A general framework for blaming in component-based systems

Gregor Gössler, Daniel Le Métayer

A General Framework for Blaming in Component-Based Systems

Gregor Gössler* Daniel Le Métayer†

September 4, 2015

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 $\mathcal{P}$. 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 $\mathcal{P}$.

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). For instance, we may want to determine whether the violation of a safety requirement of a cruise control system was caused by an observed buffer overflow in component C_1 or by an observed timing failure of C_2, or by the combination of both events. 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

*INRIA Grenoble – Rhône-Alpes and Univ. Grenoble – Alpes, France
†INRIA Grenoble – Rhône-Alpes and Univ. Lyon, France

at the same time as e_2 in the original sequence of events; it could also possibly have occurred in a 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 [14] 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 [2], 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 operational 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 testing or 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.

This article extends the preliminary work of [8]. In particular, we have entirely replaced the characterization of temporal causality with the notion of unaffected prefixes (Section 5.1), which precisely distinguishes dependencies between events in the component traces on the semantic

level, and does not require the user to provide an information flow relation. In order to illustrate the instantiation of our general formalization with a specific model of computation, we apply the approach to a system whose components are specified in a simple synchronous language inspired by LUSTRE [13].

The remainder of the article is organized as follows. In the next section we discuss some fundamental issues in defining causality, and define variants of causality. In Sections 3 and 4 we introduce our language-based modeling framework and a running example. In Section 5 we formalize necessary and sufficient causality and establish some fundamental properties. Section 6 shows how the framework can be instantiated to blame components in a data-flow model *à la Lustre*. Section 7 compares our approach with related work, and Section 8 concludes.

2 Setting the Stage: Variants of Causality

Causality is a powerful but also very subtle notion, with many variants and interpretations depending on the discipline, application domain and context of use. As an illustration, legal systems introduce distinctions between actual causes, factual causes, intervening causes, intervening efficient causes, remote causes, necessary causes, probable causes, unforeseeable causes, concurrent causes, etc. This complexity is inherent to the concept of causality itself because it relies on assumptions or analyses of hypothetical actions or courses of events. Before starting the presentation of our formal framework in the next section, we first provide in this section a high-level and informal outline of a range of options for the interpretation of causality in the context of computer science.

As mentioned in the Introduction, we are interested in causality as a criterion to identify the component responsible (in a technical sense) for a failure of the system, or, more generally, for the occurrence of a given event. We assume that the minimum amount of available information to conduct the causality analysis is a set L of logs L_i containing the sequence of events observed for each component C_i of the system, a specification $\mathcal{S}_i$ for each component and a global property $\mathcal{P}$ such that $\wedge_{i\in[1,n]}\mathcal{S}_i \Rightarrow \mathcal{P}$. The set of logs L is assumed to be faulty (i.e. not to be consistent with the required property $\mathcal{P}$). The next sections show how these notions can be expressed formally in terms of signatures and traces.

In the same way as in civil law, two conditions have to be met to declare a component C_i responsible for a given (undesired) event: its behavior[1] must have been faulty and this fault must be the (or *a*) cause of the event. The first condition implies that a model of the expected (correct) behavior of the component must be available; we call this model the specification of the component in the sequel. The second condition naturally leads to another question:

> What would have been the course of events *if* C_i *had behaved correctly*?

But this question is very difficult to answer because it depends on many parameters that may be or may not be available for the analysis. A key parameter is the assumptions on the actual behaviors of the components C_i. Depending on the context, different types of information can be available:

- In some cases, no information at all is available on any component, which requires a "black box" analysis.
- In other cases, the code of each component is available and the assumption can be made that this code is actually the code that has been executed to produce the log (which is not necessarily obvious). This leads to what is sometimes called "white box" analysis.

[1] In the sequence of events leading to the undesired event.

- In yet other cases, partial information may be available, for example the code of certain components, or assumptions on the sequences of events that can or cannot be produced by a component.

In the sequel, we use BH_i to denote the assumption on the behavior of C_i: for example BH_i can be the model of the actual code in a white box analysis or the set of all potential behaviors in a black box analysis.

Another type of assumption that must be made explicit to reason about alternative behaviors[2], in order to answer the question above, concerns the consistency between individual logs, for example the fact that a message cannot be received by a component if it has not first been sent by another component. We call this assumption the behavioral model B in this paper.

Starting from this set of parameters, the general structure of a causality analysis can be pictured as follows:

$$\begin{array}{c} \textit{Observed logs } L_i \rightarrow \textit{Potential behaviors } Bh_i \\ \downarrow \\ \textit{Hypothetical logs } L'_i \leftarrow \textit{Hypothetical behaviors } Bh'_i \end{array}$$

The potential behaviors $Bh_i \in BH_i$ are the behaviors of the components that are consistent with the observed logs L_i; the hypothetical behaviors $Bh'_i \in BH_i$ are modifications of behaviors Bh_i in which certain erroneous behaviors[3] are replaced by correct behaviors; and the hypothetical logs are the logs produced by the execution of the hypothetical components. The causality analysis consists in performing these three steps and then checking whether the hypothetical logs L'_i meet the property $\mathcal{P}$[4].

