9-2015

A Hybrid Approach to Causality Analysis

Shaohui Wang
University of Pennsylvania, shaohui@seas.upenn.edu

Yoann Geoffroy

Gregor Gössler

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Recommended Citation

Shaohui Wang, Yoann Geoffroy, Gregor Gössler, Oleg Sokolsky, and Insup Lee, "A Hybrid Approach to Causality Analysis", , 250-265. September 2015. http://dx.doi.org/10.1007/978-3-319-23820-3_16

6th International Conference on Runtime Verification (RV 2015), Vienna, Austria, September 22 – 25, 2015.

A Hybrid Approach to Causality Analysis

Abstract

In component-based safety-critical systems, when a system safety property is violated, it is necessary to analyze which components are the cause. Given a system execution trace that exhibits component faults leading to a property violation, our causality analysis formalizes a notion of counterfactual reasoning (\what would the system behavior be if a component had been correct?") and algorithmically derives such alternative system behaviors, without re-executing the system itself. In this paper, we show that we can improve precision of the analysis if 1) we can emulate execution of components instead of relying on their contracts, and 2) take into consideration input/output dependencies between components to avoid blaming components for faults induced by other components. We demonstrate the utility of the extended analysis with a case study for a closed-loop patient-controlled analgesia system.

Disciplines

Computer Engineering | Computer Sciences

Comments

6th International Conference on Runtime Verification (RV 2015), Vienna, Austria, September 22 – 25, 2015.

A Hybrid Approach to Causality Analysis***

Shaohui Wang[1], Yoann Geoffroy[2],
Gregor Gössler[2], Oleg Sokolsky[1], and Insup Lee[1]

[1] Department of Computer and Information Science
University of Pennsylvania
shaohui@seas.upenn.edu,{sokolsky,lee}@cis.upenn.edu
[2] INRIA Grenoble – Rhône-Alpes, France
{yoann.geoffroy,gregor.goessler}@inria.fr

Abstract. In component-based safety-critical systems, when a system safety property is violated, it is necessary to analyze which components are the cause. Given a system execution trace that exhibits component faults leading to a property violation, our causality analysis formalizes a notion of counterfactual reasoning ("what would the system behavior be if a component had been correct?") and algorithmically derives such alternative system behaviors, without re-executing the system itself. In this paper, we show that we can improve precision of the analysis if 1) we can emulate execution of components instead of relying on their contracts, and 2) take into consideration input/output dependencies between components to avoid blaming components for faults induced by other components. We demonstrate the utility of the extended analysis with a case study for a closed-loop patient-controlled analgesia system.

1 Introduction

A key idea in systems engineering is that complex systems are built by assembling components. Component-based systems are desirable because they allow independent development of system components by different suppliers, as well as their incremental construction and modification. The down side of component-based development is that no single entity – neither the integrator, nor component suppliers – have a complete understanding of component behaviors and possible interactions between them. This incomplete knowledge, in turn, requires us to resort to black-box analysis methods, when only the input-output behavior of a component is specified.

In this work, we are interested in the forensic analysis of a component-based system following the discovered violation of system safety properties. Diagnosis of the root cause is crucial for the subsequent recovery and follow-up prevention

* Research is supported in part by grants NSF CNS-1035715, IIS-1231547, ACI-1239324, and INRIA associate team Causalysis.

** This version of the paper corrects a typo in Definition 9 that was introduced in the original published version.

measures. Such diagnosis requires recording of system executions leading to the failure, as well as methods for the efficient analysis of the recorded data.

There has been a great amount of research following the seminal work of [5] and [16] in the study of fault diagnosis. In our previous work in [9,18,8], we took a step further and considered the problem of causality analysis for component-based systems. We formalized counterfactual reasoning ("what would the system execution be should a component have behaved correctly?") as a basis for the analysis. Our analysis provided a plausible explanation to how the component faults had contributed to the system property violation.

Specifically, we proposed a general causality analysis framework in [18,8], and identified four major steps in causality analysis. First, the set $\mathcal{F}$ of all faulty components with respect to their corresponding component properties are identified. Second, the set of possible counterfactual behaviors for a suspected subset $\mathcal{S} \subseteq \mathcal{F}$ is constructed. Third, based on our formalization of causality, it is determined whether the suspected subset $\mathcal{S}$ is the culprit. Lastly, minimal culprits are determined based on the results from the third step. The causality analysis we proposed in [18] assumes that the only information available to the analysis are the system definition (system property, component properties, and system topology) and a single system execution trace on which the system property is violated. It was assumed that we cannot re-run the system with some of the component faults removed, risking another failure if the true culprit was not corrected. This assumption limits precision of the analysis, as little additional information can be obtained from the system itself.

We show in this paper that we can improve the analysis without relaxing this assumption for the whole system, if some components of the system—which we refer to as *separable components*—can be run in isolation from the rest of the system to assist in counterfactual trace generation. The use of separable components during analysis phase provides a hybrid way to construct counterfactual traces that combines component traces generated statically, based on the system definition and the observed trace, and dynamic traces of separable components, containing actual outputs of the component on inputs generated during the analysis.

Another piece of under-utilized information in determining causes is the relation in between component property violations, i.e., horizontal causes [9]. For instance, when components A and B together are determined to be a cause for system property violation, by investigating and concluding that component A's fault is the cause of that of component B's, we can exclude B from blame.

To evaluate these extensions, we applied the proposed causality analysis to a patient controlled analgesia (PCA) infusion pump case study from the medical device domain. A post-surgery patient can request pain relief medication by pressing a button on the PCA pump. A pump controller monitors patient state using readings from a pulse-oximeter. To avoid potential overdose, the controller issues tickets to the pump that can limit its ability to respond to repeated patient requests. Errors in computing, delivering, and processing tickets can lead to an overdose, and the faulty component needs to be determined. Results from the

case study show that our proposed causality analysis can provide a more fine-grained analysis than our previous approach.

