined. AC3, lastly, enforces the cause to be minimal by requiring that all variables in $\vec{X}$ are strictly necessary to achieve AC1 and AC2. For an illustration of Halpern and Pearl's actual causality, see Example 1 in Sect. 3.

3 Running Example

Consider a security-critical setting with two security levels: a high-security level h and a low-security level l. Inputs and outputs labeled as high-security, denoted by hi and ho respectively, are confidential and thus only visible to the user itself, or, e.g., admins. Inputs and outputs labeled as low-security, denoted by li and lo respectively, are public and are considered to be observable by an attacker.

Our system of interest is modeled by the state graph representation shown in Fig. 1, which is treated as a black box by an attacker. The system is run without any low-security inputs, but branches depending on the given high-security inputs. If in one of the first two steps of an execution, a high-security input hi is encountered, the system outputs only the high-security variable ho directly afterwards and in the subsequent steps both outputs, regardless of inputs. If no high-security input is given in the first step, the low-security output lo is enabled and after the second step, again both outputs are enabled, regardless of what input is fed into the system.

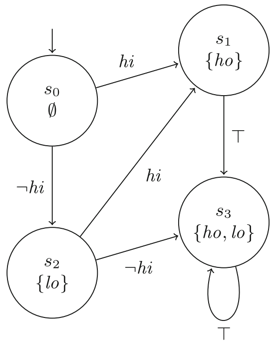

Fig. 1. State graph representation of our example system.

A prominent example hyperproperty is *observational determinism* from the introduction which states that any sequence of low-inputs always produces the same low-outputs, regardless of what the high-security level inputs are. $\varphi = \forall \pi . \forall \pi' . \square(li_\pi \leftrightarrow li_{\pi'}) \rightarrow \square(lo_\pi \leftrightarrow lo_{\pi'})$. The formula states that all traces π and π' must agree in the low-security outputs if they agree in the low-security inputs. Our system at hand does not satisfy observational determinism, because the low-security outputs in the first two steps depend on the present high-security inputs. Running MCHyper, a model checker for HyperLTL, results in the following counterexample: $t_1 = \{\}\{lo\}\{ho, lo\}^\omega$ and $t_2 = \{hi\}\{hi, ho\}\{ho, lo\}^\omega$. With the same low-security input (none) the traces produce different low-security outputs by visiting s_1 or s_2 on the way to s_3.

In this paper, our goal is to explain the violation of a HyperLTL formula on such a counterexample. Following Halpern and Pearl's explanation framework [46], an actual cause that is considered to be possibly true or possibly false constitutes an explanation for the user. We only consider causes over input variables, which can be true and false in any model. Hence, finding an explanation amounts to answering which inputs caused the violation on a specific counterexample. Before we answer this question for HyperLTL and the corresponding counterexamples given by sets of traces (see Sect. 4), we first illustrate Halpern and Pearl's actual causality (see Sect. 2.2) with the above running example.

Example 1. Finite executions of a system can be modeled in Halpern and Pearl's causal models. Consider inputs as exogenous variables $\mathcal{U} = \{hi_0, hi_1\}$ and outputs as endogenous variables $\mathcal{V} = \{lo_1, lo_2, ho_1, ho_2\}$. The indices model at

which step of the execution the variable appears. We omit the inputs at the third position and the outputs at the first position because they are not relevant for the following exposition. We have that $\mathcal{D}(Y) = \{0, 1\}$ for every $Y \in \mathcal{U} \cup \mathcal{V}$. Now, the following manually constructed structural equations encode the transitions: (1) $lo_1 = \neg hi_0$, (2) $ho_1 = hi_0$, (3) $lo_2 = \neg hi_1 \vee \neg lo_1$ and (4) $ho_2 = lo_1 \vee ho_1$. Consider context $\vec{u} = \{hi_0 = 0, hi_1 = 1\}$, effect $\varphi \equiv lo_1 = 1 \vee lo_2 = 1$, and candidate cause $hi_0 = 0$. Because of (1), we have that $(\mathcal{M}, \vec{u}) \vDash hi_0 = 0$ and $(\mathcal{M}, \vec{u}) \vDash lo_1 = 1$, hence AC1 is satisfied. Regarding AC2, this example allows us to illustrate the need for contingencies to accurately determine the actual cause: If we only consider intervening on the candidate cause $hi_0 = 0$, we still have $(\mathcal{M}, \vec{u})[hi_0 \leftarrow 1] \vDash \varphi$, because with $lo_1 = 0$ and (3) it follows that $(\mathcal{M}, \vec{u}) \vDash lo_2 = 1$. However, in the actual world, the second high input has no influence on the effect. We can control for this by considering the contingency $lo_2 = 0$, which is satisfied in the actual world, but not after the intervention on hi_0. Because of this contingency, we then have that $(\mathcal{M}, \vec{u})[hi_0 \leftarrow 1, lo_2 \leftarrow 0] \vDash \neg\varphi$, and hence, AC2 holds. Because a singleton set automatically satisfies AC3, we can infer that the first high input hi_0 was the actual cause for any low output to be enabled in the actual world. Note that, intuitively, the contingency allows us to ignore some of the structural equations by ignoring the value they assign to lo_2 in this context. Our definitions in Sect. 4 will allow similar modifications for counterexamples to hyperproperties.

4 Causality for Hyperproperty Violations

Our goal in this section is to formally define actual causality for the violation of a hyperproperty described by a general HyperLTL formula φ, observed in a counterexample to φ. Such a counterexample is given by a trace assignment to the trace variables appearing in φ. Note that, for universal quantifiers, the assignment of a single trace to the bounded variable suffices to define a counterexample. For existential quantifiers, this is not the case: to prove that an existential quantifier cannot be instantiated we need to show that no system trace satisfies the formula in its body, i.e., provide a proof for the whole system. In this work, we are interested in explaining violations of hyperproperties, and not proofs of their satisfaction [16]. Hence, we limit ourselves to instantiations of the outermost universal quantifiers of a HyperLTL formula, which can be returned by model checkers like MCHyper [24,35]. Since our goal is to explain counterexamples, restricting ourselves to results returned by existing model checkers is reasonable. Note that MCHyper can still handle formulas of the form $\forall^n \exists^m \varphi$ where φ is quantifier free, including interesting information flow policies like generalized noninterference [61]. The returned counterexample then only contains n traces that instantiate the universal quantifiers, the existential quantifiers are not instantiated for the above reason. In the following, we restrict ourselves to formulas and counterexamples of this form.

Definition 2 (Counterexample). *Let T be a transition system and denote $Traces(T) := Tr$, and let φ be a HyperLTL formula of the form $\forall \pi_1 \ldots \forall \pi_k \psi$,*

where ψ is a HyperLTL formula that does not start with $\forall$. A counterexample to φ in T is a partial trace assignment $\Gamma : \{\pi_1, \ldots, \pi_k\} \to Tr$ such that $\Gamma, 0 \vDash_{Tr} \neg\psi$.