This high-level picture shows that the analysis goes from logs to behaviors and back to logs: it starts from logs, tries to infer the behaviors that can have produced these logs, considers variants of these behaviors and comes back to the logs corresponding to these hypothetical behaviors. Looking at it more closely, we can see that each step in the above figure actually gives rise to a range of options:

- The first step can be interpreted as a universal or an existential quantification. In other words, we may want to consider all behaviors consistent with the observed log or just require the existence of a behavior. Universal quantification leads to notions of "strong" causality and existential quantification to "weak causality" (or potential causality).

- In the second step, different choices are possible for the components whose behavior is modified: for example, if we are interested in the responsibility of a given component C_i, we may replace the behavior of C_i by a correct behavior or replace the behavior of all components but C_i by a correct behavior. As explained below, these choices lead to two classes of causality called necessary and sufficient causality respectively.

- Just like the first step, the third step can be interpreted as a universal or an existential quantification: we may want to consider all hypothetical logs obtained from the hypothetical components or just consider the existence of hypothetical logs meeting (or not meeting) the property $\mathcal{P}$. This choice has an impact on the treatment of non-determinism in the execution of the components.

[2]We call alternative behaviors the other possible behaviors of the components in the counterfactual reasoning, which typically involves the replacement of the behavior of a component by a correct behavior.

[3]Typically the behaviors of the components which are suspected of being the causes of the failure of the system.

[4]In which case, causality will be established.

The combination of the above choices leads to eight possible forms of causality which can be noted Necessary$^{\exists,\exists}$, Necessary$^{\exists,\forall}$, ..., Sufficient$^{\exists,\exists}$, Sufficient$^{\exists,\forall}$, ... For example, Necessary$^{\forall,\forall}$ (for one component C_i) corresponds to the following informal definition:

Considering the evidence provided by the set of logs L, Component C_i is a Necessary$^{\forall,\forall}$ cause for the failure of the system if for all potential behaviors Bh of the system consistent with L, all behaviors Bh' similar to Bh except for the behavior of C_i which is made correct, lead to correct execution logs.

The next sections provide a formal model of the intuitions introduced here. In the rest of this paper, we do not make any assumption on potential behaviors (black box analysis), we consider both necessary and sufficient causality, and focus on strong forms of causality.

3 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 (Prefix $\sqsubseteq$, $\sqcap$, $\sqcup$) *A finite word w' is a prefix of w, written $w' \sqsubseteq w$, if there exists a word w'' such that $w = w' \cdot w''$, where $\cdot$ stands for concatenation. Let ϵ denote the empty word. For two words w_1 and w_2 let $w_1 \sqcap w_2$ be their longest common prefix. For a set P of prefixes of a given word let $\bigsqcap P$ and $\bigsqcup P$ denote the infimal and the supremal element of $\mathcal{P}$ with respect to $\sqsubseteq$, respectively.*

A language L is *upward-closed* if $(L, \sqsubseteq)$ is a complete partial order, that is, if for any ascending chain of words $w_1 \sqsubseteq w_2 \sqsubseteq ...$ in L, $\bigsqcup_i w_i \in L$.

Definition 2 (Component signature) *A* component signature *C_i is a tuple $(\Sigma_i, \mathcal{S}_i)$ where Σ_i is an alphabet of component actions and $\mathcal{S}_i \subseteq \Sigma_i^*$ is a prefix-closed and upward-closed language over Σ_i called* specification.

The component signature is the abstraction of an actual component. Σ_i is the alphabet of actions the actual component may produce, whereas the alphabet actually used in $\mathcal{S}_i$ may be a subset of Σ_i. Prefix closure means that $\mathcal{S}_i$ is a safety specification, while upward closure ensures the least upper bound of any ascending chain to be included in the specification.

Similarly, a system signature is the abstraction of a system composed of a set of interacting components.

Definition 3 (System signature) *A* system signature *is a tuple (C, Σ, B) 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 $\Sigma'_i = \Sigma_i \cup \{\varnothing\}$ where an interaction $\alpha = (a_1, ..., a_n) \in \Sigma$ is a tuple of simultaneous actions, and $a_i = \varnothing$ means that component C_i does not participate in α;*
- *$B \subseteq \Sigma^* \cup \Sigma^\omega$ is a prefix-closed and upward-closed behavioral model.*

The behavioral model B is used to express assumptions and constraints on the possible (correct and incorrect) behaviors. For instance, in a model of components communicating by asynchronous message passing, B may be used to express the fact that a message cannot be received before it has been sent; in a real-time model it may be used to express the hypothesis that time progresses uniformly for all components.

Notations Given a word $w = \alpha_1 \cdot \alpha_2 \cdots \in \Sigma^*$ and an index $i \in \mathbb{N}$ let $w[1..i] = \alpha_1 \cdots \alpha_i$ and let $w[i] = \alpha_i$. For $\alpha = (a_1, ..., a_n) \in \Sigma$ let $\alpha[k] = a_k$ denote the action of component k in α ($a_k = \varnothing$ if k does not participate in α); for $w = \alpha_1 \cdots \alpha_k \in \Sigma^*$ and $i \in \{1, ..., n\}$ let $\pi_i(w)$ be the word obtained by removing all $\varnothing$ letters from $\alpha_1[i] \cdots \alpha_k[i]$.

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 tuples of languages.