The contribution of this paper is a new algorithm for counterfactual reasoning that incorporates the two extensions to our existing approach, namely the use of separable components and horizontal causality. Evaluation of the new approach shows that the analysis becomes more intuitive and precise. To the best of our knowledge, the proposed approach is also the first to have incorporated both static and run-time analysis for causality analysis. On the application side, we have demonstrated the applicability of the approach to a case study in the safety-critical systems domain.

2 The PCA Example

Patient controlled analgesia (PCA) infusion pumps are used for post-surgical pain treatment in an intensive care unit (ICU). A potential hazard for the patient is overdosing. When continuously in infusion, the patient vital signs, e.g. blood oxygen saturation level (SpO2), would gradually decrease. It is considered a critical condition for the patient when SpO2 drops below 70%. To prevent overdosing, smart PCA pumps are usually equipped with a controller which reads measurements of patient vital signs and issues to PCA pumps *tickets*, i.e., maximum duration allowed for infusion given the patient vital sign readings. Figure 1 shows a simplified schematic view of the scenario, adapted from a system modeled out of a real component-based system in the clinical setting, presented in [1].

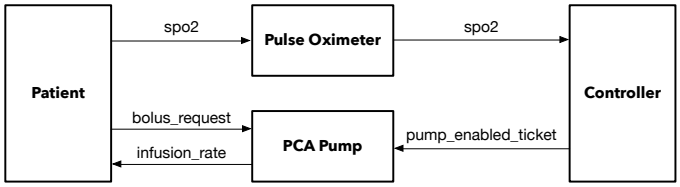

Fig. 1. Schematic View of PCA Case Study

System and Components. The PCA system consists of four components: the Patient, the pulse-oximeter (PO), the controller (Ctrl), and the PCA pump. The system level safety property is that Patient SpO2 never drops below 70%.

The Patient component is simulated based on given patient physiological reactions when infusion is and is not in progress. The differential equations describing the dynamics are given in [1]. For the purpose of this case study,

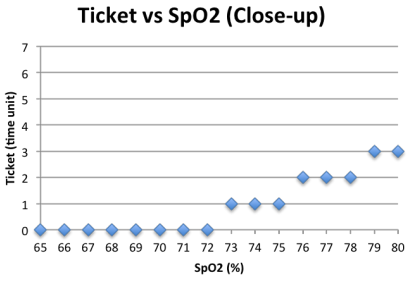

Fig. 2. Ticket vs SpO2

Snapshot	Patient.SpO2	PO.SpO2	Controller.Ticket	Patient.BR	PCA.IR
0.	71	75	3	true	400
1.	70	72	0	false	400
2.	69				

Table 1. Example Faulty Trace for PCA Case Study

it suffices to know that patient SpO2 value
would gradually decrease (resp., increase) while infusion is in progress (resp., stopped).

PO measures the Patient SpO2 and transmits the measurements to Ctrl.

Ctrl reads the patient SpO2, computes the ticket, and outputs the calculated ticket. The ticket-SpO2 relation given in [1] is pre-calculated and shown in Figure 2 with values for SpO2 in the range 65%–80%.

The PCA component takes patient bolus request and the ticket as input. When there is a patient bolus request, the PCA delivers (a) no drug if ticket is 0; (b) one time unit of infusion if ticket is 1; and (c) two consecutive time units of infusion if ticket is greater than 1.

Traces. An execution of the PCA system with manually injected faults is recorded as a system trace, i.e., a sequence of snapshots of variable values of components' input and output. We use a tabular form to represent a trace illustrated in Table 1.

Example of Causality Analysis Problem. On the trace in Table 1, in Snapshot 0, PO is faulty in measuring the value of the patient SpO2 in that the measured value (75%) is larger than the actual value (71%). Ctrl takes the wrong value as input, and makes a faulty computation as well (by outputting the ticket value 3 instead of the expected 1 as shown in Figure 2). The PCA receives a patient bolus request and reads the ticket value to be 3, so it initiates infusion for two time units consecutively until in Snapshot 2 where the patient SpO2 drops below the critical value 70%—a patient adverse event represented as a system level property violation occurs.

The causality analysis problem aims to study, in component-based systems where multiple components have committed faults in a given system execution so that system level property violation occurs, which subsets of component faults are the culprits for the system level property violation.

3 Definition of the Causality Analysis Problem

Preliminaries. A *type* T is a set of values. A *typed variable* $x : T$ is a variable with values in T. We only consider finite types in this work. A *snapshot* s_X for a set $X = \{x_1, \ldots, x_m\}$ of typed variables $x_1 : T_1, \ldots, x_m : T_m$ is an assignment of each variable x_i to its value $s_X(x_i)$ in T_i, for $1 \leq i \leq m$. A *trace* $Tr = s_0, s_1, \ldots$ for a set X of typed variables is a sequence of snapshots, where every snapshot s_i is for X. The snapshot at location i on a trace Tr is denoted $Tr[i]$. The suffix of Tr starting at location i is denoted $Tr[i...]$. A segment of Tr

starting from location i and ending at location j (inclusive) is denoted $Tr[i..j]$. For convenience, we denote by $|Tr[i..j]|$ the length of $Tr[i..j]$, i.e., the number of snapshots on $Tr[i..j]$. A set $\mathbf{TR}$ of traces is called *prefix-closed* if $\forall Tr \in \mathbf{TR}.\, \forall l < |Tr|.\, Tr[0..l] \in \mathbf{TR}$. We denote $\mathbf{T}_X$ the set of all possible traces over a set X.[1] The *projection* $\pi_Y(s)$ of a snapshot s for X on to a subset $Y \subseteq X$ is a snapshot for domain Y such that $\forall y \in Y.\big(\pi_Y(s)\big)(y) = s(y)$. The *projection* $\pi_Y(Tr)$ of a trace $Tr = s_0, s_1, \ldots$ on to a subset $Y \subseteq X$ is a trace for Y, defined as $\pi_Y(Tr) = \pi_Y(s_0), \pi_Y(s_1), \ldots$. We write $\pi_v(\cdot)$ for $\pi_{\{v\}}(\cdot)$.