For ease of notation, we sometimes refer to Γ simply as the tuple of its instantiations $\Gamma = \langle \Gamma(\pi_1), \ldots, \Gamma(\pi_k)\rangle$. In terms of Halpern and Pearl's actual causality as outlined in Sect. 2.2, a counterexample describes the actual world at hand, which we want to explain. As a next step, we need to define an appropriate language to reason about possible causes and contingencies in our counterexample. We will use sets of *events*, i.e., values of atomic propositions at a specific position of a specific trace in the counterexample.

Definition 3 (Event). *An event is a tuple $e = \langle l_a, n, t\rangle$ such that $l_a = a$ or $l_a = \neg a$ for some atomic proposition $a \in AP$, $n \in \mathbb{N}$ is a point in time, and $t \in (2^{AP})^\omega$ is a trace of a system T. We say that a counterexample $\Gamma = \langle t_1, \ldots t_k\rangle$* satisfies *a set of events $\mathcal{C}$, and denote $\Gamma \vDash \mathcal{C}$, if for every event $\langle l_a, n, t\rangle \in \mathcal{C}$ the two following conditions hold:*

1. $t = t_i$ for some $i \in \{1, \ldots, k\}$, i.e., all events in $\mathcal{C}$ reason about traces in Γ,
2. $l_a = a$ iff $a \in t_i[n]$, i.e., a holds on trace t_i of the counterexample at time n.

We assume that the set AP is a disjoint union of input an output propositions, that is, $AP = I \cupdot O$. We say that $\langle l_a, n, t\rangle$ is an *input event* if $a \in I$, and we call it an *output event* if $a \in O$. We denote the set of input events by IE and the set of output events by OE. These events have a direct correspondence with the variables appearing in Halpern and Pearl's causal models: we can identify input events with exogenous variables (because their value is determined by factors outside of the system) and output events with endogenous variables.

We define a cause as a set of input events, while an effect is a possibly infinite Boolean formula over OE. Note that, similar to [37], every HyperLTL formula can be represented as a first order formula over events, e.g. $\forall\pi\forall\pi' \square(a_\pi \leftrightarrow a_{\pi'}) = \forall\pi\forall\pi' \bigwedge_{n\in\mathbb{N}}(\langle a, n, \pi\rangle \leftrightarrow \langle a, n, \pi'\rangle)$. For some set of events $\mathcal{S}$, let ${}^{+}\mathcal{S}^k_\pi = \{a \in AP \mid \langle a, k, \pi\rangle \in \mathcal{S}\}$ denote the set of atomic propositions defined positively by $\mathcal{S}$ on trace π at position k. Dualy, we define ${}^{-}\mathcal{S}^k_\pi = \{a \in AP \mid \langle \neg a, k, \pi\rangle \in \mathcal{S}\}$.

In order to define actual causality for hyperproperties we need to formally define how we obtain the counterfactual executions under some contingency for the case of events on infinite traces. We define a contingency as a set of output events. Mapping Halpern and Pearl's definition to transition systems, contingencies reset outputs in the counterfactual traces back to their value in the original counterexample, which amounts to changing the state of the system, and then following the transition function from the new state. For a given trace of the counterexample, we describe all possible behaviors under *arbitrary* contingencies with the help of a counterfactual automaton. The concrete contingency on a trace is defined by additional input variables. In the following, let $IC = \{o^C \mid o \in O\}$ be a set of auxiliary input variables expressing whether a contingency is invoked at the given step of the execution and $c : O \to IC$ be a function s.t. $c(o) = o^C$.

Definition 4 (Counterfactual Automaton). *Let $T = (S, s_0, AP, \delta, l)$ be a system with $S = 2^O$, i.e., every state is uniquely labeled, and there exists a state*

for every combination of outputs. Let $\pi = \pi_0 \dots \pi_i(\pi_j \dots \pi_n)^\omega \in traces(T)$ *be a trace of* T *in a finite, lasso-shaped representation. The counterfactual automaton* $T_\pi^C = (S \times \{0 \dots n\}, (s_0, 0), (IC \cup I) \cup (O \cup \{0 \dots n\}), \delta^C, l^C)$ *is defined as follows:*

- $\delta^C((s,k), Y) = (s', k')$ *where* $k' = j$ *if* $k = n$, *else* $k' = k + 1$, *and* $l(s') = \{o \in O \mid (o \in \delta(s, Y \cap I) \wedge c(o) \notin Y) \vee (o \in \pi_{k'} \wedge c(o) \in Y)\}$,
- $l^C(s,k) = l(s) \cup \{k\}$.

A counterfactual automaton is effectively a chain of copies of the original system, of the same length as the counterexample. An execution through the counterfactual automaton starts in the first copy corresponding to the first position in the counterexample trace, and then moves through the chain until it eventually loops back from copy n to copy j. A transition in the counterfactual automaton can additionally specify setting as a contingency some output variable o if the auxiliary input variable o^C is enabled. In this case, the execution will move to a state in the next automaton of the chain where all the outputs are as usual, except o, which will have the same value as in the counterexample π. Note that, under the assumption that all states of the original system are uniquely labeled and there exists a state for every combination of output variables, the function δ^C is uniquely determined.[1] A counterfactual automaton for our running example is described in the full version of this paper [22].

Next, we need to define how we intervene on a set of traces with a candidate cause given as a set of input events, and a contingency given as a set of output events. We define an intervention function, which transforms a trace of our original automaton to an input sequence of an counterfactual automaton.

Definition 5 (Intervention). *For a cause* $\mathcal{C} \subseteq IE$, *a contingency* $\mathcal{W} \subseteq OE$ *and a trace* π, *the function* $intervene : (2^{AP})^\omega \times 2^{IE} \times 2^{OE} \to (2^{I \cup IC})^\omega$ *returns a trace such that for all* $k \in \mathbb{N}$ *the following holds:* $intervene(\pi, \mathcal{C}, \mathcal{W})[k] = (\pi[k] \setminus {}^+\mathcal{C}_\pi^k) \cup {}^-\mathcal{C}_\pi^k \cup \{c(o) \mid o \in {}^+\mathcal{W}_\pi^k \cup {}^-\mathcal{W}_\pi^k\}$. *We lift the intervention function to counterexamples given as a tuple* $\Gamma = \langle \pi_1, \dots, \pi_k \rangle$ *as follows:* $intervene(\Gamma, \mathcal{C}, \mathcal{W}) = \langle T_{\pi_1}^C(intervene(\pi_1, \mathcal{C}, \mathcal{W})), \dots, T_{\pi_k}^C(intervene(\pi_k, \mathcal{C}, \mathcal{W}))\rangle$.

Intuitively, the intervention function *flips* all the events that appear in the cause Γ: If some $a \in I$ appears positively in the candidate cause $\mathcal{C}$, it will appear negatively in the resulting input sequence, and vice-versa. For a contingency $\mathcal{W}$, the intervention function enables their auxiliary input for the counterfactual automaton at the appropriate time point irrespective of their value, as the counterfactual automaton will take care of matching the atomic propositions value to the value in the original counterexample Γ.

[1] The same reasoning can be applied to arbitrary systems by considering for contingencies largest sets of outputs for which the assumption holds, with the caveat that the counterfactual automaton may model fewer contingencies. Consequently, computed causes may be less precise in case multiple causes appear in the counterexample.

4.1 Actual Causality for HyperLTL Violations

We are now ready to formalize what constitutes an actual cause for the violation of a hyperproperty described by a HyperLTL formula.

