A General Trace-Based Framework of Logical Causality

Gregor Gössler, Daniel Le Métayer

A General Trace-Based Framework of Logical Causality*

Gregor Gössler and Daniel Le Métayer

INRIA Grenoble – Rhône-Alpes, France

Abstract. In component-based safety-critical embedded systems it is crucial to determine the cause(s) of the violation of a safety property, be it to issue a precise alert, to steer the system into a safe state, or to determine liability of component providers. In this paper we present an approach to blame components based on a single execution trace violating a safety property P. The diagnosis relies on counterfactual reasoning ("what would have been the outcome if component C had behaved correctly?") to distinguish component failures that actually contributed to the outcome from failures that had little or no impact on the violation of P.

1 Introduction

In a concurrent, possibly embedded and distributed system, it is often crucial to determine which component(s) caused an observed failure. Understanding causality relationships between component failures and the violation of system-level properties can be especially useful to understand the occurrence of errors in execution traces, to allocate responsibilities, or to try to prevent errors (by limiting error propagation or the potential damages caused by an error).

The notion of causality inherently relies on a form of counterfactual reasoning: basically the goal is to try to answer questions such as "would event e_2 have occurred if e_1 had not occurred?" to decide if e_1 can be seen as a cause of e_2 (assuming that e_1 and e_2 have both occurred, or could both occur in a given context). But this question is not as simple as it may look:

1. First, we have to define what could have happened if e_1 had not occurred, in other words what are the *alternative worlds*.
2. In general, the set of alternative worlds is not a singleton and it is possible that in some of these worlds e_2 would occur while in others e_2 would not occur.
3. We also have to make clear what we call an event and when two events in two different traces can be considered as similar. For example, if e_1 had not occurred, even if an event potentially corresponding to e_2 might have occurred, it would probably not have occurred at the same time as e_2 in the original sequence of events; it could also possibly have occurred in a

* to appear in Formal Aspects of Component Software (FACS) 2013.

slightly different way (for example with different parameters, because of the potential effect of the occurrence of e_1 on the value of some variables).

Causality has been studied in many disciplines (philosophy, mathematical logic, physics, law, etc.) and from different points of view. In this paper, we are interested in causality for the analysis of execution traces in order to establish the origin of a system-level failure. The main trend in the use of causality in computer science consists in mapping the abstract notion of event in the general definition of causality proposed by Halpern and Pearl in their seminal contribution [12] to properties of execution traces. Halpern and Pearl's model of causality relies on a counterfactual condition mitigated by subtle contingency properties to improve the accurateness of the definition and alleviate the limitations of the counterfactual reasoning in the occurrence of multiple causes. While Halpern and Pearl's model is a very precious contribution to the analysis of the notion of causality, we believe that a fundamentally different approach considering traces as first-class citizens is required in the computer science context considered here: The model proposed by Halpern and Pearl is based on an abstract notion of event defined in terms of propositional variables and causal models expressed as sets of equations between these variables. The equations define the basic causality dependencies between variables (such as $F = L_1$ or L_2 if F is a variable denoting the occurrence of a fire and L_1 and L_2 two lightning events that can cause the fire). In order to apply this model to execution traces, it is necessary to map the abstract notion of event onto properties of execution traces. But these properties and their causality dependencies are not given a priori, they should be derived from the system under study. In addition, a key feature of trace properties is the temporal ordering of events which is also intimately related to the idea of causality but is not an explicit notion in Halpern and Pearl's framework (even if notions of time can be encoded within events). Even though this application is not impossible, as shown by [4], we believe that definitions in terms of execution traces are preferable because (a) in order to determine the responsibility of components for an observed outcome, component traces provide the relevant granularity, and (b) they can lead to more direct and clearer definitions of causality.

As suggested above, many variants of causality have been proposed in the literature and used in different disciplines. It is questionable that one single definition of causality could fit all purposes. For example, when using causality relationships to establish liabilities, it may be useful to ask different questions, such as: "could event e_2 have occurred in some cases if e_1 had not occurred?" or "would event e_2 have occurred if e_1 had occurred but not e'_1?". These questions correspond to different variants of causality which can be perfectly legitimate and useful in different situations. To address this need, we propose two definition of causality relationships that can express these kinds of variants, called *necessary* and *sufficient* causality.

The framework introduced here distinguishes a set of black-box components, each equipped with a specification. On a given execution trace, the causality of the components is analyzed with respect to the violation of a system-level

property. In order to keep the definitions as simple as possible without losing generality — that is, applicability to various models of computation and communication —, we provide a language-based formalization of the framework. We believe that our general, trace-based definitions are unique features of our framework.

Traces can be obtained from an execution of the actual system, but also as counter-examples from model-checking. For instance, we can model-check whether a behavioral model satisfies a property; causality on the counter-example can then be established against the component specifications.

2 Modeling Framework

In order to focus on the fundamental issues in defining causality on execution traces we introduce a simple, language-based modeling framework.

Definition 1 (Component signature). *A* component signature C_i *is a tuple* $(\Sigma_i, \mathcal{S}_i)$ *where* Σ_i *is an alphabet and* $\mathcal{S}_i \subseteq \Sigma_i^*$ *is a prefix-closed specification (set of allowed behaviors) over* Σ_i*.*

A component signature is the abstraction of an actual component that is needed to apply the causality analysis introduced here. Similarly, a system signature is the abstraction of a system composed of a set of interacting components.

Definition 2 (System signature). *A* system signature *is a tuple* (C, Σ, B, ρ) *where*

- $C = \{C_1, ..., C_n\}$ *is a finite set of component signatures* $C_i = (\Sigma_i, \mathcal{S}_i)$ *with pairwise disjoint alphabets;*
- $\Sigma \subseteq \Sigma_1' \times ... \times \Sigma_n'$ *is a system alphabet with* $\Sigma_i' = \Sigma_i \cup \{\epsilon\}$ *is a distinct element denoting that* C_i *does not participate in an interaction* $\alpha \in \Sigma$*;*
- $B \subseteq \Sigma^* \cup \Sigma^\omega$ *is a prefix-closed behavioral model;*
- $\rho \subseteq \left(\bigcup_i \Sigma_i\right) \times \left(\bigcup_i \Sigma_i\right)$ *is a relation modeling* information flow *among components.*

The behavioral model B is used to express assumptions and constraints on the possible (correct and incorrect) behaviors. The relation ρ models possible information flow among components. Intuitively, $(a, b) \in \rho$ means that any occurrence of a may influence the next occurrence of b (possibly in the same interaction), e.g., by triggering or constraining the occurrence of b, or by transmitting information.