3.1 System Definition

Definition 1 (Component signature)**.** *A* component signature *is a tuple* $C = \langle I, O, \mathbf{P}_C \rangle$*, where* I *and* O *are disjoint, and*

- $I = \{i_1 : T_1^i, \ldots, i_m : T_m^i\}$ *is a set of typed variables called the* input*,*
- $O = \{o_1 : T_1^o, \ldots, o_n : T_n^o\}$ *is a set of typed variables called the* output*, and*
- $\mathbf{P}_C \subseteq \mathbf{T}_{I \cup O}$ *is a prefix-closed set of traces called the* component property*.*

A component signature describes all knowledge of the component available to causality analysis. The component property can be characterized by formal languages such as regular expressions, first-order logic, temporal logics, etc. Here we take a model-theoretic view and identify the property and the set of traces that satisfy the property. By $Tr_C \subseteq \mathbf{T}_{I \cup O}$ we denote the set of traces that a component may exhibit during its execution. Note that the relationship between Tr_C and $\mathbf{P}_C$ is not known *a priori*. As we will see below, we have $Tr_C \subseteq \mathbf{P}_C$ in a correct (non-faulty) component.

Definition 2 (Channel and connection)**.** *A* channel $c = (x, y)$ *is a pair of typed variables* $(x : T_x, y : T_y)$ *such that* $T_x \subseteq T_y$*. A* connection θ *is a set of channels.*

We make the following assumptions on component composition: (a) Fan-in connections are not allowed, i.e., it is required that $\forall (x_1, y_1), (x_2, y_2) \in \theta.y_1 = y_2 \rightarrow x_1 = x_2$. (b) Variable name clashes are resolved by associating them with the component names, as is common in component-oriented languages. (c) Channels are reliable. A value passed into a channel will be successfully received by the connected component.

Definition 3 (Composition of components)**.** *Let* $\mathcal{A} = \{C_1, \ldots, C_J\}$ *be a set of* J *components with disjoint sets of variables, where* $C_j = \langle I_j, O_j, \mathbf{P}_j \rangle$ *for* $1 \leq j \leq J$*. Let* θ *be a connection where* $\forall (x, y) \in \theta. \exists j, k \in \{1, \ldots, J\}. x \in O_j \wedge y \in I_k$*. The* composition of components $C_1, \ldots, C_J$*, denoted* $C_1 \| \ldots \| C_J$*, is defined as a component* $A = \langle I, O, \mathbf{P} \rangle$*, where*

- $I = \big(\bigcup_{j=1}^{J} I_j\big) \setminus \{y \mid \exists x. (x, y) \in \theta\}$ *and* $O = \bigcup_{j=1}^{J} O_j$*, and*
- $\mathbf{P} = \big\{Tr = s_0 s_1 \ldots s_\ell \in \mathbf{T}_{I \cup O} \mid \forall 1 \leq j \leq J. \pi_j(Tr) \in \mathbf{P}_j \wedge \forall (x, y) \in \theta\ \forall k \in [0, \ell]. s_k(x) = s_k(y)\big\}$.

[1] Throughout the paper we use bold font to represent a set of traces (e.g., $\mathbf{TR}$, $\mathbf{T}_X$) or a property (e.g., $\mathbf{P}$) and calligraphic font to represent a set of components (e.g., $\mathcal{A}$ in Definition 3).

Definition 4 (System)**.** *A system* $Sys = \langle \mathcal{A}, \theta, \mathbf{P} \rangle$ *is a tuple where* Sys *is composed of components in* $\mathcal{A} = \{C_1, \ldots, C_J\}$ *by connection* θ*, and* $\mathbf{P}$ *is a prefix-closed superset of* $\mathbf{P}_{C_1 \| \ldots \| C_J}$ *called the* (safety) property *for system* Sys*.*

The system property $\mathbf{P}$ may contain more behaviors than $\mathbf{P}_{C_1 \| \ldots \| C_J}$. This means the composition of $C_1, \ldots, C_J$ is essentially a *refinement* of the system property. This is also an equivalent assumption as in [8,18] which stipulates that when a system property violation occurs, there must be at least one component property violation among $C_1, \ldots, C_J$.

Definition 5 (System trace)**.** *A* system trace Tr *for* $Sys = \langle \mathcal{A}, \theta, \mathbf{P} \rangle$ *composed of components* $\mathcal{C} = \{C_1, \ldots, C_J\}$ *is a trace where each snapshot is for the set of all components' input and output,* $\bigcup_{j=1}^{J} (I_j \cup O_j)$*.*

We assume in this work that a violation of a property $\mathbf{P}$ on a trace Tr can be detected and that the minimal prefix of Tr that violates $\mathbf{P}$ can be determined, i.e., if $Tr \notin \mathbf{P}$, then $i_{Tr,\mathbf{P}} = \min\{i \mid Tr[0..i] \notin \mathbf{P}\}$ is well defined. Our formalization of component and system property violation naturally aligns respectively with the definitions of fault and failure: a component property violation is a manifestation of a fault, whereas a system property violation is a failure.

Definition 6 (Faults and failures)**.** *Given a system* Sys *and a system trace* Tr *for* Sys*, a component fault (system failure, resp.) on* Tr *is a violation of the component (system, resp.) property.*

3.2 Causality Definitions

We state the causality analysis for component-based faulty tolerant systems as follows. Given

- a system definition $Sys = \langle \mathcal{A}, \theta, \mathbf{P} \rangle$,
- a trace Tr for Sys on which the system property is violated, and
- a causality definition CD,

determine the minimal subsets of faulty components that are causes for the system property violation, with respect to a given causality definition CD.

Reasoning for causality is based on counterfactuals. For instance, to establish that event e_1 is a necessary cause of event e_2, we consider whether the event e_2 would happen if event e_1 does not occur in any alternative system execution.