Definition 6 (Actual Causality for HyperLTL). *Let Γ be a counterexample to a HyperLTL formula φ in a system T. The set $\mathcal{C}$ is an actual cause for the violation of φ on Γ if the following conditions hold.*

SAT $\Gamma \vDash \mathcal{C}$.
CF *There exists a contingency $\mathcal{W}$ and a non-empty subset $\mathcal{C}' \subseteq \mathcal{C}$ such that: $\Gamma \vDash \mathcal{W}$ and $intervene(\Gamma, \mathcal{C}', \mathcal{W}) \vDash_{traces(T)} \varphi$.*
MIN *$\mathcal{C}$ is minimal, i.e., no subset of $\mathcal{C}$ satisfies SAT and CF.*

Unlike in Halpern and Pearl's definition (see Sect. 2.2), the condition SAT requires Γ to satisfy only the cause, as we already know that the effect $\neg\varphi$, i.e., the violation of the specification, is satisfied by virtue of Γ being a counterexample. CF is the counterfactual condition corresponding to AC2 in Halpern and Pearl's definition, and it states that after intervening on the cause, under a certain contingency, the set of traces satisfies the property. (Note that we use a conjunction of two statements here while Halpern and Pearl use an implication. This is because they implicitly quantify universally over the values of the variables in the set W (which should be as in the actual world) where in our setting the set of contingencies already defines explicit values.) MIN is the minimality criterion directly corresponding to AC3.

Example 2. Consider our running example from Sect. 3, i.e., the system from Fig. 1 and the counterexample to observational determinism $\Gamma = \langle t_1, t_2 \rangle$. Let us consider what it means to intervene on the cause $\mathcal{C}_1 = \{\langle hi, 0, t_2 \rangle\}$. Note that we have $\Gamma \vDash \mathcal{C}_1$, hence the condition SAT is satisfied. For CF, let us first consider an intervention without contingencies. This results in $intervene(\Gamma, \mathcal{C}_1, \emptyset) = \langle t_1', t_2' \rangle = \langle t_1, \{\}\{hi, lo\}\{ho\}\{ho, lo\}^\omega \rangle$. However, $intervene(\Gamma, \mathcal{C}_1, \emptyset) \vDash_{traces(T)} \neg\varphi$, because the low outputs of t_1' and t_2' differ at the third position: $lo \in t_1'[2]$ and $lo \notin t_2'[2]$. This is because now the second high input takes effect, which was preempted by the first cause in the actual counterexample. The contingency $\mathcal{W}_2 = \{\langle lo, 2, t_2 \rangle\}$ now allows us to control this by *modyfing the state* after taking the second high input as follows: $intervene(\Gamma, \mathcal{C}_2, \mathcal{W}_2)) = \langle t_1'', t_2'' \rangle = \langle t_1, \{\}\{hi, lo\}\{ho, lo\}\{ho, lo\}^\omega \rangle$. Note that t_2'' is not a trace of the model depicted in Fig. 1, because there is no transition that explains the step from $t_2''[1]$ to $t_2''[2]$. It is, however, a trace of the counterfactual automaton $T_{t_2}^C$ (see full version [22]), which encodes the set of counterfactual worlds for the trace t_2. The fact that we consider executions that are not part of the original system allows us to infer that only the first high input was an actual cause in our running example. Disregarding contingencies, we would need to consider both high inputs as an explanation for the violation of observational determinism, even though the second high input had no influence. Our treatment of contingencies corresponds directly to Halpern and Pearl's causal models, which allow to ignore certain structural equations as outlined in Example 1.

Remark: With our definitions, we strictly generalize Halpern and Pearl's actual causality to reactive systems modeled as Moore machines and effects expressed as HyperLTL formulas. Their structural equation models can be encoded in a one-step Moore machine; effect specifying a Boolean combination of primitive events can be encoded in the more expressive logic HyperLTL. Just like for Halpern and Pearl, our actual causes are not unique. While there can exist several different actual causes, the set of all actual causes is always unique. It is also possible that no actual cause exists: If the effect occurs on all system traces, there may be no actual cause on a given individual trace.

4.2 Finding Actual Causes with Model Checking

In this section, we consider the relationship between finding an actual cause for the violation of a HyperLTL formula starting with a universal quantifier and model checking of HyperLTL. We show that the problem of finding an actual cause can be reduced to a model checking problem where the generated formula for the model checking problem has one additional quantifier alternation. While there might be a reduction resulting in a more efficient encoding, our current result suggests that causality checking is the harder problem. The key idea of our reduction is to use counterfactual automata (that encode the given counterexample and the possible counterfactual traces) together with the HyperLTL formula described in the proof to ensure the conditions SAT, CF, and MIN on the witnesses for the model checking result.

Proposition 1. *We can reduce the problem of finding an actual cause for the violation of an HyperLTL formula starting with a universal quantifier to the HyperLTL model checking problem with one additional quantifier alternation.*

Proof. Let $\Gamma = \langle t_1, \dots t_k \rangle$ be a counterexample for the formula $\forall \pi_1 \dots \forall \pi_k . \varphi$ where φ is a HyperLTL formula that does not have a universal first quantifier. We provide the proof for the case of $\Gamma = \langle t_1, t_2 \rangle$ for readability reasons, but it can be extended to any natural number k. We assume that t_1, t_2 have some ω-regular representation, as otherwise the initial problem of computing causality is not well defined. That is, we denote $t_i = u_i(v_i)^\omega$ such that $|u_i \cdot v_i| = n_i$.

In order to find an actual cause, we need to find a pair of traces t'_1, t'_2 that are counterfactuals for t_1, t_2; satisfy the property φ; and the changes from t_1, t_2 to t'_1, t'_2 are minimal with respect to set containment. Changes in inputs between t_i and t'_i in the loop part v_i should reoccur in t'_i repeatedly. Note that the differences between the counterexample $\langle t_1, t_2 \rangle$ and the witness of the model checking problem $\langle t'_1, t'_2 \rangle$ encode the actual cause, i.e. in case of a difference, the cause contains the event that is present on the counterexample. To reason about these changes, we use the counterfactual automaton T_i^C for each t_i, which also allows us to search for the contingency $\mathcal{W}$ as part of the input sequence of T_i^C. Note that each T_i^C consists of n_i copies, that indicate in which step the automaton is with respect to t_i and its loop v_i. For $m > |u_i|$, we label each state (s_i, m) in T_i^C with the additional label $L_{s_m,i}$, to indicate that the system is now

in the loop part of t_i. In addition, we add to the initial state of T_i^C the label l_i, and we add to the initial state of the system T the label l_{or}. The formula ψ^i_{loop} below states that the trace π begins its run from the initial state of T_i^C (and thus stays in this component through the whole run), and that every time π visits a state on the loop, the same input sequence is observed. This way we enforce the periodic input behavior of the traces t_1, t_2 on t'_1, t'_2.

$$\psi^i_{loop}(\pi) := l_{i,\pi} \wedge \bigwedge_{L_{s_m,i}} \bigvee_{A \subseteq I} \square(L_{s_m,i,\pi} \rightarrow (\bigwedge_{a \in A} a_\pi \wedge \bigwedge_{a \notin A} \neg a_\pi))$$

