Science of Computer Programming 113 (2015) 223–235

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# A general framework for blaming in component-based systems

Gregor Gössler [a,b], Daniel Le Métayer [a,c]

[a] INRIA, France
[b] Univ. Grenoble Alpes, France
[c] Univ. of Lyon, France

ARTICLE INFO

*Article history:*
Received 30 June 2014
Received in revised form 30 January 2015
Accepted 29 June 2015
Available online 3 July 2015

*Keywords:*
Causality
Failure
Log
Counterfactual analysis

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 at the same time as $e_2$ in the original sequence of events; it could also possibly

*E-mail address:* gregor.goessler@inria.fr (G. Gössler).

http://dx.doi.org/10.1016/j.scico.2015.06.010
0167-6423/

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 [1] 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 [3]. 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 [4].

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