Here, an alternative system execution when certain system events are changed is called a *counterfactual trace*, whereas the observed system execution is called the *actual trace*. The key to reasoning about causality is to construct the set of all possible counterfactual traces.

In an abstract level, the set of counterfactual traces can be viewed as a function on Tr, Sys, and $\mathcal{S}$. We use the notation $\sigma(Tr, Sys, \mathcal{S})$ to represent the reconstructed sets of traces. Note that on those traces, the property violations for components in $\mathcal{S}$ are corrected. For simplicity we only consider necessary causality in this paper, but we note that the formalism can be used to express other notions of causality, such as sufficient causality.

In addition, we can distinguish vertical and horizontal causality. Vertical causality refers to the causal relationship between component faults and system

failures, whereas horizontal causality refers to the causal relationship between component faults.

Definition 7 (Necessary vertical cause)**.** *Given a system definition Sys and a trace Tr for Sys, let $\mathcal{F} = \{C \in Sys \mid \pi_C(Tr) \notin \mathbf{P}_C\}$ be the set of faulty components on trace Tr and $\mathcal{S} \subseteq \mathcal{F}$ be a non-empty suspected subset of faulty components. The component property violation in $\mathcal{S}$ is a* necessary vertical cause *for system property violation on trace Tr if and only if $\forall Tr' \in \sigma(Tr, Sys, \mathcal{S}).Tr' \in \mathbf{P}$.*

Definition 8 (Necessary horizontal cause)**.** *Let $\mathcal{F}$ and $\mathcal{S}$ be as in Definition 7, and let $C \in \mathcal{F} \backslash \mathcal{S}$ be another faulty component. The component property violation in $\mathcal{S}$ is a* necessary horizontal cause *for the property violation of component C on trace Tr if and only if $\forall Tr' \in \sigma(Tr, Sys, \mathcal{S}).\pi_C(Tr') \in \mathbf{P}_C$.*

4 Approach

In this work, we propose two extensions to our existing causality analysis framework in [18,19], illustrated in Figure 3, one using *separable components* and the other using *horizontal causality*. The four steps involved in a causality analysis are briefly discussed below, whereas using separable components and horizontal causality intervene in Step 2 (*Trace Reconstruction*) and Step 4 (*Culprit Minimization*), respectively.

Step 1. *Offline Analysis & Powerset Construction.* We first determine the set $\mathcal{F}$ as in Definition 7, and construct the powerset $2^{\mathcal{F}}$ of $\mathcal{F}$.

Step 2. *Trace Reconstruction.* For each subset $\mathcal{S} \in 2^{\mathcal{F}} \setminus \emptyset$, called a *suspect*, we construct $\sigma(Tr, Sys, \mathcal{S})$.

Step 3. *Causality Analysis & Collecting Causes.* Based on $\sigma(Tr, Sys, \mathcal{S})$ we check whether $\mathcal{S}$ is a cause according to the causality definition CD. If yes,

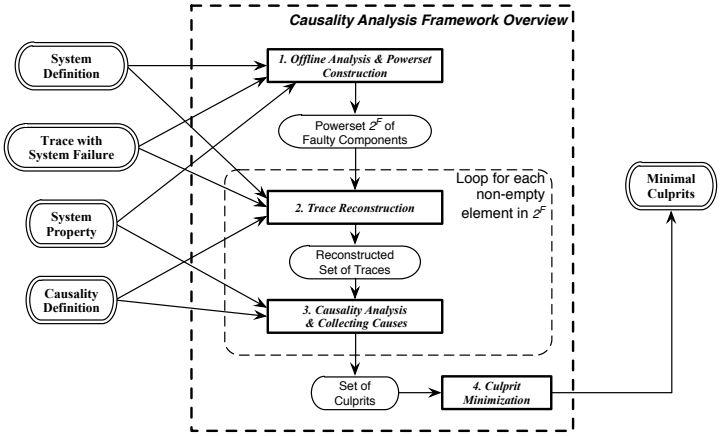

Fig. 3. Causality Analysis Framework Overview

$\mathcal{S}$ is a *culprit* and is collected for the subsequent culprit minimization; otherwise $\mathcal{S}$ is not a cause for the violation of system property $\mathbf{P}$.

Step 4. *Culprit Minimization.* The last step of causality analysis is to check the minimality of each collected culprit. Non-minimal culprits are exempted from blame.

4.1 Separable Components

Determining component behavior, i.e., the set $\sigma(Tr, Sys, \mathcal{S})$ in Step 2 of the causality analysis framework, poses a challenge in previous approaches when a component is faulty but not suspected. Unlike suspected components, which have their outputs corrected according to their contracts, unsuspected components are not supposed to be corrected. When an input of an unsuspected component has changed as a result of correcting outputs of a suspected component, we need to determine, which outputs, possibly faulty, we should use in trace reconstruction. In [18,19], we assumed that this output was the same as observed on the given system trace. This assumption was due to the unavailable information of how a faulty should behave in such a scenario, where the component's behavior has to be assumed to obtain the output. This assumption may lead to an imprecise analysis if the actual component follows a different behavior model when it is faulty.

A more realistic treatment is to re-execute the components that are available to the analyzer so that the actual output of a component given a changed input can be produced by the component itself whether it being faulty or not. We call such components *separable* since they can be separated from the original system and re-executed in a controlled experiment setting.