For a subset of locations $N \subseteq [1, n_i]$ and a subset of input propositions $A \subseteq I$ we define $\psi^i_{diff}[N, A](\pi)$ that states that π differs from t_i in at least all events $\langle l_a, m, t_i \rangle$ for $a \in A, m \in N$; and the formula $\psi^i_{eq}[N, A](\pi)$ that states that for all events that are not defined by A and N, π is equal to t_i.

$$\psi^i_{diff}[N, A](\pi) = \bigwedge_{j \in N, a \in A} \bigcirc^j (a_\pi \not\leftrightarrow a_{t_i})$$

$$\psi^i_{eq}[N, A](\pi) = \bigwedge_{j \notin N, a \in I} \bigcirc^j (a_\pi \leftrightarrow a_{t_i}) \wedge \bigwedge_{j \in [1, n_i], a \notin A} \bigcirc^j (a_\pi \leftrightarrow a_{t_i})$$

We now define the formula ψ^i_{min} that states that the set of inputs (and locations) on which trace π differs from t_i is not contained in the corresponding set for π'. We only check locations up until the length n_i of t_i.

$$\psi^i_{min}(\pi, \pi') := \bigwedge_{N \subseteq [i, n_i]} \bigwedge_{A \subseteq \mathcal{I}} \big(\big(\psi^i_{diff}[N, A](\pi) \wedge \psi^i_{eq}[N, A](\pi)\big) \rightarrow \neg\psi^i_{eq}[N, A](\pi')\big)$$

Denote $\varphi := Q_1\tau_1 \ldots Q_n\tau_n.\ \varphi'(\pi_1, \pi_2)$ where $Q_i \in \{\forall, \exists\}$ and τ_i are trace variables for $i \in [1, n]$. The formula ψ_{cause} described below states that the two traces π'_1 and π'_2 are part of the systems T_1^C, T_2^C, and have the same loop structure as t_1 and t_2, and satisfy φ. That is, these traces can be obtained by changing the original traces t_1, t_2 and avoid the violation.

$$\psi_{cause}(\pi'_1, \pi'_2) := \varphi'(\pi'_1, \pi'_2) \wedge \bigwedge_{i=1,2} \psi^i_{loop}(\pi'_i)$$

Finally, ψ_{actual} described below states that the counterfactuals π'_1, π'_2 correspond to a minimal change in the input events with respect to t_1, t_2. All other traces that the formula reasons about start at the initial state of the original system and thus are not affected by the counterfactual changes. We verify ψ_{actual} against the product automaton $T \times T_1^C \times T_2^C$ to find these traces $\pi'_i \in T_i^C$ that witness the presence of a cause, counterfactual and contingency.

$$\psi_{actual} := \exists\pi'_1\exists\pi'_2.\ \forall\pi''_1\pi''_2.\ Q_1\tau_1 \ldots Q_n\tau_n.\ \psi_{cause}(\pi'_1, \pi'_2) \wedge \bigwedge_{i=1,2} (l_{i,\pi'_i} \wedge l_{i,\pi''_i})$$
$$\wedge \bigwedge_{i \in [1,n]} l_{or,\tau_i} \wedge \left(\psi_{cause}(\pi''_1, \pi''_2) \rightarrow \left(\bigwedge_{i=1,2} \psi^i_{min}(\pi'_i, \pi''_i)\right)\right)$$

Then, if there exists two such traces π_1', π_2' in the system $T \times T_1^C \times T_2^C$, they correspond to a minimal cause for the violation. Otherwise, there are no traces of the counterfactual automata that can be obtained from t_1, t_2 using counterfactual reasoning and satisfy the formula φ. □

We have shown that we can use HyperLTL model checking to find an actual cause for the violation of a HyperLTL formula. The resulting model checking problem has an additional quantifier alternation which suggests that identifying actual causes is a harder problem. Therefore, we restrict ourselves to finding actual causes for violations of universal HyperLTL formulas. This keeps the algorithms we present in the next section practical as we start without any quantifier alternation and need to solve a model checking problem with a single quantifier alternation. While this restriction excludes some interesting formulas, many can be strengthened into this fragment such that we are able to handle close approximations (c.f. [25]). Any additional quantifier alternation from the original formula carries over to an additional quantifier alternation in the resulting model checking problem which in turn leads to an exponential blow-up. The scalability of our approach is thus limited by the complexity of the model checking problem.

5 Computing Causes for Counterexamples

In this section, we describe our algorithm for finding actual causes of hyperproperty violations. Our algorithm is implemented on top of MCHyper [35], a model checker for hardware circuits and the alternation-free fragment of HyperLTL. In case of a violation, our analysis enriches the provided counterexample with the actual cause which can explain the reason for the violaiton to the user.

We first provide an overview of our algorithm and then discuss each step in detail. First, we compute an over-approximation of the cause using a satisfiability analysis over transitions taken in the counterexample. This analysis results in a set of events $\tilde{\mathcal{C}}$. As we show in Proposition 2, every actual cause $\mathcal{C}$ for the violation is a subset of $\tilde{\mathcal{C}}$. In addition, in Proposition 3 we show that the set $\tilde{\mathcal{C}}$ satisfies conditions SAT and CF. To ensure MIN, we search for the smallest subset $\mathcal{C} \subseteq \tilde{\mathcal{C}}$ that satisfies SAT and CF. This set $\mathcal{C}$ is then our minimal and therefore actual cause.

To check condition CF, we need to check the counterfactual of each candidate cause $\mathcal{C}$, and potentially also look for contingencies for $\mathcal{C}$. We separate our discussion as follows. We first discuss the calculation of the over-approximation $\tilde{\mathcal{C}}$ (Sect. 5.1), then we present the ActualCause algorithm that identifies a minimal subset of $\tilde{\mathcal{C}}$ that is an actual cause (Sect. 5.2), and finally we discuss in detail the calculation of contingencies (Sect. 5.3). In the following sections, we use a reduction of the universal fragment of HyperLTL to LTL, and the advantages of the linear translation of LTL to alternating automata, as we now briefly outline.

HyperLTL to LTL. Let φ be a $\forall^n$-HyperLTL formula and Γ be the counterexample. We construct an LTL formula φ' from φ as follows [31]: atomic propositions indexed with different trace variables are treated as different atomic propositions and trace quantifiers are eliminated. For example $\forall\pi,\pi'.a_\pi \wedge a_{\pi'}$ results in the LTL formula $a_\pi \wedge a_{\pi'}$. As for Γ, we use the same renaming in order to zip all traces into a single trace, for which we assume the finite representation $t'' = u'' \cdot (v'')^\omega$, which is also the structure of the model checker's output. The trace t'' is a violation of the formula φ', i.e., t'' satisfies $\neg\varphi'$. We denote $\bar{\varphi} := \neg\varphi'$. We can then assume, for implementation concerns, that the specification (and its violation) is an LTL formula, and the counterexample is a single trace. After our causal analysis, the translation back to a cause over hyperproperties is straightforward as we maintain all information about the different traces in the counterexample. Note that this translation works due to the synchronous semantics of HyperLTL.