Notations. Given a trace $tr = \alpha_1 \cdot \alpha_2 \cdots \in \Sigma^*$ and an index $i \in \mathbb{N}$ let $tr[1..i] = \alpha_1 \cdots \alpha_i$, let $tr[i] = \alpha_i$, and $tr[i...] = \alpha_i \alpha_{i+1} \cdots$. Let $|tr|$ denote the length of tr. For $\alpha = (a_1, ..., a_n) \in \Sigma$ let $\alpha[k] = a_k$ denote the action of component k in α ($a_k = \epsilon$ if k does not participate in α); for $w = \alpha_1 \cdots \alpha_k \in \Sigma^*$ and $i \in \{1, ..., n\}$ let $\pi_i(w) = \alpha_1[i] \cdots \alpha_k[i]$ (where ϵ letters are removed from the resulting word).

For the sake of compactness of notations we define composition $\| : \Sigma_1^* \times ... \times \Sigma_n^* \to \Sigma^*$ such that $w_1 \| ... \| w_n = \{w \in \Sigma^* \mid \forall i = 1, ..., n : \pi_i(w) = w_i\}$, and extend $\|$ to languages.

2.1 Logs

A (possibly faulty) execution of a system may not be fully observable; therefore we base our analysis on *logs*. A log of a system $S = (C, \Sigma, B, \rho)$ with components $C = \{C_1, ..., C_n\}$ of alphabets Σ_i is a vector $\boldsymbol{tr} = (tr_1, ..., tr_n) \in \Sigma_1^* \times ... \times \Sigma_n^*$ of component traces such that there exists a trace $tr \in \Sigma^*$ with $\forall i = 1, ..., n : tr_i = \pi_i(tr)$. A log $\boldsymbol{tr} \in \mathcal{L}$ is thus the projection of an actual system-level trace $tr \in B$. This relation between the actual execution and the log on which causality analysis will be performed allows us to model the fact that only a partial order between the events in tr may be observable rather than their exact precedence.[1]

Let $\mathcal{L}(S)$ denote the set of logs of S. Given a log $\boldsymbol{tr} = (tr_1, ..., tr_n) \in \mathcal{L}(S)$ let $\boldsymbol{tr}^{\uparrow} = \{tr \in B \mid \forall i = 1, ..., n : \pi_i(tr) = tr_i\}$ be the set of behaviors resulting in $\boldsymbol{tr}$.

Definition 3 (Consistent specification). *A consistently specified system is a tuple $(S, \mathcal{P})$ where $S = (C, \Sigma, B, \rho)$ is a system signature with $C = \{C_1, ..., C_n\}$ and $C_i = (\Sigma_i, \mathcal{S}_i)$, and $\mathcal{P} \subseteq B$ is a prefix-closed property such that for all traces $tr \in B$,*

$$(\forall i = 1, ..., n : \pi_i(tr) \in \mathcal{S}_i) \implies tr \in \mathcal{P}$$

Under a consistent specification, property $\mathcal{P}$ may be violated only if at least one of the components violates its specification. Throughout this paper we focus on consistent specifications.

3 Motivating Example

Consider a database system consisting of three components communicating by message passing over point-to-point FIFO buffers. Component C_1 is a client, C_2 the database server, and C_3 is a journaling system. The specifications of the three components are as follows:

$\mathcal{S}_1$: sends a lock request lock to C_2, followed by a request m to modify the locked data.

$\mathcal{S}_2$: receives a write request m, possibly preceded by a lock request lock. Access control is optimistic in the sense that the server accepts write requests without checking whether a lock request has been received before; however, in case of a missing lock request, a conflict may be detected later on and signaled by an event x. After the write, a message journ is sent to C_3.

$\mathcal{S}_3$: keeps receiving journ events from C_2 for journaling.

The system is modeled by the system signature (C, Σ, B, ρ) where $C = \{C_1, C_2, C_3\}$ with component signatures $C_i = (\Sigma_i, \mathcal{S}_i)$, and

[1] It is straight-forward to allow for additional information in traces $tr \in B$ that is not observable in the log, by adding to the cartesian product of Σ another alphabet that does not appear in the projections. For instance, events may be recorded with some timing uncertainty rather than precise time stamps [23].

- $\Sigma_1 = \{\mathsf{a}, \mathsf{m!}, \mathsf{lock!}\}$, $\Sigma_2 = \{\mathsf{m?}, \mathsf{journ!}, \mathsf{x}, \mathsf{lock?}\}$, and $\Sigma_3 = \{\mathsf{b}, \mathsf{journ?}\}$, where $\mathsf{m!}$ and $\mathsf{m?}$ stand for the emission and reception of a message m, respectively, and a, b, and x are internal events;
- $\mathcal{S}_1 = \{\mathsf{lock!.m!}\}^2$, $\mathcal{S}_2 = \{\mathsf{lock?.m?.journ!},\ \mathsf{m?.journ!.x}\}$, and $\mathcal{S}_3 = \{\mathsf{journ?}^{\mathsf{i}} \mid \mathsf{i} \in \mathbb{N}\}$;
- $\Sigma = (\Sigma_1 \times \{\epsilon\} \times \{\epsilon\}) \cup (\{\epsilon\} \times \Sigma_2 \times \{\epsilon\}) \cup (\{\epsilon\} \times \{\epsilon\} \times \Sigma_3)$: component actions interleave;
- $B = \big\{w \in \Sigma^* \cup \Sigma^\omega \mid \forall u, v : w = u.v \implies (|u|_{\mathsf{m?}} \leq |u|_{\mathsf{m!}} \wedge |u|_{\mathsf{journ?}} \leq |u|_{\mathsf{journ!}} \wedge |u|_{\mathsf{lock?}} \leq |u|_{\mathsf{lock!}} \wedge w$ respects lossless FIFO semantics$)\big\}$ (where $|u|_a$ stands for the number of occurrences of a in w): communication buffers are point-to-point FIFO queues;
- $\rho = \{(\mathsf{m!}, \mathsf{m?}), (\mathsf{journ!}, \mathsf{journ?}), (\mathsf{lock!}, \mathsf{lock?})\}$: any component may influence another component's state only by sending a message that is received by the latter.