Example 1. For the running example, in the trace reconstruction for the case of {PO}, when PO outputs a correct value 71 corresponding to the input 71 from Patient, Ctrl has a changed input 71 other than 75 on the observed trace. With Ctrl being a separable component, it is possible to experimentally feed the new input to Ctrl so as to observe its output. Note that Ctrl is also a faulty component (that we are not suspecting when analyzing {PO}), it may or may not produce the expected value in Figure 2. If the output ticket value from Ctrl is 0 (resp., 1), then the patient gets no infusion (resp., infusion for 1 time unit). In either case, the patient SpO2 correspondingly does not drop below the 70 threshold, therefore there is no system property violation, so {PO} is a necessary cause. On the other hand, if the output ticket value from Ctrl is 2 or above, then the PCA would continue the infusion for 2 time units, so there is still system property violation. In this case {PO} is not a necessary cause. □

Being able to re-execute separable components increases analysis precision. The output obtained by assuming the faulty, unsuspected components to always produce the same output as on the observed trace in [18] is essentially one case included in the analysis with separable components. For the example, in the analysis in [18] Ctrl would produce a ticket output 3 as observed, which is on of the cases when the Ctrl output is 2 or above in the analysis in Example 1. With separable components, a more detailed causality analysis result is obtained.

We now provide a formal definition of separable components, of the *re-execute* function, and a property on them. A separable component is simply a trace generator that takes a sequence of input and produces a corresponding output sequence. We require that the behavior of the separable component is deterministic with respect to the sequence of input and the internal states, even when faulty.

Definition 9 (Separable component)**.** *Let* $C = \langle I, O, \mathbf{P}_C \rangle$ *be a component.* C *is* separable *if* $\forall Tr, Tr' \in \mathbf{T}_C.(\pi_I(Tr) = \pi_I(Tr') \implies Tr = Tr'$.

This definition ensures that the outputs of the component are deterministic and only depend on the inputs fed to it.

Definition 10 (Re-execute function, re-execute(C, Tr))**.** *Let* $C = \langle I, O, \mathbf{P}_C \rangle$ *be a component and* Tr *be a trace. re-execute*(C, Tr) *is the trace given by* C *if* $\pi_I(Tr)$ *is fed to* C *as input.*

Property 1. *Let* $C = \langle I, O, \mathbf{P}_C \rangle$ *be a separable component and* Tr *and* Tr' *be two traces, then* $\pi_I(Tr) = \pi_I(Tr') \implies$ *re-execute*$(C, Tr) =$ *re-execute*(C, Tr').

This property is a direct consequence of the two previous definitions. It means that if we execute the separable component with the same input, it will always produce the same trace as output.

Trace Reconstruction with Separable Components. We use the construction of *cone of influence*, adapted from [8], to over-approximate the impact of the faulty components on the rest of the system. Informally, when a value has changed in a snapshot on a system trace, then the ones in the cone of influence must be updated accordingly to reflect the impact of this change. A trace Tr' is deemed as a counterfactual for a trace Tr if they share the same prefix outside of the cone of influence.

Definition 11 (Cone of influence with separable components, $\mathbf{K}(Tr, \mathcal{S}, \mathcal{R})$)**.** *Given a system* $Sys = \langle \mathcal{A}, \theta, \mathbf{P} \rangle$*, with* $\mathcal{A} = \{C_1, ..., C_J\}$ *and* $C_j = \langle I_j, O_j, \mathbf{P}_j \rangle$ *for* $1 \leq j \leq J$*, a system trace* Tr*, a set* $\mathcal{S} \subseteq \{1, ..., J\}$ *of suspected component indices, and a set* $\mathcal{R} \subseteq \{1, ..., J\}$ *of separable component indices. Let* $A = \langle I, O, \mathbf{P} \rangle$ *be the composition of the components in* $\mathcal{A}$*. The* cone of influence $\mathbf{K} = \mathbf{K}(Tr, \mathcal{S}, \mathcal{R}) = (\ell_v)_{v \in I \cup O}$ *is a vector of maximal indexes which satisfies the following properties:*
$\forall v \in \left(\bigcup_{i \in \{1,...,J\}} I_i\right) \cup O$*:* $\ell_v \leqslant |Tr|$ *and*
(1) $(v \in O \wedge C(v) \in \mathcal{S}) \implies l_v \leqslant fv_{C(v)}(Tr)$
(2) $\exists v' \in O.(v', v) \in \theta \implies l_v = l_{v'}$
(3) $\Big(v \in O \wedge \big(C(v) \in \mathcal{R} \ \vee \ fd(v, Tr) \leqslant fv_{C(v)}(Tr)\big)\Big)$
$\implies \ell_v \leqslant min(fv_{C(v)}(Tr), fd(v, Tr))$
with $C(v) = i$ *such that* $v \in I_i \cup O_i$*,* $fv_i(Tr) = min(\{\ell \in \{0, ..., |Tr| - 1 \mid \pi_i(Tr[0..\ell]) \notin \mathbf{P}_{C(v)}\} \cup \{|Tr|\})$*,* $l_{min}(i) = min\{\ell_v \mid v \in I_i\}$*, and*

$$
\begin{aligned}
fd(v, Tr) = min(\{\ell \in \{l_{min}(v) - 1, ..., |Tr|\} \mid \exists Tr' \in \mathbf{P}_{C(v)}.\forall v' \in I_{C(v)}. \\
\pi_{v'}(Tr[0..\ell_{v'} - 1]) = \pi_{v'}(Tr'[0..\ell_{v'} - 1]) \wedge \\
\pi_v(Tr[0..l_{min}(v) - 1]) = \pi_v(Tr'[0..l_{min}(v) - 1]) \wedge \\
\pi_v(Tr[\ell + 1]) \neq \pi_v(Tr'[\ell + 1])\})
\end{aligned}
$$

$C(v)$ is the component to which variable v belongs. $fv_i(Tr)$ is the index of the first violation of $\mathbf{P}_i$ in Tr. $l_{min}(i)$ is the minimal l_v for the inputs v

of component C_i. The first constraint means that an output variable from a suspected component must be in the cone if the component is faulty. The second constraint propagates the entry of outputs in the cone to the inputs on which they are linked. The third constraint means that an output v of a non-faulty, or separable, component i receiving inputs from a component in the cone must enter the cone at the latest when component i becomes faulty or at $fd(v, Tr)$. $fd(v, Tr)$ is the first index on which v can differ from its observed value when the component producing v is fed its observed input up to the cone, followed by arbitrary values.