3.1 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 $S = (C, \Sigma, B)$ with components $C = \{C_1, ..., C_n\}$ of alphabets Σ_i is a vector $\vec{tr} = (tr_1, ..., tr_n) \in \Sigma_1^* \times ... \times \Sigma_n^*$ of component traces (i.e., words over the component alphabets) such that there exists a system-level trace (i.e., a word over the system alphabet) $tr \in B$ with $\forall i = 1, ..., n : tr_i = \pi_i(tr)$. A log $\vec{tr}$ is thus the projection of a system-level trace tr. This relation between an 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 (i.e., occurrences of component actions) in tr may be observable rather than their exact precedence[5]. Similarly, the component specifications may ignore part of the logged events. Let $\mathcal{L}(S)$ denote the set of logs of S. Given a log $\vec{tr} = (tr_1, ..., tr_n) \in \mathcal{L}(S)$ let $\vec{tr}^{\uparrow} = \{tr \in B \mid \forall i = 1, ..., n : \pi_i(tr) = tr_i\}$ be the set of behaviors resulting in $\vec{tr}$.

Definition 4 (Consistent specification) *A consistently specified system is a tuple $(S, \mathcal{P})$ where $S = (C, \Sigma, B)$ 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 safety 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.

4 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 policy violation 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 messages from C_2 for journaling, and acknowledges them with ok.

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

[5]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 [27].

- $\Sigma_1 = \{\mathsf{a}, \mathsf{m!}, \mathsf{lock!}\}$, $\Sigma_2 = \{\mathsf{m?}, \mathsf{journ!}, \mathsf{x}, \mathsf{lock?}, \mathsf{ok?}\}$, and $\Sigma_3 = \{\mathsf{b}, \mathsf{journ?}, \mathsf{ok!}\}$, where $m!$ and $m?$ stand for the emission and reception of a message m, respectively, and a, b, and x are internal actions;
- $\mathcal{S}_1 = \{\mathsf{lock!} \cdot \mathsf{m!}\}^6$, $\mathcal{S}_2 = \{\mathsf{lock?} \cdot \mathsf{m?} \cdot \mathsf{journ!} \cdot \mathsf{ok?},\ \mathsf{m?} \cdot \mathsf{journ!} \cdot \mathsf{ok?} \cdot \mathsf{x}\}$, and $\mathcal{S}_3 = \{(\mathsf{journ?} \cdot \mathsf{ok!})^{\mathsf{i}} \mid \mathsf{i} \in \mathbb{N}\}$;
- $\Sigma = (\Sigma_1 \times \{\varnothing\} \times \{\varnothing\}) \cup (\{\varnothing\} \times \Sigma_2 \times \{\varnothing\}) \cup (\{\varnothing\} \times \{\varnothing\} \times \Sigma_3)$: component actions interleave;
- $B = \big\{w \in \Sigma^* \cup \Sigma^\omega \mid \forall u \in \Sigma^*\ \forall v : w = u \cdot v \implies (|u|_{\mathsf{m?}} \leq |u|_{\mathsf{m!}} \wedge |u|_{\mathsf{journ?}} \leq |u|_{\mathsf{journ!}} \wedge |u|_{\mathsf{lock?}} \leq |u|_{\mathsf{lock!}} \wedge |u|_{\mathsf{ok?}} \leq |u|_{\mathsf{ok!}} \wedge w$ 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.

We are interested in the global safety property $\mathcal{P} = \Sigma_{ok}^* \cup \Sigma_{ok}^\omega$ with $\Sigma_{ok} = \Sigma \setminus \{(\varnothing, \mathsf{x}, \varnothing)\}$ 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 $\vec{tr} = (tr_1, tr_2, tr_3)$. 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.

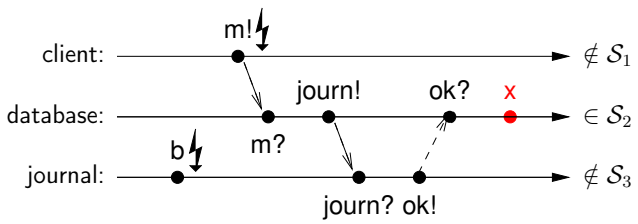


Figure 1: A scenario with three component logs.

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.

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 $\vec{tr'} = (tr_1', tr_2, tr_3)$. 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 $\vec{tr'}$ 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

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

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.

5 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 given:

- A system signature (C, Σ, B) with component signatures $C_i = (C_i, \Sigma_i)$.
- A log $\vec{tr} = (tr_1, ..., tr_n)$. 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 safety property $\mathcal{P} \subseteq B$ such that $(S, \mathcal{P})$ is consistently specified.
- 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 component providers that are responsible for more than one component.

5.1 Unaffected Prefixes

Intuitively, in order to verify whether the violations of $\mathcal{S}_i$ by tr_i, $i \in \mathcal{I}$, are a cause for the violation of $\mathcal{P}$ in $\vec{tr}$, we have to identify and remove the effect of these component failures on $\vec{tr}$, replace it with behaviors that are consistent with a correct execution of the components in $\mathcal{I}$, and verify whether all obtained counterfactual traces satisfy $\mathcal{P}$. In order to determine and eliminate the impact of component failures on the traces of the remaining components, we compute the set of prefixes that are *unaffected* by the failures. This approach has the advantage of analyzing the propagation of failures on the semantic level, in contrast to the less precise approach of [8] where the impact of component failures on other components is over-approximated using a worst-case information flow relation between component actions.