We are interested in the global safety property $\mathcal{P} = \Sigma_{ok}^* \cup \Sigma_{ok}^\omega$ with $\Sigma_{ok} = \Sigma \setminus \{(\epsilon, \mathsf{x}, \epsilon)\}$ modeling the absence of a conflict event x. It can be seen that if all three components satisfy their specifications, x will not occur.

Figure 1 shows the log $\boldsymbol{tr} = (tr_1, tr_2, tr_3)$. In the log, tr_1 violates $\mathcal{S}_1$ at event a and tr_3 violates $\mathcal{S}_3$ at b. The dashed lines between $\mathsf{m!}$ and $\mathsf{m?}$, and between $\mathsf{journ!}$ and $\mathsf{journ?}$ stand for communications.

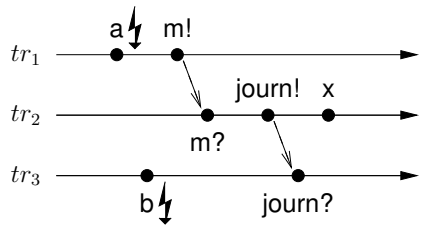


Fig. 1: A scenario with three component logs.

In order to analyze which component(s) caused the violation of $\mathcal{P}$ we can use an approach based on *counterfactual reasoning*. Informally speaking,

- C_i is a *necessary cause* for the violation of $\mathcal{P}$ if in all executions where C_i behaves correctly and all other components behave as observed, $\mathcal{P}$ is satisfied.
- Conversely, C_i is a *sufficient cause* for the violation of $\mathcal{P}$ if in all executions where all incorrect traces of components other than C_i are replaced with correct traces, and the remaining traces (i.e., correct traces and the trace of C_i) are as observed, $\mathcal{P}$ is still violated.

[2] For the sake of readability we omit the prefix closure of the specifications in the examples.

Applying these criteria to our example we obtain the following results:

If C_1 had worked correctly, it would have produced the trace $tr'_1 = \mathsf{lock!} \,.\, \mathsf{m!}$. This gives us the counterfactual scenario consisting of the traces $\boldsymbol{tr'} = (tr'_1, tr_2, tr_3)$. However, this scenario is not consistent as C_1 now emits lock, which is not received by C_2 in tr_2. According to B, the FIFO buffers are not lossy, such that lock would have been received before m if it had been sent before m. By vacuity (as no execution yielding the traces $\boldsymbol{tr'}$ exists), C_1 is a necessary cause and C_3 is a sufficient cause according to our definitions above. While the first result matches our intuition, the second result is not what we would expect. As far as C_2 is concerned, it is not a cause since its trace satisfies $\mathcal{S}_2$.

Why do the above definitions fail to capture causality? It turns out that our definition of counterfactual scenarios is too narrow, as we substitute the behavior of one component (e.g., tr_1 to analyze sufficient causality of C_3) without taking into account the impact of the new trace on the remainder of the system. When analyzing causality "by hand", one would try to evaluate the effect of the altered behavior of the first component on the other components. This is what we will formalize in the next section.

4 Causality Analysis

In this section we improve our definition of causality of component traces for the violation of a system-level property. We suppose the following inputs to be available:

- A system signature (C, Σ) with components $C_i = (C_i, \Sigma_i)$.
- A log $\boldsymbol{tr} = (tr_1, ..., tr_n)$. In the case where the behavior of two or more components is logged into a common trace, the trace of each component can be obtained by projection.
- A set $\mathcal{I} \subseteq \{1, ..., n\}$ of component indices, indicating the set of components to be jointly analyzed for causality. Being able to reason about *group causality* is useful, for instance, to determine liability of software vendors that have provided several components.

4.1 Temporal Causality

As stated in the introduction, the temporal order of the events has an obvious impact on causality relations. We use Lamport's temporal causality [17] to over-approximate the parts of a log that are impacted by component failures. This technique will allow us, in the next section, to give counterfactual definitions of causality addressing the question of "what would have been the outcome if the failure of component C had not occurred?".

Given a trace $tr \in B$ let $tr_i = \pi_i(tr)$. The trace tr is analyzed as follows, for a fixed set $\mathcal{I}$ of components to be checked.

Definition 4 (Cone of influence, $\mathcal{C}(\boldsymbol{tr}, \mathcal{I})$). *Given a consistently specified system $(S, \mathcal{P})$ with $S = (C, \Sigma, B, \rho)$, $C = \{C_1, ..., C_n\}$, and $C_i = (\Sigma_i, \mathcal{S}_i)$, a log*

$\boldsymbol{tr} \in \mathcal{L}(S)$, and a set of component indices $\mathcal{I} \subseteq \{1, ..., n\}$, let $g_i : \mathbb{N} \to \{\bot, \top\}$ be a function associating with the length of each prefix of tr_i a value in $\{\bot, \top\}$ (with $\bot < \top$). Let $(g_1^*, ..., g_n^*)$ be the least fixpoint of

$$g_i(\ell) = \begin{cases} \top \text{ if } \big(\ell = \min\{k \mid tr_i[1..k] \notin \mathcal{S}_i\} \wedge i \in \mathcal{I}\big) \vee \\ \qquad \big(\exists k < \ell : g_i(k) = \top\big) \vee \\ \qquad \big(\exists tr' \in \boldsymbol{tr}^{\uparrow}\ \exists j, k, m, n : m \leq n \wedge k = |\pi_j(tr'[1..m])| \wedge \\ \qquad\quad \ell = |\pi_i(tr'[1..n])| \wedge g_j(k) = \top \wedge (tr'[m][j], tr'[n][i]) \in \rho \wedge \\ \qquad\quad tr_i[1..\ell - 1] \in \mathcal{S}_i\big) \\ \bot \text{ otherwise} \end{cases}$$

for $i \in \{1, ..., n\}$ and $1 \leq \ell \leq |tr_i|$. Let $\mathcal{C}(\boldsymbol{tr}, \mathcal{I}) = (c_1, ..., c_n)$ such that

$$\forall i = 1, ..., n : c_i = \min\big(\{|tr_i| + 1\} \cup \{\ell \mid g_i^*(\ell) = \top\}\big)$$