Definition 12 (Counterfactuals)**.** *Let Sys be a system definition, Tr be a system trace, $\mathcal{R} \subseteq \{1, \ldots, J\}$ be the set of separable components indices, and $\mathbf{K} = \mathbf{K}(Tr, \mathcal{S}, \mathcal{R})$ be the cone of influence. Let $A = \langle I, O, \mathbf{P} \rangle$ be the composition of the components in $\mathcal{A}$. We define the* counterfactuals *of trace Tr given cone $\mathbf{K}$ and separable components $\mathcal{R}$ to be*

$\sigma(Tr, \mathbf{K}, \mathcal{R}) = \big\{Tr' \in \mathbf{T} \mid \forall v \in I \cup O.\pi_v(Tr')[0..\ell_v - 1] = \pi_v(Tr[0..\ell_v - 1]) \wedge$
$\forall i \in \{1, ..., J\}.\big[(i \in \mathcal{S} \vee i \notin \mathcal{R}) \wedge \pi_i(Tr') \in \mathbf{P}_i \vee (i \notin \mathcal{S} \wedge i \in \mathcal{R}) \wedge \pi_i(Tr') \in RX(i)\big]\big\}$,
where $RX(C_i) = \{Tr \in \mathbf{T}_{I_i \cup O_i} \mid \exists Tr' \in \mathbf{T}_{I_i}.Tr = \textit{re-execute}(C_i, Tr')\}$.

The notion of counterfactuals represents the reconstructed set of possible system traces when the faulty suffixes of the suspected components' observed traces are replaced with correct ones, and the effects of such faults are reconstructed by the separable components. The first condition in the definition of $\sigma(Tr, \mathbf{K}, \mathcal{R})$ states that the counterfactual must begin with the observed *unaffected* prefixes $\pi_v(Tr[0..\ell_v - 1])$ before the cone is entered. The second condition states that if a component is not separable, it will be prolonged using its property; for separable components, we re-execute them to build the new trace.

In practice, Tr' is constructed incrementally, without explicitly computing $RX(C_i)$, provided that all components have a finite specification. In that case, the cone and the counterfactuals are computed by the algorithm in Figure 4. The trace reconstruction depends on the causality definition being used, where we illustrate necessary vertical cause here. In the general case, the algorithm takes the causality definition as an input, based on which different trace reconstruction procedures are selected.

The use of separable components is a hybrid approach in trace reconstruction. For the system behavior that comes before the violations of the suspected components or that is generated by non-faulty components, we want to keep the reconstructed trace as similar as the observed trace, so static information (observed trace Tr and system definition Sys) that is already available at the time of the analysis is used. For the alternative system behavior that depends on dynamic component output, separable components are used to generate run-time output to be used in trace reconstruction.

Input:
- observed trace Tr,
- set $\mathcal{F}$ of faulty components,
- set $\mathcal{S} \subseteq \mathcal{F}$ of suspected components,
- set $\mathcal{R}$ of separable components, and
- system definition $Sys = \langle \mathcal{A}, \theta, \mathbf{P} \rangle$.

Output:
- a prefix Tr' of a system trace on which either $Tr' \models \mathbf{P}$ or $Tr' \not\models \mathbf{P}$.

Algorithm for Trace Reconstruction:
(1) Compute the cone of influence $\mathbf{K} = \mathbf{K}(Tr, \mathcal{S}, \mathcal{R})$.
(2) Feed maximal unaffected prefixes $pr_v = \pi_v(Tr[0..\ell_v - 1])$ (Definition 12) of input variables to components. Keep pr_i on output trace Tr'.
(3) Starting from Snapshot 0 on Tr', for each missing value in Snapshot k, (a) identify component C_i responsible for producing the value, (b) read input values for C_i in Snapshot k, (c) obtain component C_i's output, depending on conditions in Definition 12, and (d) add the output to Snapshot k of Tr'.
(4) Repeat the above step until either $Tr' \not\models \mathbf{P}$, or an application-specific length of trace is reached. Return Tr' as the reconstructed trace.

Fig. 4. Trace Reconstruction Algorithm

4.2 Culprit Minimization with Horizontal Causality

A second extension to our existing approach is to replace the use of set containment checking in [18] with the use of horizontal causality, in order to exclude non-minimal subsets of causes from blame. The approach in [18] starts from a viewpoint that always aims to blame the minimal number of components and removes a culprit from blame if one of its proper subsets is also a culprit. While this treatment may have provided one approach to reduce the number of components in a culprit, it is counter-intuitive. Dependency in between component interactions is completely overlooked with this treatment.

With the use of horizontal causality, relationships in between components can be utilized to improve the precision of causality analysis. In details, if the analysis determines that a non-singleton subset $\mathcal{S}$ of faulty components is a culprit, then the horizontal causalities between the component property violations are investigated. Let $I = \{i_{Tr_C, \mathbf{P}_C} \mid C \in \mathcal{S}\}$ be the set of indices for component violations in $\mathcal{S}$ on trace Tr. Then for each $i, j \in I$ such that $i < j$, the horizontal causality between the subset $\mathcal{S}_i = \{C_l \in \mathcal{S} \mid l = i\}$ and each component C_r in $\mathcal{S}_j = \{C_l \in \mathcal{C} \mid l = j\}$ is investigated. If the property violations in $\mathcal{S}_i$ is a horizontal cause for the property violation in C_r, then C_r is removed from $\mathcal{S}$.