Definition 5 (Critical prefix *cp*) *Given a trace $tr = \alpha_1\alpha_2\cdots$ over Σ and a language $\mathcal{S}$ over Σ, let $cp(tr, \mathcal{S}) = \bigsqcup\{tr' \mid tr' \sqsubseteq tr \wedge tr' \in \mathcal{S}\}$ be the* critical prefix *of tr with respect to $\mathcal{S}$.*

$cp(tr, \mathcal{S})$ is the supremum of all prefixes of tr that satisfy $\mathcal{S}$. Since by definition the component specifications $\mathcal{S}_i$ are upward-closed, $cp(tr, \mathcal{S}) \in \mathcal{S}$.

Definition 6 (Trace extension *extend*) *Given a vector $\vec{\mathcal{S}}$ of specifications, let*

$$extend_i(tr^0, tr) = \begin{cases} \{tr' \in \mathcal{S}_i \mid tr \sqsubseteq tr'\} & \text{if } tr \neq tr^0 \wedge tr \in \mathcal{S}_i \\ \{tr\} & \text{otherwise} \end{cases}$$

The definition of trace extension plays a pivotal role in our definitions of causality.

Before formalizing the notion of unaffected prefixes, we need the following auxiliary definition.

Definition 7 (Least constraining components *lcc*) *Consider a language B over Σ and a log $\vec{tr}^0$. For a vector of traces $\vec{tr}$ over Σ, $w \in \Sigma^*$, and $\alpha \in \Sigma$ let*

$$cons(\vec{tr}, w, \alpha) = \left\{i \mid \pi_i(w \cdot \alpha) \sqsubseteq extend_i(tr_i^0, tr_i)\right\}$$

be the indices of components whose extension of tr_i is consistent with $w \cdot \alpha$. Let

$$lcc(\vec{tr}, L) = \sqcup\{cons(\vec{tr}, w, \alpha) \mid w \in L \wedge \alpha \in \Sigma \wedge ok(w, \alpha)\}$$

be the set of component indices that are least constraining the set of symbols with which the words of L may be extended, where $\sqcup S$ is the greatest element of S with respect to set inclusion, or $\emptyset$ if no greatest element exists, and

$$ok(w, \alpha) = w \cdot \alpha \in B \wedge \bigwedge_i \big(tr_i \sqsubseteq \pi_i(w) \implies \pi_i(w \cdot \alpha) \sqsubseteq extend_i(tr_i^0, tr_i)\big).$$

Thus, consistency is checked only over traces in B whose projections are either shorter than tr_i, or prefixes of the extensions.

Definition 8 (Unaffected prefixes UP) *Given vectors $\vec{tr}$ of traces and $\vec{\mathcal{S}}$ of specifications, and an index set $\mathcal{I}$, we define the* unaffected prefixes *of $\vec{tr}$ as follows. Let*

$$tr_i^1 = \begin{cases} cp(tr_i, \mathcal{S}_i) & \text{if } i \in \mathcal{I} \\ tr_i & \text{otherwise} \end{cases}$$

and $\forall i = 1, ..., n\ \forall j \geq 1$:

$$tr_i^{j+1} = \begin{cases} tr_i^j & \text{if } i \in lcc(\vec{tr}^j, L^j) \\ \bigsqcup\{tr_i^j \sqcap \pi_i(w) \mid w \in L^j\} & \text{otherwise} \end{cases} \quad (1)$$

where $L^j = \sup\{w \in B \mid \forall i\ \exists tr' \in extend(tr_i, tr_i^j) : \pi_i(w) \sqsubseteq tr'\}$.

Let $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I}) = (tr_1^*, ..., tr_n^*)$ with $tr_i^* = \bigsqcap_j tr_i^j$, $i = 1, ..., n$, be the vector of prefixes of $\vec{tr}$ that are unaffected by the failures of components in $\mathcal{I}$.

Intuitively, the vector of unaffected prefixes is computed by first removing the incorrect suffixes from tr_i, $i \in \mathcal{I}$, and then computing, for each component i, a decreasing sequence of prefixes tr_i^j until a fixpoint is reached. In each iteration we trim, for the set of components whose current prefixes constrain the possible extensions, the prefix to the longest trace that is the projection of some word in L^j (that is, on which all extended prefixes agree). The unaffected prefixes $(tr_1^*, ..., tr_n^*)$ to which the sequence converges, are the maximal prefixes that could also have been observed if all components in $\mathcal{I}$ had behaved correctly, whereas the suffixes $(s_1, ..., s_n)$ — such that $tr_i = tr_i^*.s_i$ — are impacted by the failures of components in $\mathcal{I}$. In the terminology of [8], the suffixes $(s_1, ..., s_n)$ define the cone of influence spanned by the failures of components in $\mathcal{I}$.

