9-2013

A Causality Analysis Framework for Component-Based Real-Time Systems

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

Anaheed Ayoub
University of Pennsylvania, anaheed@seas.upenn.edu

BaekGyu Kim
University of Pennsylvania, baekgyu@seas.upenn.edu

Gregor Gössler

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

See next page for additional authors

Recommended Citation (OVERRIDE)

Wang, S., Ayoub, A., Kim, B., Gössler, G., Sokolsky, O., & Lee, I. (2013). A causality analysis framework for component-based real-time systems. In A. Legay & S. Bensalem (Eds.), *Lecture notes in computer science: Runtime verification* (Vol. 8174, pp. 285-303). Springer-Verlag Berlin Heidelberg. DOI: 10.1007/978-3-642-40787-1_17

4th International Conference on Runtime Verification (RV '13), INRIA Rennes, France, September 24-27, 2013.

A Causality Analysis Framework for Component-Based Real-Time Systems

Abstract
We propose an approach to enhance the fault diagnosis in black-box component-based systems, in which only events on component interfaces are observable, and assume that causal dependencies between component interface events within components are not known. For such systems, we describe a causality analysis framework that helps us establish the causal relationship between component failures and system failures, given an observed system execution trace. The analysis is based on a formalization of counterfactual reasoning, and applicable to real-time systems. We illustrate the analysis with a case study from the medical device domain.

Comments
4th International Conference on Runtime Verification (RV '13), INRIA Rennes, France, September 24-27, 2013.

Author(s)
Shaohui Wang, Anaheed Ayoub, BaekGyu Kim, Gregor Gössler, Oleg Sokolsky, and Insup Lee

A Causality Analysis Framework for Component-based Real-time Systems*

Shaohui Wang[1], Anaheed Ayoub[1], BaekGyu Kim[1],
Gregor Gössler[2], Oleg Sokolsky[1], and Insup Lee[1]

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

Abstract. We propose an approach to enhance the fault diagnosis in black-box component-based systems, in which only events on component interfaces are observable, and assume that causal dependencies between component interface events within components are not known. For such systems, we describe a causality analysis framework that helps us establish the causal relationship between component failures and system failures, given an observed system execution trace. The analysis is based on a formalization of counterfactual reasoning, and applicable to real-time systems. We illustrate the analysis with a case study from the medical device domain.

1 Introduction

Component-based design in systems engineering enables independent development of system components as well as their incremental construction and modification. The complexity of systems that are built with component-based design renders it difficult to determine the culprit components of the system that are responsible for the discovered system failure on a given system execution. We in this paper aim to present a formal framework for the analysis of the causal relation between the faulty components and an observed system failure on a given system execution.

While this problem is common to all safety-critical domains, our immediate motivation comes from the domain of medical devices. In the United States, the Food and Drug Administration (FDA) is responsible for assessing safety of medical devices and regulating their use in health care. When a system failure that harms a patient, known as an *adverse event* occurs, the hospital is required to report it to the FDA-maintained database [9]. Diagnosis of the root cause is crucial for the subsequent recovery and follow-up prevention measures. Such diagnosis requires recording of system executions leading to the failure, as well as methods for the efficient analysis of the recorded system trace.

* Research is supported in part by the National Science Foundation grants CNS-0930647 and CNS-1035715, and NSF/FDA SiR grant CNS-1042829.

Existing work in fault diagnosis (e.g., [6,21,8,5,23,17] to name only a few) aims to study (1) the discovery of existence of faults in the system, and (2) the identification of the types and locations of the faults. A main assumption implicitly used in the work of fault diagnosis is that, the computed fault propagation chain is the actual cause-effect chain [17].

We in this work consider systems whose components are black-boxes, where only events on component interfaces are observable, and assume that causal dependencies between component interface events within components are not known. The presence of uncertainty in computing fault propagation chain inside components leads to an over-approximation of the fault propagation chain. We have shown in our preliminary study [26] that, the precision of this over-approximation can be improved by *causality analysis*, i.e., reasoning about whether a fault inside a component is the cause for system failure.

Causality is commonly defined by the use of counterfactual reasoning [13,16,19]. Some recent work in the engineering domain has discussed several versions of causality definitions for finite state automata [11] and temporal logics [4,14,15]. In this work, we extend our previous result in [26] to consider the case of real-time systems where a system execution trace is a sequence of timestamped events, and the system/component specifications are based on the timing of events.

Contributions. We present a framework for the causality analysis for component-based systems. We identify the steps of the analysis and the input and output for each step. We show with a case study from the medical device domain how to use the proposed framework to establish the causal relationship between component failures and the system failure. In particular, we extend our approach presented in [26] to handle the causality analysis for real-time systems.