Example 2. For the running example, when the analysis determined that the set $\{\text{PO}, \text{Ctrl}\}$ is a culprit, the analysis in [18] used simple set minimization to exclude $\{\text{PO}, \text{Ctrl}\}$ from blame, since the singleton set $\{\text{Ctrl}\}$ is a cause as well. With horizontal causality: In the case of $\{\text{PO}, \text{Ctrl}\}$ being a cause and the fault in PO occurs before the fault in Ctrl, we remove the blame on Ctrl only if the fault in PO causes the fault in Ctrl. In this example, should PO output a correct value 71, the expected output from Ctrl should be 0 time unit. In the approach in [18],

Analysis in [18, 19]	Analysis in [18, 19] with Horizontal Causality
— {Ctrl} and {PO, Ctrl} are necessary causes — Blame only {Ctrl} — No blame on {PO, Ctrl} due to set minimization	— {Ctrl} and {PO, Ctrl} are necessary causes — No horizontal causality of property violations between {PO} and {Ctrl} — Blame {Ctrl} and {PO, Ctrl}
Analysis in [18, 19] with Separable Components	**Analysis in [18, 19] with both Horizontal Causality and Separable Components**
— {Ctrl} and {PO, Ctrl} are necessary causes — {PO} is a necessary cause when Ctrl outputs 0 or 1 (given input 71) during trace reconstruction — Blame {Ctrl} and {PO} when Ctrl output is 0 or 1; otherwise blame {Ctrl} only — In neither case is {PO, Ctrl} blamed due to set minimization	— {Ctrl} and {PO, Ctrl} are necessary causes — {PO} is a necessary cause when Ctrl outputs 0 or 1 (given input 71) during trace reconstruction — {Ctrl} is blamed — {PO, Ctrl} is blamed only when Ctrl output ≥ 2 — {PO} is blamed if Ctrl output is 0 or 1. When Ctrl output is 1, {PO} is the vertical cause and blamed. When Ctrl output is 0, there may be two explanations: (a) {PO} is the vertical cause, or (b) {PO, Ctrl} is a vertical cause, and {PO} is the horizontal cause for Ctrl, so {PO} is blamed, not {PO, Ctrl}

Table 2. Comparison of Causality Analysis Results with Extensions

Ctrl outputs 3 as on the observed trace, i.e., the property violation in Ctrl does not disappear, so the fault in PO is not a horizontal cause for the fault in Ctrl. Therefore, it is not proper to simply remove $\{\text{PO}, \text{Ctrl}\}$ from blame. Both $\{\text{Ctrl}\}$ and $\{\text{PO}, \text{Ctrl}\}$ should be taken as blame, as they represent different scenarios that the system property violation can be prevented. □

We note that the two extensions we introduced in this work are orthogonal, and can be applied to our existing approach individually. However, the analysis results, as summarized in Table 2, do not achieve the same precision as when both extensions are incorporated, as shown in Example 3 below.

Example 3. Continuing the running example where $\{\text{PO}, \text{Ctrl}\}$ is determined to be a necessary vertical cause, if separable component Ctrl is used for causality analysis, then the Ctrl output, given the changed input 71, is not necessarily 3 as on the observed trace. If the Ctrl output is 1 or greater, then the property violation in $\{\text{PO}\}$ is not a horizontal cause for the property violation in $\{\text{Ctrl}\}$, thus the blame on $\{\text{PO}, \text{Ctrl}\}$ is not excluded. On the other hand, if Ctrl's output is 0, the horizontal causality between property violations in between $\{\text{PO}\}$ and $\{\text{Ctrl}\}$ is established. The blame on $\{\text{PO}, \text{Ctrl}\}$ is reduced to $\{\text{PO}\}$, even if $\{\text{PO}\}$ itself is not determined as a necessary vertical cause in the first place. □

5 Implementing Causality Analyzer

In this section, we present some key implementation details for the proposed causality analysis. We implemented trace recording, reconstruction, and causality analysis modules based on the publicly available medical device coordination framework (MDCF) [12]. MDCF is a message exchange platform for medical devices operating in a cooperative fashion. The framework implements message publish/subscribe model as specified in the ICE standard [2].

Trace Recording. The PCA example for our case study is component-based. Each component registers itself as a message publisher to send messages to others and as a subscriber to receive messages from others. We also implemented a network-wide data logger for MDCF. The data logger declares itself as a subscriber to all relevant messages exchanged via message bus. When a message originates from a medical device, it is time-stamped, serialized, and sent to MDCF. MDCF will push the message to the data logger (as well as other subscribers to the message) so that the data logger is able to capture the message together with its time-stamp of creation. The recorded traces are then normalized to the formalism presented in Section 3.1 based on the case study's setting that the components are are stepped by a timer of 500ms.

Implementing Separable Components. In our case study, a component can be viewed as a trace generator in that it takes a sequence of input and produces a sequence of output. A component in MDCF is then naturally separable in that the component can be incorporated in the trace reconstruction module in a controlled fashion.

In details, for components with no internal states, the component implementation is simply executed each time an output is needed. For a separable component C with internal states, the trace reconstruction starts with the initial internal states of C and replays the recorded snapshots on the observed trace Tr back to C, up to the boundary of cone of influence. Afterwards, input to C may have changed as other faulty suspected components may generate new output that is fed to C. The separable component C then reads the input and produces an output. In this way, the internal states of C are implicitly kept within C's implementation, without being explicitly monitored.

Causality Analyzer. The implementation of the causality analyzer follows the functional blocks shown in Figure 3. Notice that for the analysis of horizontal causality in Step 4 of the causality analysis framework, no additional trace reconstruction is required if we cache the reconstructed traces. For instance, when having determined $\{\text{PO}, \text{Ctrl}\}$ as a culprit, we then need to investigate whether the property violation in PO is a cause of the property violation in Ctrl. This requires the reconstruction of the trace when PO alone is suspected. This has already been done, when investigating $\{\text{PO}\}$ for vertical causality.

Result. We have instructed the causality analyzer to output useful human-readable information with regard to the analysis process. A sample run of the analysis outputs (a) set of faulty components, (b) for each suspect, whether it is a culprit, (c) if a suspect is a non-singleton culprit, whether there are horizontal causal relations between its components, and (d) a list of the minimal culprits. The expected analysis result as discussed in Section 4 for the running example is summarized in the bottom right cell of Table 2. Note that the result does not invalidate our previous analysis presented in [18,19] as the criteria of determining minimal culprits are different. Also, with separable components, our proposed analysis is equipped with a more realistic trace reconstruction technique that produces more accurate counterfactual traces, thus we obtain culprits that match our intuition better for this case study.