Finite Trace Model Checking Using Alternating Automata. In verifying condition CF (that is, in computing counterfactuals and contingencies), we need to apply finite trace model checking, as we want to check if the modified trace in hand still violates the specification φ, that is, satisfies $\bar{\varphi}$. To this end, we use the linear algorithm of [36], that exploits the linear translation of $\bar{\varphi}$ to an alternating automaton $\mathcal{A}_{\bar{\varphi}}$, and using backwards analysis checks the satisfaction of $\bar{\varphi}$. An alternating automaton [68] generalizes non-deterministic and universal automata, and its transition relation is a Boolean function over the states. The run of alternating automaton is then a *tree run* that captures the conjunctions in the formula. We use the algorithm of [36] as a black box (see App. A.2 in [22] for a formal definition of alternating automata and App. A.3 in [22] for the translation from LTL to alternating automata). For the computation of contingencies we use an additional feature of the algorithm of [36] – the algorithm returns an accepting run tree $\mathcal{T}$ of $\mathcal{A}_{\bar{\varphi}}$ on t'', with annotations of nodes that represent atomic subformulas of $\bar{\varphi}$ that take part in the satisfaction of $\bar{\varphi}$. We use this feature also in Sect. 5.1 when calculating the set of candidate causes.

5.1 Computing the Set of Candidate Causes

The events that might have been a part of the cause to the violation are in fact all events that appear on the counterexample, or, equivalently, all events that appear in u'' and v''. Note that due to the finite representation, this is a finite set of events. Yet, not all events in this set can cause the violation. In order to remove events that could not have been a part of the cause, we perform an analysis of the transitions of the system taken during the execution of t''. With this analysis we detect which events appearing in the trace locally cause the respective transitions, and thus might be part of the global cause. Events that did not trigger a transition in this specific trace cannot be a part of the cause. Note that causing a transition and being an actual cause are two different notions - actual causality is defined over the behaviour of the system,

not on individual traces. We denote the over-approximation of the cause as $\tilde{\mathcal{C}}$. Formally, we represent each transition as a Boolean function over inputs and states. Let δ_n denote the formula representing the transition of the system taken when reading $t''[n]$, and let $c_{a,n,i}$ be a Boolean variable that corresponds to the event $\langle a_{t_i}, n, t''\rangle$.[2] Denote $\psi_n^t = \bigwedge_{a_{t_i} \in t''[n]} c_{a,n,i} \wedge \bigwedge_{a_{t_i} \notin t''[n]} \neg c_{a,n,i}$, that is, ψ_n^t expresses the exact set of events in $t''[n]$. In order to find events that might trigger the transition δ_n, we check for the *unsatisfiable core* of $\psi_n = (\neg\delta_n) \wedge \psi_n^t$. Intuitively, the unsatisfiable core of ψ_n is the set of events that force the system to take this specific transition. For every $c_{a,n,i}$ (or $\neg c_{a,n,i}$) in the unsatisfiable core that is also a part of ψ_n^t, we add $\langle a, n, t_i\rangle$ (or $\langle \neg a, n, t_i\rangle$) to $\tilde{\mathcal{C}}$.

We use unsatisfiable cores in order to find input events that are necessary in order to take a transition. However, this might not be enough. There are cases in which inputs that appear in formula $\bar{\varphi}$ are not detected using this method, as they are not essential in order to take a transition; however, they might be considered a part of the actual cause, as negating them can avoid the violation. Therefore, as a second step, we apply the algorithm of [36] on the annotated automaton $\mathcal{A}_{\bar{\varphi}}$ in order to find the specific events that affect the satisfaction of $\bar{\varphi}$, and we add these events to $\tilde{\mathcal{C}}$. Then, the unsatisfiable core approach provides us with inputs that affect the computation and might cause the violation even though they do not appear on the formula itself; while the alternating automaton allows us to find inputs that are not essential for the computation, but might still be a part of the cause as they appear on the formula.

Proposition 2. *The set $\tilde{\mathcal{C}}$ is indeed an over-approximation of the cause for the violation. That is, every actual cause $\mathcal{C}$ for the violation is a subset of $\tilde{\mathcal{C}}$.*

Proof (sketch). Let $e = \langle l_a, n, t\rangle$ be an event such that e is not in the unsatisfiable core of ψ_n and does not directly affect the satisfaction of $\bar{\varphi}$ according to the alternating automata analysis. That is, the transition corresponding to ψ_n^t is taken regardless of e, and thus all future events on t remain the same regardless of the valuation of e. In addition, the valuation of the formula $\bar{\varphi}$ is the same regardless of e, since: (1) e does not directly affect the satisfaction of $\bar{\varphi}$; (2) e does not affect future events on t (and obviously it does not affect past events). Therefore, every set $\mathcal{C}'$ such that $e \in \mathcal{C}'$ is not minimal, and does not form a cause. Since the above is true for all events $e \notin \mathcal{C}$, it holds that $\mathcal{C} \subseteq \tilde{\mathcal{C}}$ for every actual cause $\mathcal{C}$. □

Proposition 3. *The set $\tilde{\mathcal{C}}$ satisfies conditions SAT and CF.*

Proof. The condition SAT is satisfied as we add to $\tilde{\mathcal{C}}$ only events that indeed occur on the counterexample trace. For CF, consider that $\tilde{\mathcal{C}}$ is a super-set of the actual cause $\mathcal{C}$, so the same contingency and counterfactual of $\mathcal{C}$ will also apply for $\tilde{\mathcal{C}}$. This is since in order to compute counterfactual we are allowed to flip any subset of the events in $\mathcal{C}$, and any such subset is also a subset of $\tilde{\mathcal{C}}$.

[2] That is, $\neg c_{a,n,i}$ corresponds to the event $\langle \neg a_{t_i}, n, t''\rangle$. Recall that the atomic propositions on the zipped trace t'' are annotated with the original trace t_i from Γ.

Algorithm 1: $\mathsf{ActualCause}(\varphi, \Gamma, \tilde{\mathcal{C}})$

Input: Hyperproperty φ, counterexample Γ violating φ, and a set of candidate causes $\tilde{\mathcal{C}}$ for which conditions SAT and CF hold.
Output: A set of input events $\mathcal{C}$ which is an actual cause for the violation.

```
1  for i ∈ [1, ..., |C̃| − 1] do
2      for C ⊂ C̃ with |C| = i do
3          let Γ^f = intervene(Γ, C, ∅);
4          if Γ^f ⊨ φ then
5              return C;
6          else
7              W̃ = ComputeContingency(φ, Γ, C);
8              if W̃ ≠ ∅ then
9                  return C;
10 return C̃;
```

In addition, in computing contingencies, we are allowed to flip any subset of outputs as long as they agree with the counterexample trace, which is independent in $\tilde{\mathcal{C}}$ and $\mathcal{C}$. □

5.2 Checking Actual Causality

Due to Proposition 2 we know that in order to find an actual cause, we only need to consider subsets of $\tilde{\mathcal{C}}$ as candidate causes. In addition, since $\tilde{\mathcal{C}}$ satisfies condition SAT, so do all of its subsets. We thus only need to check conditions CF and MIN for subsets of $\tilde{\mathcal{C}}$. Our actual causality computation, presented in Algorithm 1 is as follows. We start with the set $\tilde{\mathcal{C}}$, that satisfies SAT and CF. We then check if there exists a more minimal cause that satisfies CF. This is done by iterating over all subsets $\mathcal{C}'$ of $\tilde{\mathcal{C}}$, ordered by size and starting with the smallest ones, and checking if the counterfactual for the $\mathcal{C}'$ manages to avoid the violation; and if not, if there exists a contingency for this $\mathcal{C}'$. If the answer to one of these questions is yes, then $\mathcal{C}'$ is a minimal cause that satisfies SAT, CF, and MIN, and thus we return $\mathcal{C}'$ as our actual cause. We now elaborate on CF and MIN.