Paper Organization. We first use a simple example as an illustration to define the causality analysis problem in Section 2. We then present a proposed causality analysis framework for component-based systems in Section 3. In Section 4, we present the main technique used for causality analysis. We show how to apply the causality analysis to our case study in Section 5. We discuss some of the assumptions of our approach in Section 6 and related work in Section 7, and conclude in Section 8.

2 Motivating Example and Problem Statement

2.1 The Generic Patient-Controlled Analgesia Pump Case Study

The Generic Patient Controlled Analgesic (GPCA) infusion pump project [10] aims at developing a reference software model for PCA infusion pump systems with which formal techniques can be performed to ensure the GPCA safety requirements [12]. We focus on the core safety requirements in this case study to demonstrate our causality analysis framework:

> *A bolus dose shall be given when requested by the patient, and when the drug reservoir is empty and an infusion session is in progress, an alarm shall be issued and the pump motor should be stopped.*

An *infusion session* is defined as the interval from the start of the pump motor till its stop.

We implemented software that captures the requirements on an Atmel SAM7X-EK development board [2], running the FreeRTOS real-time operating system [3]. The development board is interfaced to the sensors and actuators of the Baxter PCA infusion pump hardware. Sensor signals from the bolus request button and the empty reservoir detection switch are captured through periodic sampling. Instructions for the pump motor to start and stop are delivered via pulse width modulation. Alarms are signaled with flashing LEDs in our experiments.

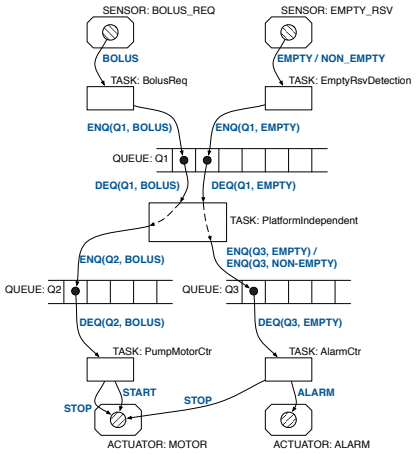

Fig. 1. Data flow of the GPCA

The FreeRTOS in our case study runs a priority-based, preemptive scheduler. Five tasks, each implemented with an independent C function, communicate with each other by sending and receiving messages in three message queues, Q1, Q2, and Q3. Some tasks have individual local variables, which we deem as unknown to the analysis due to our black-box assumption, but they do not share global variables in our implementation. In this case study, we view each task as a component; the terms *task* and *component* are used interchangeably.

The five tasks are summarized in Table 1. The Priority column indicates the task priorities in addition to the system idle task's priority of FreeRTOS. The left (right, resp.) arrow means that the corresponding task reads (sends, resp.) messages from (to, resp.) the queue. An *event* in the system is a single action performed by a task. We attach each event with a timestamp, denoted as a pair (e, t), where e is the event and $t \in \mathbb{R}_{\geq 0}$ is the timestamp. For example, $(\text{ALARM}, 9760)$ represents an event where the AC task has raised an alarm, at time 9760 ms since the system starts execution. The events we recorded for this case study is summarized in Table 2. We assume it is known which task produces which event. In particular, we assume we know whether an instance of the STOP event is produced by AC or PM.

Task Name	Abbreviation	Period	Priority	Queues Accessed
PumpMotorCtr	PM	300 ms	+6	$\leftarrow Q2$
PlatformIndependent	PI	Aperiodic	+5	$\leftarrow Q1, \rightarrow Q2, Q3$
BolusReq	BR	500 ms	+4	$\rightarrow Q1$
AlarmCtr	AC	500 ms	+3	$\leftarrow Q3$
EmptyRsvDetection	ER	1000 ms	+2	$\leftarrow Q3$

Table 1. Tasks in FreeRTOS Implementation for GPCA

Event	Task	Description
START	PM	The pump motor has been started.
STOP	PM or AC	The pump motor has been stopped.
ALARM	AC	The alarm has been fired.
ENQ(Q, M)	BR, ER, or PI	MessageM has been put to queue Q.
DEQ(Q, M)	PI, PM, or AC	MessageM has been retrieved from queue Q.