Example 1 *Coming back to the example of Section 4, the unaffected prefixes $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{client}\})$ of the database example of Figure 1 with respect to* client *are $(\epsilon,\ \epsilon,\ b \cdot journ? \cdot ok!)$. The unaffected prefixes with respect to* journal *are $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{journal}\}) = (\mathsf{m}!,\ \mathsf{m}? \cdot \mathsf{journ}! \cdot \mathsf{ok}? \cdot \mathsf{x},\ \epsilon)$.*

5.2 Counterfactuals

Using the unaffected prefixes defined above we are able to define, for a given log $\vec{tr}$ and set of component indices $\mathcal{I}$, the set of *counterfactual traces* modeling *alternative worlds* in which the failures of components in $\mathcal{I}$ do not happen, and the unaffected prefixes of the remaining components are as observed in $\vec{tr}$.

Definition 9 (Counterfactuals $\mathcal{C}$) *Given vectors $\vec{tr}$ of traces and $\vec{\mathcal{S}}$ of specifications, and an index set $\mathcal{I}$, let*

$$\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I}) = \{w \in B \mid \forall i : \pi_i(w) \in extend_i(tr_i, tr_i^*)\}$$

where $(tr_1^*, ..., tr_n^*) = \mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I})$, be the set of counterfactuals *to $\vec{tr}$.*

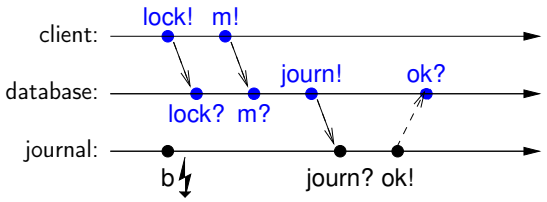
The set of counterfactuals is the set of system-level traces whose projections on the components extend the unaffected prefixes with correct behaviors. Incorrect prefixes and prefixes that amount to the whole observed trace, are not extended.

The rationale behind Definition 9 is to compute the set of *alternative worlds* where the failures of components in $\mathcal{I}$ 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, and unaffected prefixes that are equal to the observed component traces, 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.

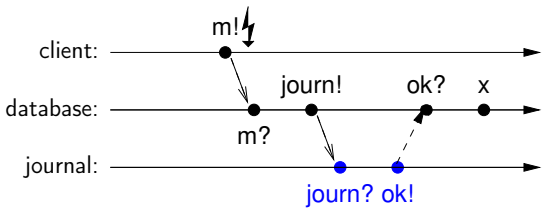
Example 2 *The set of counterfactuals $\mathcal{C}_{\mathcal{S}}(\vec{tr}, \{\mathsf{client}\})$ with respect to the failure of* client *in our running example is computed as follows (where we use the subscripts* c, db, *and* j *for* client, database, *and* journal, *respectively):*

$$\begin{aligned}\mathcal{C}_{\mathcal{S}}(\vec{tr}, \{\mathsf{client}\}) &= \{w \in B \mid \pi_{\mathsf{c}}(w) \in extend_{\mathsf{c}}(tr_{\mathsf{c}}, \epsilon) \wedge \pi_{\mathsf{db}}(w) \in extend_{\mathsf{db}}(tr_{\mathsf{db}}, \epsilon) \\ &\qquad \wedge \pi_{\mathsf{j}}(w) \in extend_{\mathsf{j}}(tr_{\mathsf{j}}, b \cdot journ? \cdot ok!)\} \\ &= \{w \in B \mid \pi_{\mathsf{c}}(w) \in \mathcal{S}_{\mathsf{c}} \wedge \pi_{\mathsf{db}}(w) \in \mathcal{S}_{\mathsf{db}} \wedge \pi_{\mathsf{j}}(w) = b \cdot journ? \cdot ok!\}\end{aligned}$$

The projections of the counterfactual traces on the three components are shown in Figure 2a. Figure 2b shows the unique counterfactual scenario with respect to the failure of journal.



(a) $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{client}\})$



(b) $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{journal}\})$

Figure 2: The counterfactual scenarios with respect to the failure of (a) client and (b) journal. Extensions of the unaffected prefixes are blue.

5.3 Logical Causality and Blaming

We are now ready to formally define two variants of causality in our framework, namely, necessary and sufficient causality.

Definition 10 (Necessary cause) *Given*

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

the incorrect suffixes of the traces indexed by $\mathcal{I}$ *are a* necessary cause *of the violation of* $\mathcal{P}$ *by* $\vec{tr}$ *if*

$$\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I}) \subseteq \mathcal{P}$$

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 unaffected prefixes are extended with correct behaviors, $\mathcal{P}$ is satisfied. In other words, if all components had behaved as in the unaffected prefixes, and the components in $\mathcal{I}$ had satisfied their specifications, then $\mathcal{P}$ would have been satisfied.

Example 3 *Coming back to our running example, we have* $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{client}\}) \subseteq \mathcal{P}$. *According to Definition 10, the failure of* client *is a necessary cause for the violation of* $\mathcal{P}$. *Since the only element of* $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{journal}\})$ *violates* $\mathcal{P}$, *the failure of* journal *is not a necessary cause for the violation of* $\mathcal{P}$.

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*.

Definition 11 (Sufficient cause) *Given*

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

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

$$\left(\sup \mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}})\right) \cap \mathcal{P} = \emptyset$$

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 incorrect suffixes of the components in the complement of $\mathcal{I}$ is replaced with 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 of influence spanned by the failures of $\overline{\mathcal{I}}$, then $\mathcal{P}$ would still have been violated.