The cone of influence *spanned by the components $\mathcal{I}$ is the vector of suffixes $tr_i[c_i...]$ of the component traces.*

That is, as soon as a component $i \in \mathcal{I}$ violates $\mathcal{S}_i$ on a prefix $tr_i[1..\ell]$, g_i is set to $\top$ (first line). Once $g_i(k) = \top$, it remains $\top$ for all larger indices (second line). Each time a component i participates in an interaction $\beta = tr'[n]$ for some possible trace tr' on which another component j has previously participated in an interaction $\alpha = tr'[m]$ after a prefix of length k such that $g_j(k) = \top$ and $(\alpha[j], \beta[i]) \in \rho$, then g_i is set to $\top$, provided that the prefix of tr_i satisfied $\mathcal{S}_i$ before (third line). The last condition $tr_i[1..\ell - 1] \in \mathcal{S}_i$ means that a possibly incorrect behavior of C_i following an endogenous violation of $\mathcal{S}_i$ is blamed on C_i rather than on the components in $\mathcal{I}$.

The cone of influence spanned by the components $\mathcal{I}$ is the vector of suffixes of the component traces starting with the first component action that may have been impacted by the behavior of the components $\mathcal{I}$ starting in one of their failures. For the sake of simplicity we will refer to $\mathcal{C}(\boldsymbol{tr}, \mathcal{I})$ as the cone.

Example 1. Figure 2 shows the cones $\mathcal{C}(\boldsymbol{tr}, \{1\}) = (1, 1, 3)$ and $\mathcal{C}(\boldsymbol{tr}, \{2, 3\}) = (3, 4, 1)$ for the example of Section 3 and Figure 1.

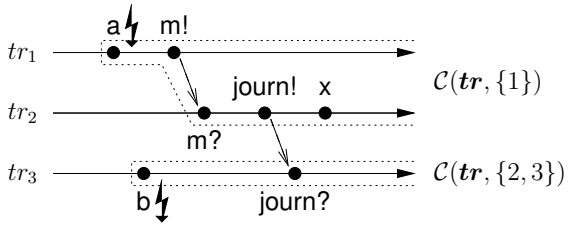


Fig. 2: The scenario with the cones $\mathcal{C}(\boldsymbol{tr}, \{1\})$ and $\mathcal{C}(\boldsymbol{tr}, \{2, 3\})$, respectively.

4.2 Logical Causality

Using the cone of influence defined above we are able to define, for a given log $\boldsymbol{tr}$ and set of component indices $\mathcal{I}$, the set of *counterfactual traces* modeling *alternative worlds* in which the failures $F_{\mathcal{I}}$ of components in $\mathcal{I}$ do not happen, and the behavior of the remaining components is as observed in $\boldsymbol{tr}$ up to the part lying inside the cone spanned by $F_{\mathcal{I}}$.

Definition 5 (Counterfactuals). *Let* $\boldsymbol{tr} = (tr_1, ..., tr_n) \in \mathcal{L}$, $\boldsymbol{\mathcal{C}} = (c_1, ..., c_n)$ *be a cone of influence, and* $\boldsymbol{\mathcal{S}} = (\mathcal{S}_1, ..., \mathcal{S}_n)$.

$$\begin{aligned}\sigma\big(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}\big) = \big\{ tr' \in B \mid \forall i : pr_i \textit{ is a prefix of } \pi_i(tr') \wedge & \quad (1)\\ (pr_i \in \mathcal{S}_i \implies \pi_i(tr') \in \mathcal{S}_i) \wedge & \quad (2)\\ (pr_i \notin \mathcal{S}_i \implies \pi_i(tr') = pr_i) \wedge & \quad (3)\\ (c_i = |tr_i| + 1 \implies \pi_i(tr') = pr_i)\big\} & \quad (4)\end{aligned}$$

where $pr_i = tr_i[1..c_i - 1]$.

Intuitively, σ returns the set of alternative behaviors $tr' \in B$ where for each component i, the prefix pr_i before entering c_i matches its logged behavior in tr_i (line 1), and if the prefix is correct and a strict prefix of tr_i then the suffix is substituted such that the whole behavior of i in trace tr' is correct (line 2); otherwise pr_i is not extended in the alternative behavior (lines 3 and 4). The rationale behind Definition 5 is to compute the set of *alternative worlds* where the failures spanning $\mathcal{C}$ do not occur. To this end we have to prune out their possible impact on the logged behavior, and substitute with correct behaviors. Prefixes violating their specifications (line 3) and component traces that never enter the cone (line 4) are not extended since we want to determine causes for system-level failures observed in the log, rather than exhibiting causality chains that are not complete yet and whose consequence would have shown only in the future.

Definition 6 (Necessary cause). *Given*

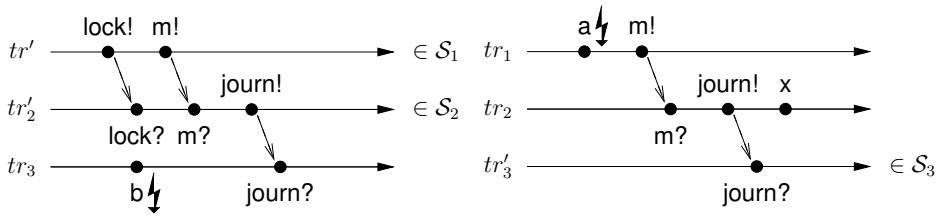
- *a consistently specified system* $(S, \mathcal{P})$ *with* $S = (C, \Sigma, B, \rho)$, $C = \{C_1, ..., C_n\}$, *and* $C_i = (\Sigma_i, \mathcal{S}_i)$,
- *a log* $\boldsymbol{tr} \in \mathcal{L}$ *such that* $\boldsymbol{tr}^{\uparrow} \cap \mathcal{P} = \emptyset$, *and*
- *an index set* $\mathcal{I}$,

let $\boldsymbol{\mathcal{C}} = \mathcal{C}(\boldsymbol{tr}, \mathcal{I})$. *The set of traces indexed by* $\mathcal{I}$ *is a necessary cause for the violation of* $\mathcal{P}$ *by* $\boldsymbol{tr}$ *if* $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) \subseteq \mathcal{P}$.