CF. As we have mentioned above, checking condition CF is done in two stages – checking for counterfactuals and computing contingencies. We first show that we do not need to consider all possible counterfactuals, but only one counterfactual for each candidate cause.

Proposition 4. *In order to check if a candidate cause $\tilde{\mathcal{C}}$ is an actual cause it is enough to test the one counterfactual where all the events in $\tilde{\mathcal{C}}$ are flipped.*

Proof. Assume that there is a strict subset $\mathcal{C}$ of $\tilde{\mathcal{C}}$ such that we only need to flip the valuations of events in $\mathcal{C}$ in order to find a counterfactual or contingency, thus $\mathcal{C}$ satisfies CF. Since $\mathcal{C}$ is a more minimal cause than $\tilde{\mathcal{C}}$, we will find it during the minimality check. □

Algorithm 2: ComputeContingency($\varphi, \Gamma, \mathcal{C}$)

Input: Hyperproperty φ, a counterexample Γ and a potential cause $\mathcal{C}$.
Output: a set of output events $\mathcal{W}$ which is a contingency for φ, Γ and $\mathcal{C}$, or $\emptyset$ if no contingency found.

1. **let** t'' be the zipped trace of Γ, φ' be the LTL formula obtained from φ, and $\bar{\varphi} = \neg\varphi'$;
2. **let** $\mathcal{A}_{\bar{\varphi}}$ be the alternating automaton for $\bar{\varphi}$;
3. **let** t^f be the counterfactual trace obtained from t'' by flipping all events in $\mathcal{C}$;
4. **let** $\mathcal{N}$ be the sets of events derived from the annotated run tree of $\mathcal{A}_{\bar{\varphi}}$ on t^f;
5. **let** $\mathcal{O}' := \{\langle l_{a_t}, n, t''\rangle \in OE \mid a_t \in t''[n] \leftrightarrow a_t \notin t^f[n]\}$;
6. **for** *every subset* $\mathcal{W}' \subseteq (\mathcal{N} \cap \mathcal{O}')$, *and then for every other subset* $\mathcal{W}' \subseteq \mathcal{O}'$ **do**
7. $\quad t^m := intervene(t'', \mathcal{C}, \mathcal{W}')$;
8. $\quad$ **if** $t^m \vDash \varphi'$ **then**
9. $\quad\quad$ **return** $\mathcal{W}'$;
10. **return** $\emptyset$;

We assume that CF holds for the input set $\tilde{\mathcal{C}}$ and check if it holds for any smaller subset $\mathcal{C} \subset \tilde{\mathcal{C}}$. CF holds for $\mathcal{C}$ if (1) flipping all events in $\mathcal{C}$ is enough to avoid the violation of φ or if (2) there exists a non-empty set of contingencies for $\mathcal{C}$ that ensures that φ is not violated. The computation of contingencies is described in Algorithm 2. Verifying condition CF involves model checking traces against an LTL formula, as we check in Algorithm 1 (line 3) if the property φ is still violated on the counterfactual trace with the empty contingency, and on the counterfactual traces resulting from the different contingency sets we consider in Algorithm 2 (line 7). In both scenarios, we apply finite trace model checking, as described at the beginning of Sect. 5 (as we assume lasso-shaped traces).

MIN. To check if $\tilde{\mathcal{C}}$ is minimal, we need to check if there exists a subset of $\tilde{\mathcal{C}}$ that satisfies CF. We check CF for all subsets, starting with the smallest one, and report the first subset that satisfies CF as our actual cause. (Note that we already established that $\tilde{\mathcal{C}}$ and all of its subsets satisfy SAT.)

5.3 Computing Contingencies

Recall that the role of contingencies is to eliminate the effect of other possible causes from the counterfactual world, in case these causes did not affect the violation in the actual world. More formally, in computing contingencies we look for a set $\mathcal{W}$ of output events such that changing these outputs from their value in the counterfactual to their value in the counterexample t'' results in avoiding the violation. Note that the inputs remain as they are in the counterfactual. We note that the problem of finding contingencies is hard, and in general is equivalent to the problem of model checking. This is since we need to consider all traces that are the result of changing some subset of events (output + time step) from the counterfactual back to the counterexample, and to check if there exists a trace in this set that avoids the violation. Unfortunately, we are unable to avoid

an exponential complexity in the size of the original system, in the worst case. However, our experiments show that in practice, most cases do not require the use of contingencies.

Our algorithm for computing contingencies (Algorithm 2) works as follows. Let t^f be the counterfactual trace. As a first step, we use the annotated run tree $\mathcal{T}$ of the alternating automaton $\mathcal{A}_{\bar{\varphi}}$ on t^f to detect output events that appear in $\bar{\varphi}$ and take part in satisfying $\bar{\varphi}$. Subsets of these output events are our first candidates for contingencies as they are directly related to the violation (Algorithm 2 lines 4–9). If we were not able to find a contingency, we continue to check all possible subsets of output events that differ from the original counterexample trace. We test the different outputs by feeding the counterfactual automaton of Definition 4 with additional inputs from the set I^C. The resulted trace is then our candidate contingency, which we try to verify against φ. The number of different input sequences is bounded by the size of the product of the counterfactual automaton and the automaton for $\bar{\varphi}$, and thus the process terminates.

Theorem 1 (Correctness). *Our algorithm is sound and complete. That is, let Γ be a counterexample with a finite representation to a $\forall^n$-HyperLTL formula ψ. Then, our algorithm returns an actual cause for the violation, if such exists.*

Proof. Soundness. Since we verify each candidate set of inputs according to the conditions SAT, CF and MIN, it holds that every output of our algorithm is indeed an actual cause. *Completeness.* If there exists a cause, then due to Proposition 2, it is a subset of the finite set $\tilde{\mathcal{C}}$. Since in the worst case we test every subset of $\tilde{\mathcal{C}}$, if there exists a cause we will eventually find it. □

6 Implementation and Experiments

We implemented Algorithm 1 and evaluated it on publicly available example instances of HyperVis [48], for which their state graphs were available. In the following, we provide implementation details, report on the running time and show the usefulness of the implementation by comparing to the highlighting output of HyperVis. Our implementation is written in Python and uses py-aiger [69] and Spot [27]. We compute the candidate cause according to Sect. 5.1 with pysat [50], using Glucose 4 [3,66], building on Minisat [66]. We ran experiments on a MacBook Pro with a 3, 3 GHz Dual-Core Intel Core i7 processor and 16 GB RAM[3].