In Definitions 10 and 11 the analysis of (in)dependence between component behaviors — represented by the unaffected prefixes — 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 4 *In our database example we have* $\left(\sup \mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{journal}\})\right) \cap \mathcal{P} = \emptyset$*. By Definition 11, the failure of* client *is a sufficient cause for the violation of* $\mathcal{P}$ *since* $\mathcal{P}$ *is still violated in the counterfactual scenario. On the other hand,* $\left(\sup \mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{\mathsf{client}\})\right) \cap \mathcal{P} \neq \emptyset$*, thus the failure of* journal *is not a sufficient cause for the violation of* $\mathcal{P}$*.*

5.4 Properties

Necessary causality is a safety property whereas checking sufficient causality amounts to verifying a liveness property on the counterfactual language. 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 1 *Consider a set* $\mathcal{I} \subseteq \{i \mid tr_i \in \mathcal{S}_i\}$ *and a log* $\vec{tr} = (tr_1, ..., tr_n)$*. We show that the set of traces indexed by* $\mathcal{I}$ *is not a necessary, nor sufficient cause for the violation of* $\mathcal{P}$ *in* $\vec{tr}$*. If all components in* $\mathcal{I}$ *satisfy their specifications, then* $(tr_1^*, ..., tr_n^*) = \mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I}) = \vec{tr}$*. By the hypothesis of Definition 10 there exists* $tr \in B$ *such that* $tr \notin \mathcal{P} \wedge \forall i : \pi_i(tr) = tr_i = tr_i^*$*. Thus* $\mathcal{I}$ *is not a necessary cause according to Definition 10.*

For sufficient causality, counterfactuals are computed by extending the unaffected prefixes $\vec{tr^*} = \mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}})$*. If all components in* $\mathcal{I}$ *satisfy their specifications, then* $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}}) \subseteq \mathcal{P}$ *since* $(S, \mathcal{P})$ *is a consistently specified system. Moreover, since the unaffected prefixes allow by construction for a common system-level trace whose projections extend them, there exists a system-level trace* $tr' \in B$ *such that* $tr' \in \mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}})$*. Thus,* $\mathcal{I}$ *is not a sufficient cause according to Definition 11.*

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

Proof 2 *Let* $\vec{tr} = (tr_1, ..., tr_n)$ *and* $\mathcal{I} = \{i \mid tr_i \notin \mathcal{S}_i\}$*. 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, the vector of unaffected prefixes is $(tr_1^*, ..., tr_n^*) = \mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I})$*. By construction of* $\vec{tr}^0$ *— and thus of* $\vec{tr}^*$ *—, all traces in* $\vec{tr}^*$ *are prefixes of the traces in* $\vec{tr}$ *and satisfy the component specifications. Since* $(S, \mathcal{P})$ *is consistently specified and hence,* $\|_{i=1}^n \mathcal{S}_i$ *satisfies* $\mathcal{P}$*, and* $\mathcal{P}$ *is prefix-closed, all traces satisfying the condition of Definition 10 also satisfy* $\mathcal{P}$*. Hence,* $\mathcal{I}$ *is a necessary cause for the violation of* $\mathcal{P}$ *in* $\vec{tr}$*.*

For sufficient causality, let $\vec{tr^*} = \mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}})$*. By the choice of* $\mathcal{I}$*,* $\vec{tr^*} = \vec{tr}$*. We thus have* $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}}) = \vec{tr}^{\uparrow}$*, thus* $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \overline{\mathcal{I}}) \cap \mathcal{P} = \emptyset$*. It follows that* $\mathcal{I}$ *is a sufficient cause for the violation of* $\mathcal{P}$ *in* $\vec{tr}$*.*

6 Application to Synchronous Data Flow

In this section we use the general framework to model a synchronous data flow example. 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 [13]-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., $change = \langle change_1, change_2, ... \rangle$.

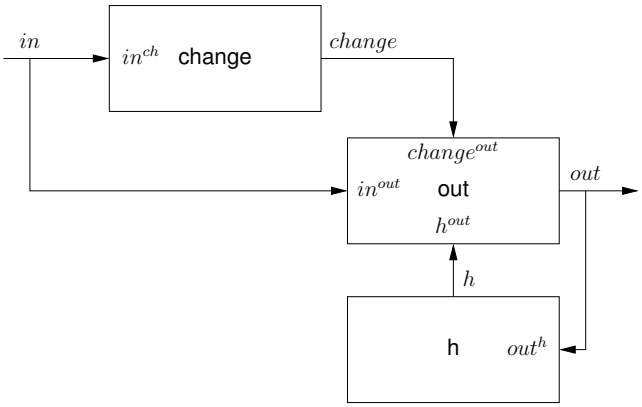


Figure 3: Architecture of the filter.

Figure 3 visualizes the architecture and signal names. We formalize the system as follows.