That is, the set of logs indexed by $\mathcal{I}$ is a necessary cause for the violation of $\mathcal{P}$ if in the observed behavior where the cone spanned by the incorrect behaviors of $\mathcal{I}$ is replaced by a correct behavior, $\mathcal{P}$ is satisfied. In other words, if the components in $\mathcal{I}$ had satisfied their specifications, and all components had behaved as in the logs before entering the cone, then $\mathcal{P}$ would have been satisfied.

According to the construction of the cone of influence, this definition of necessary causality makes the assumption that the violation of a component specification $\mathcal{S}_j$ within the cone of other components $\mathcal{I}$, $j \notin \mathcal{I}$, cannot be blamed for certain on component j.

Example 2. Coming back to Example 1, let $\boldsymbol{\mathcal{C}} = \mathcal{C}\big(\boldsymbol{tr}, \{1\}\big)$. We have $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) = \mathcal{S}_1 \| \mathcal{S}_2 \| \{tr_3\}$, as shown in Figure 3(a). According to Definition 6, tr_1 is a necessary cause for the violation of $\mathcal{P}$ since $\mathcal{P}$ is satisfied in $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$. It can be shown that tr_3 is not a necessary cause.



(a) $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$ with $\boldsymbol{\mathcal{C}} = \mathcal{C}\big(\boldsymbol{tr}, \{1\}\big)$ (b) $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$ with $\boldsymbol{\mathcal{C}} = \mathcal{C}\big(\boldsymbol{tr}, \{2, 3\}\big)$

Fig. 3: The scenario where the cone (a) $\mathcal{C}(\boldsymbol{tr}, \{1\})$ and (b) $\mathcal{C}(\boldsymbol{tr}, \{2, 3\})$ is substituted with suffixes satisfying the component specifications.

The definition of *sufficient causality* is dual to necessary causality, where in the alternative worlds we remove the failures of components *not in* $\mathcal{I}$ and verify whether $\mathcal{P}$ is *still violated*.

For a set of traces S, let $\sup S = \{s \in S \mid \forall t \in S : s \text{ is not a strict prefix of } t\}$.

Definition 7 (Sufficient cause). *Given*

- *a consistently specified system* $(S, \mathcal{P})$ *with* $S = (C, \Sigma, B, \rho)$, $C = \{C_1, ..., C_n\}$, *and* $C_i = (\Sigma_i, \mathcal{S}_i)$,
- *a log* $\boldsymbol{tr} \in \mathcal{L}$ *with* $\boldsymbol{tr}^{\uparrow} \cap \mathcal{P} = \emptyset$, *and*
- *an index set* $\mathcal{I}$,

let $\overline{\mathcal{I}} = \{1, ..., n\} \setminus \mathcal{I}$ *and* $\boldsymbol{\mathcal{C}} = \mathcal{C}(\boldsymbol{tr}, \overline{\mathcal{I}})$. *The set of traces indexed by* $\mathcal{I}$ *is a sufficient cause for the violation of* $\mathcal{P}$ *by* $\boldsymbol{tr}$ *if*

$$\big(\sup \sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})\big) \cap \mathcal{P} = \emptyset$$

That is, the set of logs indexed by $\mathcal{I}$ is a sufficient cause for the violation of $\mathcal{P}$ if in the observed behavior where the cone spanned by the violations of specifications by the complement of $\mathcal{I}$ is replaced by a correct behavior, the violation of $\mathcal{P}$ is inevitable (even though $\mathcal{P}$ may still be satisfied for non-maximal counterfactual traces). In other words, even if the components in the complement $\overline{\mathcal{I}}$ of $\mathcal{I}$ had satisfied their specifications and no component had failed in the cone

spanned by the failures of $\overline{\mathcal{I}}$, then $\mathcal{P}$ would still have been violated. The inclusion of infinite traces in the behavioral model B (Definition 2) ensures the least upper bound of the set of counterfactual traces to be included in B.

In Definitions 6 and 7 the use of temporal causality helps in constructing alternative scenarios in B where the components indexed by $\mathcal{I}$ (resp. $\overline{\mathcal{I}}$) behave correctly while keeping the behaviors of all other components close to their observed behaviors.

Example 3. In Example 2 let $\boldsymbol{\mathcal{C}} = \mathcal{C}\big(\boldsymbol{tr}, \{2,3\}\big)$. We obtain $\sigma\big(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}\big) = \{tr_1\} \| \{tr_2\} \| \mathcal{S}_3$, as shown in Figure 3(b). By Definition 7, tr_1 is a sufficient cause for the violation of $\mathcal{P}$ since $\mathcal{P}$ is still violated in $\sigma\big(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}\big)$. It can be shown that tr_3 is not a sufficient cause.

Properties. The following results show that our analysis does not blame any set of innocent components, and that it finds a necessary and a sufficient cause for every system-level failure.

Theorem 1 (Soundness). *Each cause contains an incorrect trace.*

Proof (sketch). Consider a set $\mathcal{I} \subseteq \{i \mid tr_i \in \mathcal{S}_i\}$. We show that the set of traces indexed by $\mathcal{I}$ is not a necessary, nor sufficient cause for the violation of $\mathcal{P}$ by $\boldsymbol{tr} = (tr_1, ..., tr_n)$.

For necessary causality, counterfactuals are computed by substituting the cone $\boldsymbol{\mathcal{C}} = \mathcal{C}(\boldsymbol{tr}, \mathcal{I})$ spanned by the failures of components in $\mathcal{I}$. If all of them satisfy their specifications, then the cone is empty, so $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) = \boldsymbol{tr}^{\uparrow}$, and $\mathcal{I}$ is not a necessary cause according to Definition 6.