Experimental Results. The results of our experimental evaluation can be found in Table 1. We report on the size of the analyzed counterexample $|\Gamma|$, the size of the violated formula $|\varphi|$, how long it took to compute the first, over-approximated cause (see time($\tilde{\mathcal{C}}$)) and state the approximation $\tilde{\mathcal{C}}$ itself, the number of computed minimal causes $\#(\mathcal{C})$ and the time it took to compute all of them (see time($\forall\mathcal{C}$)). The `Running Example` is described in Sect. 3, the instance `Security in & out`

[3] Our prototype implementation and the experimental data are both available at: https://github.com/reactive-systems/explaining-hyperproperty-violations.

Table 1. Experimental results of our implementation. Times are given in ms.

Instance	$\lvert\Gamma\rvert$	$\lvert\varphi\rvert$	time($\tilde{\mathcal{C}}$)	$\tilde{\mathcal{C}}$	#($\mathcal{C}$)	time($\forall\mathcal{C}$)
`Running example`	10	9	19	$\neg hi^0_{t_1}, hi^0_{t_2}$	2	55
`Security in & out`	35	19	292	$hi^2_{t_1}, \neg hi^0_{t_1}, \neg hi^3_{t_1}, \neg hi^1_{t_1}$ $hi^2_{t_2}, hi^0_{t_2}, hi^1_{t_2}, hi^3_{t_2}$	8	798
`Drone example 1`	24	19	33	$bound^2_{t_1}, \neg bound^1_{t_1}, up^1_{t_1}, \neg up^2_{t_1}$ $bound^2_{t_2}, \neg bound^1_{t_2}, \neg up^1_{t_2}$	5	367
`Drone example 2`	18	36	31	$bound^1_{t_1}, \neg bound^1_{t_2}, up^1_{t_2}$	3	256
`Asymmetric arbiter '19`	28	35	53	see App. A.4 in [22]	10	490
`Asymmetric arbiter`	72	35	70	see App. A.4 in [22]	24	1480

refers to a system which leaks high security input by not satisfying a noninterference property, the `Drone examples` consider a leader-follower drone scenario, and the `Asymmetric Arbiter` instances refer to arbiter implementations that do not satisfy a symmetry constraint. Specifications can be found in the full version of this paper [22].

Our first observation is that the cause candidate $\tilde{\mathcal{C}}$ can be efficiently computed thanks to the iterative computation of unsatisfiable cores (Sect. 5.1). The cause candidate provides a tight over-approximation of possible minimal causes. As expected, the runtime for finding minimal causes increases for larger counterexamples. However, as our experiments show, the overhead is manageable, because we optimize the search for all minimal causes by only considering every subset in $\tilde{\mathcal{C}}$ instead of naively going over every combination of input events (see Proposition 2). Compared to the computationally heavy task of model checking to get a counterexample, our approach incurs little additional cost, which matches our theoretical results (see Proposition 1). During our experiments, we have found that computing the candidate $\tilde{\mathcal{C}}$ first has, additionally to providing a powerful heuristic, another benefit: Even when the computation of minimal causes becomes increasingly expensive, $\tilde{\mathcal{C}}$ can serve as an intermediate result for the user. By filtering for important inputs, such as high security inputs, $\tilde{\mathcal{C}}$ already gives great insight to why the property was violated. In the asymmetric arbiter instance, for example, the input events $\langle \neg tb_secret, 3, t_0\rangle$ and $\langle tb_secret, 3, t_1\rangle$ of $\tilde{\mathcal{C}}$, which cause the violation, immediately catch the eye (c.f App. A.4 in [22]).

Comparison to HyperVis. HyperVis [48] is a tool for visualizing counterexamples returned from the HyperLTL model checker MCHyper [35]. It highlights the events in the trace that it considers responsible for the violation based on the formula and the set of traces, without considering the system model. However, violations of many relevant security policies such as observational determinism are not caused by events whose atomic propositions appear in the formula, as can be seen in our running example (see Sect. 3 and Example 2). When running the highlight function of HyperVis for the counterexample traces t_1, t_2 on `Running example`, the output events $\langle lo, 1, t_1\rangle$ and $\langle \neg lo, 1, t_2\rangle$ will be highlighted, neglecting the decisive high security input hi. Using our method additionally reveals

the input events $\langle \neg hi, 0, t_1 \rangle$ and $\langle hi, 0, t_2 \rangle$, i.e., an actual cause (see Table 1). This pattern can be observed throughout all considered instances in our experiments. For instance in the `Asymmetric arbiter` instance mentioned above, the input events causing the violation also do not occur in the formula (see App. A.5 in [22]) and thus HyperVis does not highlight this important cause for the violation.

7 Related Work

With the introduction of HyperLTL and HyperCTL* [20], temporal hyperproperties have been studied extensively: satisfiability [29,38,60], model checking [34,35,49], program repair [11], monitoring [2,10,32,67], synthesis [30], and expressiveness studies [23,37,53]. Causal analysis of hyperproperties has been studied theoretically based on counterfactual builders [40] instead of actual causality, as in our work. Explanation methods [4] exist for trace properties [5,39,41,42,70], integrated in several model checkers [14,15,19]. Minimization [54] has been studied, as well as analyzing several system traces together [9,43,65]. There exists work in explaining counterexamples for function block diagrams [51,63]. MODCHK uses a causality analysis [7] returning an over-approximation, while we provide minimal causes. Lastly, there are approaches which define actual causes for the violation of a trace property using Event Order Logic [13,56,57].

8 Conclusion

We present an explanation method for counterexamples to hyperproperties described by HyperLTL formulas. We lift Halpern and Pearl's definition of actual causality to effects described by hyperproperties and counterexamples given as sets of traces. Like the definition that inspired us, we allow modifications of the system dynamics in the counterfactual world through contingencies, and define these possible counterfactual behaviors in an automata-theoretic approach. The evaluation of our prototype implementation shows that our method is practically applicable and significantly improves the state-of-the-art in explaining counterexamples returned by a HyperLTL model checker.

References

1. Log4j vulnerabilities. https://logging.apache.org/log4j/2.x/security.html
2. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in hyperltl. In: CSF 2016. https://doi.org/10.1109/CSF.2016.24
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI 2009. http://ijcai.org/Proceedings/09/Papers/074.pdf
4. Baier, C., et al.: From verification to causality-based explications. In: ICALP 2021. https://doi.org/10.4230/LIPIcs.ICALP.2021.1

5. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL 2003. https://doi.org/10.1145/604131.604140
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Math. Struct. Comput. Sci. **21**(6), 1207–1252 (2011)
7. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 94–108. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_11
8. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Technical report 07/1, Inst. f. Form. Model. u. Verifikation, Johannes Kepler University (2007)
9. Bochot, T., Virelizier, P., Waeselynck, H., Wiels, V.: Paths to property violation: a structural approach for analyzing counter-examples. In: HASE 2010. https://doi.org/10.1109/HASE.2010.15
10. Bonakdarpour, B., Finkbeiner, B.: The complexity of monitoring hyperproperties. In: CSF 2018. https://doi.org/10.1109/CSF.2018.00019
11. Bonakdarpour, B., Finkbeiner, B.: Program repair for hyperproperties. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 423–441. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_25
12. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
13. Caltais, G., Guetlein, S.L., Leue, S.: Causality for general LTL-definable properties. In: CREST@ETAPS 2018. https://doi.org/10.4204/EPTCS.286.1
14. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**(6), 388–402 (2004)
15. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: ACM SIGSOFT Foundations of Software Engineering (2004). https://doi.org/10.1145/1029894.1029908
16. Chockler, H., Halpern, J.Y., Kupferman, O.: What causes a system to satisfy a specification? ACM Trans. Comput. Log. **9**(3), 20:1–20:26 (2008). https://doi.org/10.1145/1352582.1352588
17. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001)
18. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981. https://doi.org/10.1007/BFb0025774
19. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
20. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
22. Coenen, N., et al.: Explaining hyperproperty violations. CoRR (2022). https://doi.org/10.48550/ARXIV.2206.02074, full version with appendix
23. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: LICS 2019. https://doi.org/10.1109/LICS.2019.8785713

24. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 121–139. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_7
25. D'Argenio, P.R., Barthe, G., Biewer, S., Finkbeiner, B., Hermanns, H.: Is your software on dope? In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 83–110. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_4
26. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
27. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
28. Durumeric, Z., et al.: The matter of heartbleed. In: IMC 2014. https://doi.org/10.1145/2663716.2663755
29. Finkbeiner, B., Hahn, C.: Deciding hyperproperties. In: CONCUR 2016. https://doi.org/10.4230/LIPIcs.CONCUR.2016.13
30. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesis from hyperproperties. Acta Informatica **57**(1-2), 137–163 (2020). https://doi.org/10.1007/s00236-019-00358-2
31. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: a runtime verification tool for temporal hyperproperties. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 194–200. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_11
32. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Formal Methods Syst. Des. **54**(3), 336–363 (2019). https://doi.org/10.1007/s10703-019-00334-z
33. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 144–163. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_8
34. Finkbeiner, B., Müller, C., Seidl, H., Zalinescu, E.: Verifying security policies in multi-agent workflows with loops. In: CCS 2017. https://doi.org/10.1145/3133956.3134080
35. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3
36. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Formal Methods Syst. Des. **24**(2), 101–127 (2004). https://doi.org/10.1023/B:FORM.0000017718.28096.48
37. Finkbeiner, B., Zimmermann, M.: The first-order logic of hyperproperties. In: STACS 2017. https://doi.org/10.4230/LIPIcs.STACS.2017.30
38. Fortin, M., Kuijer, L.B., Totzke, P., Zimmermann, M.: HyperLTL satisfiability is Σ_1^1-complete, HyperCTL* satisfiability is Σ_1^2-complete. In: MFCS 2021. https://doi.org/10.4230/LIPIcs.MFCS.2021.47
39. Gössler, G., Le Métayer, D.: A general trace-based framework of logical causality. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 157–173. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07602-7_11

40. Gössler, G., Stefani, J.: Causality analysis and fault ascription in component-based systems. Theor. Comput. Sci. **837**, 158–180 (2020). https://doi.org/10.1016/j.tcs.2020.06.010
41. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. **8**(3), 229–247 (2006). https://doi.org/10.1007/s10009-005-0202-0
42. Groce, A., Kroening, D., Lerda, F.: Understanding Counterexamples with explain. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 453–456. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_35
43. Groce, A., Visser, W.: What went wrong: explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–136. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44829-2_8
44. Halpern, J.Y.: A modification of the Halpern-Pearl definition of causality. In: IJCAI 2015. http://ijcai.org/Abstract/15/427
45. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach. Part I: causes. Br. J. Philos. Sci. **56**(4), 843–887 (2005). http://www.jstor.org/stable/3541870
46. Halpern, J.Y., Pearl, J.: Causes and explanations: a structural-model approach. Part II: explanations. Br. J. Philos. Sci. **56**(4), 889–911 (2005). http://www.jstor.org/stable/3541871
47. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
48. Horak, T., et al.: Visual analysis of hyperproperties for understanding model checking results. IEEE Trans. Vis. Comput. Graph. **28**(1), 357–367 (2022). https://doi.org/10.1109/TVCG.2021.3114866
49. Hsu, T.-H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: Groote, J.F., Larsen, K.G. (eds.) TACAS 2021. LNCS, vol. 12651, pp. 94–112. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_6
50. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: a Python toolkit for prototyping with SAT oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 428–437. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_26
51. Jee, E., et al.: FbdVerifier: interactive and visual analysis of counterexample in formal verification of function block diagram. J. Res. Pract. Inf. Technol. **42**(3), 171–188 (2010)
52. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: SP 2019. https://doi.org/10.1109/SP.2019.00002
53. Krebs, A., Meier, A., Virtema, J., Zimmermann, M.: Team semantics for the specification and verification of hyperproperties. In: MFCS 2018. https://doi.org/10.4230/LIPIcs.MFCS.2018.10
54. Lahtinen, J., Launiainen, T., Heljanko, K., Ropponen, J.: Model checking methodology for large systems, faults and asynchronous behaviour: SARANA 2011 work report. No. 12 in VTT Tech., VTT Tech. Research Centre of Finland (2012)
55. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2) (1997). https://doi.org/10.1007/s100090050010
56. Leitner-Fischer, F., Leue, S.: Causality checking for complex system models. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 248–267. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_16

57. Leitner-Fischer, F., Leue, S.: Probabilistic fault tree synthesis using causality computation. Int. J. Crit. Comput. Based Syst. **4**(2), 119–143 (2013). https://doi.org/10.1504/IJCCBS.2013.056492
58. Lewis, D.: Causation. J. Philos. **70**(17), 556–567 (1973)
59. Lipp, M., et al.: Meltdown: reading kernel memory from user space. Commun. ACM **63**(6), 46–56 (2020)
60. Mascle, C., Zimmermann, M.: The keys to decidable HyperLTL satisfiability: small models or very simple formulas. In: CSL 2020. https://doi.org/10.4230/LIPIcs.CSL.2020.29
61. McCullough, D.: Noninterference and the composability of security properties. In: Proceedings. 1988 IEEE Symposium on Security and Privacy, pp. 177–186 (1988)
62. Moore, E.F.: Gedanken-experiments on sequential machines. Aut. stud. **34** (1956)
63. Pakonen, A., Buzhinsky, I., Vyatkin, V.: Counterexample visualization and explanation for function block diagrams. In: INDIN 2018. https://doi.org/10.1109/INDIN.2018.8472025
64. Pnueli, A.: The temporal logic of programs. In: FOCS 1977 (1977)
65. Schuppan, V., Biere, A.: Shortest counterexamples for symbolic model checking of LTL with past. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 493–509. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_32
66. Sörensson, N.: Minisat 2.2 and minisat++ 1.1. SAT Race 2010
67. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 406–424. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_25
68. Vardi, M.Y.: Alternating automata: unifying truth and validity checking for temporal logics. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 191–206. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63104-6_19
69. Vazquez-C., M., Rabe, M.: py-aiger. https://github.com/mvcisback/py-aiger
70. Wang, C., Yang, Z., Ivančić, 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). https://doi.org/10.1007/11901914_9

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.