- $\Sigma_{ch} = \mathbb{R} \times \mathbb{B} \times \mathbb{N}$ where the first two components stand for the current value of the input to *change* and the output from *change* and the third component is the index of the clock tick. Similarly, let $\Sigma_h = \mathbb{R} \times (\mathbb{R} \cup \{\perp\}) \times \mathbb{N}$ and $\Sigma_{out} = \mathbb{R}^2 \times \mathbb{B} \times \mathbb{R} \times \mathbb{N}$. In particular, for component h the tuple only encompasses the input at the *previous* instant, which will allow us to log the values on which the specified current output depends.
- $\mathcal{S}_{ch} = \{(r_1, r_2, ...) \in \Sigma_{ch}^* \mid r_i = (in_i, change_i, i) \wedge change_1 = false \wedge (i \geq 2 \implies change_i = in_{i-1} \neq in_i)\}$ is the specification of *change*. Similarly, $\mathcal{S}_h = \{(r_1, r_2, ...) \in \Sigma_h^* \mid r_i = (out_{i-1}, h_i, i) \wedge (i \geq 2 \implies h_i = out_{i-1})\}$ and

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

- $\Sigma = \{(r_{ch}, r_h, r_{out}) \in \Sigma_{ch} \times \Sigma_h \times \Sigma_{out} \mid r_{ch} = (\cdot, \cdot, i_1) \wedge r_h = (\cdot, \cdot, i_2) \wedge r_{out} = (\cdot, \cdot, \cdot, \cdot, i_3) \mid i_1 = i_2 = i_3\}$ is the system alphabet (where all components react synchronously).
- $B = \{(r_1, r_2, ...) \in \Sigma^* \cup \Sigma^\omega \mid \forall i : r_i = \big((in_i^{ch}, change_i, i_1), (out_{i-1}^h, h_i, i_2), (in_i^{out}, h_i^{out}, ch_i^{out}, out_i, i_3)\big) \wedge in_i^{ch} = in_i^{out} \wedge change_i = ch_i^{out} \wedge out_i^h = out_i \wedge h_i = h_i^{out}\}$ is the set of possible behaviors, meaning that connected flows are equal.

in_i	0	0	3	2	2
change: $(in_i, change_i)$	(0,*false*)	(0,*false*)	(3,*true*)	(2,*true*)	(2,*false*)
h: (out_{i-1}, h_i)	$(\perp, \perp)$	(0,0)	(0,0)	(3,0)	(2,0)
out: $(in_i, change_i, h_i, out_i)$	(0,*false*,$\perp$,0)	(0,*false*,0,0)	(3,*true*,0,0)	(2,*true*,0,0)	(2,*false*,0,2)

Figure 4: A correct log of the filter.

- $\mathcal{P} = \{(r_1, r_2, ...) \in B \mid \forall i : r_i = \big(..., (..., out_i, ...)\big) \wedge out_i = out_{i+1} \vee out_{i+1} = out_{i+2}\}$ is the stability property, meaning that there are no two consecutive changes in output.

A log of a valid execution is shown in Figure 4 (where the tick number is omitted).

Figure 5a shows the logs of a faulty execution. 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 in order to analyze causality of each of the two faulty components.

- In order to check whether *change* is a necessary cause, we first compute the unaffected prefixes $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \{change\})$ with respect to the violation by *change*, as shown in Figure 5b. Next we compute the set of counterfactuals, according to Definition 9, as $(\vec{tr'})^{\uparrow}$, where $\vec{tr'}$ is shown in Figure 5c. $\mathcal{P}$ is satisfied by the (unique) counterfactual trace, hence *change* is a necessary cause. We can show, using the same construction, that h is not a sufficient cause for the violation of $\mathcal{P}$.
- In order to check whether *change* is a sufficient cause, we compute the unaffected prefixes $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \{h\})$ with respect to the violation by h, as shown in Figure 5d. Due to *change* being (incorrectly) *true*, the only possible counterfactual trace according to Definition 9 is the one shown in Figure 5e. $\mathcal{P}$ is satisfied by the unique counterfactual trace, hence *change* is not a sufficient cause. We can show, using the same construction, that h is a necessary cause for the violation of $\mathcal{P}$.

The log of Figure 5a shows a case of *joint causation*: both *change* and h are necessary causes for the violation of $\mathcal{P}$ in $\vec{tr}$.

7 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 [15]:

> 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, program code, 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.

in	0	0	0	0
change	(0,*false*)	(0,*false*)	(0,<u>*true*</u>)	(0,<u>*true*</u>)
h	($\perp$, $\perp$)	(0,0)	(0,<u>-1</u>)	(0,<u>1</u>)
out	(0,*false*,$\perp$,0)	(0,*false*,0,0)	(0,*true*,-1,-1)	(0,*true*,1,1)

(a) $\vec{tr}$: joint causation.

in	0	0	0	0
change	(0,*false*)	(0,*false*)		
h	($\perp$, $\perp$)	(0,0)	(0,<u>-1</u>)	(0,<u>1</u>)
out	(0,*false*,$\perp$,0)	(0,*false*,0,0)		

(b) $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \{change\})$

in	0	0	0	0
change	(0,*false*)	(0,*false*)	(0,*false*)	(0,*false*)
h	($\perp$, $\perp$)	(0,0)	(0,<u>-1</u>)	(0,0)
out	(0,*false*,$\perp$,0)	(0,*false*,0,0)	(0,*false*,-1,0)	(0,*false*,0,0)

(c) $\vec{tr'}$ such that $(\vec{tr'})^{\uparrow} = \mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{change\}$

in	0	0	0	0
change	(0,*false*)	(0,*false*)	(0,<u>*true*</u>)	(0,<u>*true*</u>)
h	($\perp$, $\perp$)	(0,0)		
out	(0,*false*,$\perp$,0)	(0,*false*,0,0)		

(d) $\mathsf{UP}_{\vec{\mathcal{S}}}(\vec{tr}, \{h\})$

in	0	0	0	0
change	(0,*false*)	(0,*false*)	(0,<u>*true*</u>)	(0,<u>*true*</u>)
h	($\perp$, $\perp$)	(0,0)	(0,0)	(0,0)
out	(0,*false*,$\perp$,0)	(0,*false*,0,0)	(0,*true*,0,0)	(0,*true*,0,0)

(e) $\vec{tr'}$ such that $(\vec{tr'})^{\uparrow} = \mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \{h\}$

Figure 5: Blaming in the scenario of joint causation.