For sufficient causality, counterfactuals are computed by substituting the cone $\boldsymbol{\mathcal{C}} = \mathcal{C}(\boldsymbol{tr}, \overline{\mathcal{I}}) = (c_1, ..., c_n)$ spanned by the failures of components in $\overline{\mathcal{I}}$. If all components in $\mathcal{I}$ satisfy their specifications, then $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) \subseteq \mathcal{P}$ since ρ — and thus, $\mathcal{C}(\boldsymbol{tr}, \overline{\mathcal{I}})$ — captures the possible impact of failures by components in $\overline{\mathcal{I}}$, and $(S, \mathcal{P})$ is a consistently specified system. Moreover, $\boldsymbol{\mathcal{C}}$ is constructed as a *cut* of the global execution, such that there exists a system-level trace $tr \in B$ with $\forall i : \pi_i(tr) = tr_i[1..c_i - 1]$. Therefore, $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) \neq \emptyset$. Thus, $\mathcal{I}$ is not a sufficient cause according to Definition 7. □

Theorem 2 (Completeness). *Each violation of $\mathcal{P}$ has a necessary and a sufficient cause.*

Proof (sketch). Consider a log $\boldsymbol{tr} = (tr_1, ..., tr_n)$ and let $\mathcal{I} = \{i \mid tr_i \notin \mathcal{S}_i\}$. Due to the duality of necessary and sufficient causality, the proof of completeness for necessary (resp. sufficient) causality is similar to the proof of soundness for sufficient (resp. necessary) causality:

For necessary causality, let $\boldsymbol{\mathcal{C}} = \mathcal{C}(\boldsymbol{tr}, \mathcal{I})$. We have $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) \subseteq \mathcal{P}$, thus $\mathcal{I}$ is a necessary cause for the violation of $\mathcal{P}$ by $\boldsymbol{tr}$.

For sufficient causality, let $\boldsymbol{\mathcal{C}} = \mathcal{C}(\boldsymbol{tr}, \overline{\mathcal{I}})$. By the choice of $\mathcal{I}$ this cone is empty. We thus have $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) = \boldsymbol{tr}^{\uparrow}$, thus $\sigma(\boldsymbol{tr}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) \cap \mathcal{P} = \emptyset$. It follows that $\mathcal{I}$ is a sufficient cause for the violation of $\mathcal{P}$ in $\boldsymbol{tr}$. □

5 Application to Synchronous Data Flow

In this section we use the general framework to model a synchronous data flow example, and illustrate a set of well-known phenomena studied in the literature.

Consider a simple filter that propagates, at each clock tick, the input when it is stable in the sense that it has not changed since the last tick, and holds the output when the input is unstable. Using LUSTRE [11]-like syntax the filter can be written as follows:

$$\begin{aligned} change &= false \rightarrow in \neq pre(in) \\ h &= pre(out) \\ out &= \begin{cases} in & \text{if } \neg change \\ h & \text{otherwise} \end{cases} \end{aligned}$$

That is, component *change* is initially *false*, and subsequently *true* if and only if the input *in* has changed between the last and the current tick. h latches the previous value of *out*; its value is $\perp$ ("undefined") at the first instant. *out* is equal to the input if *change* is false, and equal to h otherwise. Thus, each signal consists of an infinite sequence of values, e.g., $change = \langle change_1, change_2, ... \rangle$. A log of a valid execution is for instance

in	0	0	3	2	2
change	*false*	*false*	*true*	*true*	*false*
h	$\perp$	0	0	0	0
out	0	0	0	0	2

We formalize the system as follows.

- $\Sigma_{ch} = \mathbb{R} \times \mathbb{B} \times \mathbb{N} \times \{\mathsf{ch}\}$ where the first two components stand for the value of the input to and output from *change*, the third component is the index of the clock tick, and ch is a tag we will use to distinguish the alphabets of different components. Similarly, let $\Sigma_h = \mathbb{R} \times (\mathbb{R} \cup \{\perp\}) \times \mathbb{N} \times \{\mathsf{h}\}$ and $\Sigma_{out} = \mathbb{R} \times \mathbb{R} \times \mathbb{B} \times \mathbb{R} \times \mathbb{N} \times \{\mathsf{out}\}$.
- $\mathcal{S}_{ch} = \{(r_1, r_2, ...) \in \Sigma_{ch}^* \mid r_i = (in_i, change_i, i, \mathsf{ch}) \wedge change_1 = false \wedge (i \geq 2 \implies change_i = in_{i-1} \neq in_i)\}$ is the specification of *change*. Similarly, $\mathcal{S}_h = \{(r_1, r_2, ...) \in \Sigma_h^* \mid r_i = (out_i, h_i, i, \mathsf{h}) \wedge (i \geq 2 \implies h_i = out_{i-1})\}$ and

$$\mathcal{S}_{out} = \Big\{ (r_1, r_2, ...) \in \Sigma_{out}^* \mid r_i = (in_i, h_i, change_i, out_i, i, \mathsf{out}) \wedge out_i = \begin{cases} in_i & \text{if } \neg change_i \\ h_i & \text{otherwise} \end{cases} \Big\}$$

- $\Sigma = \{(r_{ch}, r_h, r_{out}) \in \Sigma_{ch} \times \Sigma_h \times \Sigma_{out} \mid r_{ch} = (in^{ch}, change, i_1, \mathsf{ch}) \wedge r_h = (out^h, h, i_2, \mathsf{h}) \wedge r_{out} = (in^{out}, h^{out}, ch^{out}, out, i_3, \mathsf{out}) \mid i_1 = i_2 = i_3\}$ is the system alphabet (where all components react synchronously).
- $B = \{(r_1, r_2, ...) \in \Sigma^* \cup \Sigma^\omega \mid \forall i : r_i = \big((in_i^{ch}, change_i, i_1, \mathsf{ch}), (out_i^h, h_i, i_2, \mathsf{h}), (in_i^{out}, h_i^{out}, ch_i^{out}, out_i, i_3, \mathsf{out})\big) \wedge in_i^{ch} = in_i^{out} \wedge change_i = ch_i^{out} \wedge out_i^h = out_i \wedge h_i = h_i^{out}\}$ is the set of possible behaviors, meaning that connected flows are equal.