Scalability. By our problem definition it is inevitable to investigate each non-empty subset of faulty components and determine if it is a culprit. Thus the overall complexity of our approach is exponential in the number of faulty components. However, we note that in practice, the number of faulty components is usually small and tractable. Also, as has been shown in [18,19] that state-of-the-art SAT/SMT solvers (e.g., Z3 [6]) can be used to efficiently solve a causality analysis problem instance, our approach could benefit from leveraging SAT/SMT solvers with proper encoding.

6 Related Work

Analysis for causes has long been a human intellectual inquisition. Recent philosophical inquisition on causality based on counterfactuals starts from Hume, and is extensively studied in [15]. Halpern & Pearl [10] were among the first to provide a formalism to reason about causality. [10] defines causality for structural equation models (constraints on variables) and does not study component-based or real-time systems, which typically exhibit much more complex behaviors.

Works following Halpern & Pearl's definitions include [3,13,14], all requiring a cause to be both necessary and sufficient. The work in [3] illustrated how causality can be used for providing an explanation of system property violation. The work in [13,14] is based on the assumption that a plethora of system traces can be obtained for analysis so that is possible to categorize the available traces by trace characteristics so that each category can be regarded as a failure mode in the failure mode and effects analysis (FMEA). The work in [3,13,14] all model traces as a sequence of observed events, and the occurrence or absence of events are potential causes to the violation of properties, which are modeled as temporal logic properties. These approaches neglect the underlying system components that generate the events as well as the interactions between components. Similarly, work based on using distance metrics [4,17] to measure the similarity between actual and counterfactual traces shares the same limitation.

The work in [11] and our previous work in [18] share similar ideas if program statements are viewed as black-box components. Encoding the program and error trace into a MAX-SAT problem instance can yield a set of program statements so that correcting the identified statements can eliminate the program error. On a larger scale, the delta-debugging technique proposed in [20] can also be viewed as an application of counterfactual reasoning: debugging is by experimentally correcting statements of a program until a set of statements are found to eliminate compiler panic should the identified statements are corrected.

Our line of work [9,18,8,19,7] starts with preliminary definitions of causalities for component-based systems [9,18] and extends to real-time settings for system definitions with logical constraints [19], synchronous systems [8], and timed automata [7]. A salient difference of our work from existing ones is that, although we assume components are black-boxes, we take expected component behaviors specified in component properties as guidelines for trace reconstruction. This directs us to a set of counterfactual traces that are more relevant to the observed

one. We in this work further employs separable components, which first appeared in combinational circuits diagnosis [5] where internal states of components are not considered.

7 Conclusion

We presented an extension of trace reconstruction algorithm for causality analysis. Using a case study from the medical domain, we show that the extension improves precision of the analysis and matches our intuition about the analysis results. The key to the improvement is the ability to re-execute some of the system components separately from the rest of the system. We further show that analysis can be improved by considering horizontal causality; that is, taking input-output dependencies between components into consideration in order to avoid induced faults. Our future work will concentrate on extending causality analysis to cover weaker component contracts that may make some of the faults unobservable.

References

1. D. Arney, M. Pajic, J. M. Goldman, I. Lee, R. Mangharam, and O. Sokolsky. Toward patient safety in closed-loop medical device systems. In *ICCPS'10*, pages 139–148, New York, NY, USA, 2010. ACM.
2. ASTM International. *F2761-2009. Medical Devices and Medical Systems — Essential Safety Requirements for Equipment Comprising the Patient-Centric Integrated Clinical Environment (ICE), Part 1*, 2009.
3. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *CAV'09*, pages 94–108. Springer, 2009.
4. S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. *SIGSOFT Softw. Eng. Notes*, 29(6):73–82, Oct. 2004.
5. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97 – 130, 1987.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*. Springer.
7. G. Gössler and L. Aştefănoaei. Blaming in component-based real-time systems. In *Proceedings of the 14th International Conference on Embedded Software*, 2014.
8. G. Gössler and D. Le Métayer. A general trace-based framework of logical causality. In *Proceedings of Formal Aspects of Component Software (FACS)*, 2013.
9. G. Gössler, D. Le Métayer, and J.-B. Raclet. Causality analysis in contract violation. In *Runtime Verification*, 2010.
10. J. Y. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *British Jnl. for the Philosophy of Sci.*, 56(4), 2005.
11. M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. *SIGPLAN Not.*, 46(6):437–446, June 2011.
12. A. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. P. Jetley, P. L. Jones, and S. Weininger. An open test bed for medical device integration and coordination. In *ICSE Companion*, pages 141–151. IEEE, 2009.
13. M. Kuntz, F. Leitner-Fischer, and S. Leue. From probabilistic counterexamples via causality to fault trees. In *Computer Safety, Reliability, and Security*. 2011.

14. F. Leitner-Fischer and S. Leue. On the synergy of probabilistic causality computation and causality checking. In *SPIN*, 2013.
15. D. Lewis. *Counterfactuals*. Wiley-Blackwell, 2nd edition, 2001.
16. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
17. M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE'03*, pages 30–39, 2003.
18. S. Wang, A. Ayoub, R. Ivanov, O. Sokolsky, and I. Lee. Contract-based blame assignment by trace analysis. In *HiCoNS'13*, pages 117–125, 2013.
19. S. Wang, A. Ayoub, B. Kim, G. Gössler, O. Sokolsky, and I. Lee. A causality analysis framework for component-based real-time systems. In *RV'13*, 2013.
20. A. Zeller. Isolating cause-effect chains from computer programs. In *ACM International Symposium on Foundations of Software Engineering*, pages 1–10, 2002.