A specification and an observation In [9], causality of components for the violation of a system-level property under the BIP interaction model [10] has been defined using a rudimentary definition of counterfactuals where only faulty traces are substituted but not their effects on the traces of other components. This definition suffers from the conditions for causality being true by vacuity when no consistent counterfactuals exist. A slightly improved approach is used in [26] for blaming in real-time systems. A preliminary version of our formalization presented here is instantiated in [11] to analyze necessary causality on real-time systems whose component specifications and expected system-level property are modeled as timed automata.

With a similar aim of independence from a specific model of computation as in our work, [24] 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 whose failure explains the observation.

A causal model [14] 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 [1].

In [16], 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 or program 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 [14] is used in [2] to determine potential causes for the first violation of an LTL formula by a trace. As [14] 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, [12] 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.

For a program and a set of observations that violate a specification, automatic error localization uses SMT to identify a minimal set of program statements that must be changed in order to satisfy the specification [17, 18]. Different approaches for blaming based on contracts in sequential functional languages are discussed in [4].

A set of traces [19] extends the definition of actual causality of [14] 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 traces used to construct the model. An approach for on-the-fly causality checking is presented in [22].

An input and a black box Delta debugging [28] 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.

8 Conclusion

In this article we have developed a general framework for causality analysis of system failures. 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.

For the sake of simplicity and generality we have provided a low-level formalization of blaming. The tagged signal model [21] may be used as a formal basis for representing specific models of communication in our approach. As analyzing necessary (resp. sufficient) causality amounts to verifying a safety (resp. liveness) property on the possibly infinite language $\mathcal{C}_{\vec{\mathcal{S}}}(\vec{tr}, \mathcal{I})$, blaming is undecidable in general. In order to make the definitions of causality effectively verifiable and automatize the analysis, 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. Previous versions of our technique have been instantiated for a subset of the BIP component framework [8] and networks of timed automata [11], implemented in a prototype tool called LOCA (*Logical Causality Analyzer*), and tested on several case studies.

In closed-loop control systems, an alternative (counterfactual) behavior of the controller is likely to impact the physical process. For instance, when analyzing causality in a cruise control system, a counterfactual trace with different brake or throttle control will impact the speed of the car. Therefore the model of computation has to be expressive enough to include a faithful model of the physical environment in the system.

The presented approach is not a push-button solution for blaming. For instance, in the case of two component failures f_1 and f_2 where f_2 does not lie within the unaffected prefixes of f_1, our framework lacks information to decide whether f_2 was entailed by f_1, or occurred independently. Future work will refine the approach by taking additional available pieces of information into account. For example, 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.

Going a step further, we intend to investigate how to ensure *accountability* [20] by construction, that is, designing systems in such a way that, under some hypotheses, causes for system-level failures can be determined without ambiguity. To this end, the code of the components should 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, 23]. Whenever component failures are not observable, we have to use fault diagnosis [25] before performing causality analysis. Similar to the approach of [3] to derive logging requirements from a privacy policy that produce minimal but sufficient logs for auditing the policy, an interesting work direction will be to study how to automatically determine from the system signature, a minimal logging requirement for blaming.

References

[1] 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.

[2] 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.

[3] Debmalya Biswas and Valtteri Niemi. Transforming privacy policies to auditing specifications. In Taghi M. Khoshgoftaar, editor, *HASE*, pages 368–375. IEEE Computer Society, 2011.

[4] Ch. Dimoulas, R.B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*, pages 215–226, 2011.

[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 and D. Le Métayer. A general trace-based framework of logical causality. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS - 10th International Symposium on Formal Aspects of Component Software - 2013*, volume 8348 of *LNCS*, pages 157–173. Springer, 2013.

[9] 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.

[10] G. Gössler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, 55(1-3):161–183, 3 2005.

[11] Gregor Gössler and Lăcrămioara Aştefănoaei. Blaming in component-based real-time systems. In *EMSOFT'14*, pages 7:1–7:10. ACM, 2014.

[12] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.

[13] 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.

[14] 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.

[15] D. Hume. *An Enquiry Concerning Human Understanding.* 1748.

[16] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *J. Comput. Syst. Sci.*, 78(2):441–460, 2012.

[17] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In Mary W. Hall and David A. Padua, editors, *PLDI'11*, pages 437–446. ACM, 2011.

[18] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In Per Bjesse and Anna Slobodová, editors, *FMCAD'11*, pages 91–100. FMCAD Inc., 2011.

[19] 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 *LNCS*, pages 71–84. Springer, 2011.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[25] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

[26] 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.

[27] 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.

[28] A. Zeller. *Why Programs Fail.* Elsevier, 2009.