– $\rho = \big\{\big((\cdot,\cdot,i,\mathsf{in}),(\cdot,\cdot,i,\mathsf{ch})\big),\ \big((\cdot,\cdot,i,\mathsf{in}),(\cdot,\cdot,i,\mathsf{out})\big),\ \big((\cdot,\cdot,i,\mathsf{ch}),(\cdot,\cdot,i,\mathsf{out})\big),\ \big((\cdot,\cdot,i,\mathsf{h}),(\cdot,\cdot,i,\mathsf{out})\big),\ \big((\cdot,\cdot,i,\mathsf{out}),(\cdot,\cdot,i+1,\mathsf{h})\big) \mid i \geq 1\big\}$ models the data dependencies.
– $\mathcal{P} = \{(r_1, r_2, ...) \in B \mid \forall i : r_i = \big(..., (..., out_i, ...)\big) \wedge out_i = out_{i+1} \vee out_{i+1} = out_{i+2}\}$ is the stability property, meaning that there are no two consecutive changes in output.

Figure 4 shows four logs of faulty executions (where connected signals only appear once, and the tick number and identity tags are omitted).

in	0	0	1	2
change	*false*	*false*	*false*	*false*
h	⊥	0	-1	-3
out	0	0	1	2

(a) $\boldsymbol{tr^1}$: early preemption.

in	0	0	0	0
change	*false*	*false*	*true*	*true*
h	⊥	0	-1	1
out	0	0	-1	1

(b) $\boldsymbol{tr^2}$: joint causation.

Fig. 4: Two logs of faulty executions.

Consider Figure 4a. Two components violate their specifications (incorrect values are underlined): *change* and h, both at the third instant. The stability property $\mathcal{P}$ is violated at the fourth output. Let us apply our definitions to analyze causality of each of the two faulty components.

1. In order to check whether *change* is a necessary cause, we first compute the cone spanned by the violation by *change* as $\mathcal{C}(\boldsymbol{tr^1}, \{change\}) = (3, 5, 3)$. Thus, the prefixes of the component traces before entering the cone are as shown in Figure 5a. Next we compute the set of counterfactuals, according to Definition 5, as $(\boldsymbol{tr'})^{\uparrow}$, where $\boldsymbol{tr'}$ is shown in Figure 5b. $\mathcal{P}$ is still violated by the (unique) counterfactual trace, hence *change* is not a necessary cause.

in	0	0	1	2
change	*false*	*false*		
h	⊥	0	-1	-3
out	0	0		

(a) $\boldsymbol{tr^1}$ after removing $\mathcal{C}(\boldsymbol{tr^1}, \{change\})$.

in	0	0	1	2
change	*false*	*false*	*true*	*true*
h	⊥	0	-1	-3
out	0	0	-1	-3

(b) $\boldsymbol{tr'}$ such that $(\boldsymbol{tr'})^{\uparrow} = \sigma(\boldsymbol{tr^1}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}})$

Fig. 5: Computing necessary causality of *change* for the violation of $\mathcal{P}$ in $\boldsymbol{tr^1}$.

We can show, using the same construction, that h is a sufficient cause for the violation of $\mathcal{P}$.

2. In order to check whether *change* is a sufficient cause, we first compute the cone spanned by the violation by h as $\mathcal{C}(\boldsymbol{tr^1}, \{h\}) = (5, 3, 3)$. That

is, the cone encompasses the last two values of h and *out*. Due to *change* being (incorrectly) *false*, the only possible counterfactual trace according to Definition 5 is $\sigma(\boldsymbol{tr^1}, \boldsymbol{\mathcal{C}}, \boldsymbol{\mathcal{S}}) = (tr_{change}, tr'_h, tr_{out})^{\uparrow}$ where tr_{change} is as observed in $\boldsymbol{tr^2}$, $tr'_h = (\bot, 0, 0, 1)$, and $tr'_{out} = (0, 0, 1, 2)$. $\mathcal{P}$ is still violated by the unique counterfactual trace, hence *change* is a sufficient cause.
We can show, using the same construction, that h is not a necessary cause for the violation of $\mathcal{P}$.

The example of log $\boldsymbol{tr^1}$ shows two phenomena called *over-determination* (there are two sufficient causes, one of which would have sufficed to violate $\mathcal{P}$) and *early preemption*: the causal chain from the violation of $\mathcal{S}_h$ to the violation of $\mathcal{P}$ is interrupted by the causal chain from the violation of $\mathcal{S}_{change}$ to the violation of $\mathcal{P}$, since due to *change* being *false*, the incorrect value of h is discarded in the computation of *out* in log $\boldsymbol{tr^1}$.

Figure 4b shows a case of *joint causation*: both *change* and h are necessary causes for the violation of $\mathcal{P}$ in $\boldsymbol{tr^2}$, but none of them alone is a sufficient cause.

6 Related Work

Causality has been studied for a long time in different disciplines (philosophy, mathematical logic, physics, law, etc.) before receiving an increasing attention in computer science during the last decade. Hume discusses definitions of causality in [13]:

> Suitably to this experience, therefore, we may define a cause to be an object, followed by another, and where all the objects similar to the first are followed by objects similar to the second. Or in other words where, if the first object had not been, the second never had existed.

In computer science, various approaches to causality analysis have been developed recently. They differ in their assumptions on what pieces of information are available for causality analysis: a model of causal dependencies, a program as a black-box that can be used to replay different scenarios, the observed actual behavior (e.g. execution traces, or inputs and outputs), and/or the expected behavior (that is, component specifications). Existing frameworks consider different subsets of these entities. We cite the most significant settings and approaches for these settings.

A specification and an observation. In the preliminary work of [8], causality of components for the violation of a system-level property under the BIP interaction model [9, 2] has been defined using a rudimentary definition of counterfactuals where only faulty traces are substituted but not the parts of other component traces impacted by the former. This definition suffered from the conditions for causality being true by vacuity when no consistent counterfactuals exist. A similar approach is used in [22] for causality analysis in real-time systems.

With a similar aim of independence from a specific model of computation as in our work, [21] formalizes a theory of diagnosis in first-order logic. A *diagnosis*

for an observed incorrect behavior is essentially defined as a minimal set of components forming a sufficient cause.

A causal model. [12] proposes what has become the most influential definition of causality for computer science so far, based on a model over a set of propositional variables partitioned into *exogenous* variables $\mathcal{U}$ and *endogenous* variables $\mathcal{V}$. A function $\mathcal{F}_X$ associated with each variable $X \in \mathcal{V}$ uniquely determines the value of X depending on the value of all variables in $(\mathcal{U} \cup \mathcal{V}) \setminus \{X\}$. These functions define a set of *structural equations* relating the values of the variables. The equations are required to be *recursive*, that is, the dependencies form an acyclic graph whose nodes are the variables. The observed values of a set X of variables is an *actual cause* for an observed property φ if with different values of X, φ would not hold, and there exists a context (a *contingency*) in which the observed values of X entail φ. With the objective of better representing causality in processes evolving over time, *CP-logic* defines actual causation based on probability trees [3].

In [14], fault localization and repair in a circuit with respect to an LTL property are formulated as a game between the environment choosing inputs and the system choosing a fix for a faulty component.

A model and a trace. In several applications of Halpern and Pearl's SEM, the model is used to encode and analyze one or more execution traces, rather than a behavioral model.

The definition of actual cause from [12] is used in [4] to determine potential causes for the first violation of an LTL formula by a trace. As [12] only considers a propositional setting without any temporal connectors, the trace is modeled as a matrix of propositional variables. In order to make the approach feasible in practice, an over-approximation is proposed. In this approach, the structure of the LTL formula is used as a model to determine which events may have caused the violation of the property.

Given a counter-example in model-checking, [10] uses a distance metric to determine a cause of the property violation as the difference between the error trace and a closest correct trace.

An approach to fault localization in a sequential circuit with respect to a safety specification in LTL is presented in [6]: given a counter-example trace, a propositional formula is generated that holds if a different behavior of a subset of gates entails the satisfaction of the specification.

A set of traces. [15] extends the definition of actual causality of [12] to totally ordered sequences of events, and uses this definition to construct from a set of traces a fault tree. Using a probabilistic model, the fault tree is annotated with probabilities. The accuracy of the diagnostic depends on the number of signals used to construct the model. An approach for on-the-fly causality checking is presented in [19].

An input and a black box. Delta debugging [24] is an efficient technique for automatically isolating a cause of some error. Starting from a failing input and a passing input, delta debugging finds a pair of a failing and a passing input with minimal distance. The approach is syntactical and has been applied to program code, configuration files, and context switching in schedules. By applying delta debugging to *program states* represented as *memory graphs*, analysis has been further refined to program semantics. Delta debugging isolates failure-inducing causes in the *input* of a program, and thus requires the program to be available.

7 Conclusion

We have presented a general approach for causality analysis of system failures based on component specifications and observed component traces. Applications include identification of faulty components in black-box testing, recovery of critical systems at runtime, and determination of the liability of component providers in the aftermath of a system failure.

This article opens a number of directions for future work. First of all, we will instantiate and implement the framework for specific models of computation and communication, such as Timed Automata [1] and functional programs. The tagged signal model [18] provides a formal basis for representing such models in our framework. In order to make the definitions of causality effectively verifiable, we will reformulate them as operations on symbolic models, and use efficient data structures such as the event structures used in [5] for distributed diagnosis.

At design time, the code of the components can be instrumented so as to log relevant information for analyzing causality with respect to a set of properties to be monitored. For instance, precise information on the actual (partial) order of execution can be preserved by tagging the logged events with vector clocks [7, 20]. Generally speaking, appropriate instrumentation of the code enables more precise causality analysis. We intend to further investigate this aspect of ensuring *accountability* [16] by design in future fork.

In this paper we assume only the logs to be available. However, in some situations such as post-mortem analysis the (black-box) components may be available, in which case counterfactual scenarios could be replayed on the system to evaluate their outcome more precisely. In the same vein, an alternative behavior of the control part of a closed-loop systems is likely to impact the physical process, as in our cruise control example: a counterfactual trace with different brake or throttle control will impact the speed of the car. This change should be propagated through a model of the physical process to make the counterfactual scenario as realistic as possible.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

2. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3):41–48, 2011.
3. Sander Beckers and Joost Vennekens. Counterfactual dependency and actual causation in cp-logic and structural models: a comparison. In Kristian Kersting and Marc Toussaint, editors, *STAIRS*, volume 241 of *Frontiers in Artificial Intelligence and Applications*, pages 35–46. IOS Press, 2012.
4. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R.J. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.
5. Eric Fabre, Albert Benveniste, Stefan Haar, and Claude Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15(1):33–84, 2005.
6. G. Fey, S. Staber, R. Bloem, and R. Drechsler. Automatic fault localization for property checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(6):1138–1149, 2008.
7. C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In K. Raymond, editor, *Proc. ACSC'88*, page 56 66, 1988.
8. G. Gössler, D. Le Métayer, and J.-B. Raclet. Causality analysis in contract violation. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G.J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *RV*, volume 6418 of *LNCS*, pages 270–284. Springer-Verlag, 2010.
9. G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 3 2005.
10. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
12. J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. part I: Causes. *British Journal for the Philosophy of Science*, 56(4):843–887, 2005.
13. D. Hume. *An Enquiry Concerning Human Understanding*. 1748.
14. B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460, 2012.
15. M. Kuntz, F. Leitner-Fischer, and S. Leue. From probabilistic counterexamples via causality to fault trees. In F. Flammini, S. Bologna, and V. Vittorini, editors, *SAFECOMP*, volume 6894 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2011.
16. R. Küsters, T. Truderung, and A. Vogt. Accountability: definition and relationship to verifiability. In *ACM Conference on Computer and Communications Security*, pages 526–535, 2010.
17. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.
18. E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.
19. Florian Leitner-Fischer and Stefan Leue. Causality checking for complex system models. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*, volume 7737 of *LNCS*, pages 248–267. Springer, 2013.
20. F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proc. Workshop on Parallel and Distributed Algorithms*, page 215 226. Elsevier, 1988.

21. Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.
22. S. Wang, A. Ayoub, B. Kim, G. Gössler, O. Sokolsky, and I. Lee. A causality analysis framework for component-based real-time systems. In A. Legay and S. Bensalem, editors, *Proc. Runtime Verification 2013*, volume 8174 of *LNCS*, pages 285–303. Springer, 2013.
23. Shaohui Wang, Anaheed Ayoub, Oleg Sokolsky, and Insup Lee. Runtime verification of traces under recording uncertainty. In Sarfraz Khurshid and Koushik Sen, editors, *RV*, volume 7186 of *LNCS*, pages 442–456. Springer, 2011.
24. A. Zeller. *Why Programs Fail.* Elsevier, 2009.