Table 2. Events Recorded in GPCA Controller

The data flow of the events in the system is depicted in Figure 1. The BolusReq (BR) and EmptyRsvDetection (ER) tasks periodically check if there are patient bolus request or empty/non-empty reservoir signals from sensors, respectively; if there are, they put the messages to Q1. We do not consider faults in these two tasks in our analysis. The aperiodic PlatformIndependent (PI) task is triggered whenever there is a message sent to Q1. It moves the bolus request and empty/non-empty reservoir messages to Q2 and Q3, respectively. The PumpMotorCtr (PM) task periodically checks if there are bolus request messages in Q2; if there are, it will start the infusion session by keeping the pump motor running for 30 periods (i.e., 9 seconds for each patient bolus request). The AlarmCtr (AC) task periodically checks if there are empty reservoir messages in Q3; if there are, it will raise an alarm and stop the pump motor. Each task has a response time of 10 ms after a message is received. We assume here that the queues in the system are reliable, i.e., no messages are lost/duplicated/altered in a queue.

The task behaviors described above reflect our black-box assumption: the two data flow paths shown in Figure 1 both pass through queue Q1 and the task PI, yet we do not know whether there is fault propagation from the EMPTY_RSV sensor to the PM task, due to the assumption that PI is a black-box to the analysis. (The dashed links inside PI in Figure 1 indicate unknown data flows.) Essentially, this is what we intend to infer from the causality analysis.

With the recorded events, we express the GPCA safety requirement as the following Metric Interval Temporal Logic (MITL) [1] property:

$$\begin{aligned}\varphi_S := \Box_{(0,\infty)}[&\text{ENQ}(\text{Q1, BOLUS}) \rightarrow \Diamond_{(0,650)}[\text{START} \wedge \\ &[\Box_{(0,9000)}\neg\text{ENQ}(\text{Q1, EMPTY}) \wedge \Diamond_{(8990,9010)}\text{STOP}] \vee \\ &[\Box_{(0,9000)}[\text{ENQ}(\text{Q1,EMPTY}) \rightarrow [\Diamond_{(0,1050)}\text{ALARM} \wedge \Diamond_{(0,1050)}\text{STOP}]]]]].\end{aligned} \tag{1}$$

The values in the formula are obtained from the system implementation. For example, the value 650, indicating the maximal allowed delay (in milliseconds) from the instance when a BOLUS is put to Q1 to the instance when the START message is delivered by PM, is due to that (a) the aperiodic task PI has a worst case delay of 10 ms to retrieve the message BOLUS from Q1, plus a possible 10 ms delay due to preemption by PM; (b) similarly a 20 ms worst case delay for PI to move the BOLUS message to Q2; (c) since PM has a period of 300 ms, in the worst case, it takes up to two periods of PM to read the message once it is enqueued in Q2 (see Figure 3 in Subsection 5.2 for details), and (d) the worst case delay of the PM task is 10 ms. The rest of the time periods are analogously specified. It is required that the behaviors of the tasks constitute a subset of the

behaviors specified by the system constraint φ_S, which is formally stated later in Hypothesis 1 in Subsection 2.2.

A system execution is captured by collecting the events with their timestamps by instrumenting the GPCA implementation. We assume in this paper that recording is perfect, i.e., no events in the system are missing on a trace, and each event on a trace actually happened at its recorded timestamp.

A trace is a set of timestamped events. For example, $\{(\text{ENQ}(\text{Q1, BOLUS}), 8500), (\text{DEQ}(\text{Q1, BOLUS}), 8502), (\text{ENQ}(\text{Q3, EMPTY}), 8503), (\text{DEQ}(\text{Q3, EMPTY}), 8701), (\text{ALARM}, 9760), (\text{STOP}, 9760)\}$ is a trace with six events observed. The events are naturally ordered by their corresponding timestamps. On this trace, PI mistakenly put the bolus request message to the wrong queue with the wrong message. AC reads the empty reservoir message but fails to ALARM and STOP within its deadline. The system property φ_S is violated since there is no bolus dose delivered to the patient after the bolus request event ENQ(Q1, BOLUS) (i.e., Equation (1)). So two faulty tasks, PM and AC, may have caused the system property violation.

In the causality analysis problem, we would like to investigate which subset of the faulty tasks, $\{PI\}$, $\{AC\}$, or $\{PI, AC\}$, caused the system property violation. We leave the details of the analysis to Section 5 but only show the result here: both $\{PI\}$ and $\{PI, AC\}$ satisfy the counterfactual test for causality, so we report the minimal subset $\{PI\}$ as the cause for the system property violation.

2.2 The Causality Analysis Problem Definition

In this subsection, we abstract the problem illustrated by the example in Subsection 2.1 and provide the formal definition of the *causality analysis problem*.

Definition 1 (Trace)**.** *A* trace *of length n is a set of n timestamped events, denoted $\{(e_1, t_1), \ldots, (e_n, t_n)\}$, such that $t_1 \leq \cdots \leq t_n$.*

A trace only contains a finite number of events. For time beyond t_n, no events happen in the system. We use logical formula to express component/system behaviors. It is assumed that given a trace Tr, the semantics of the chosen logic is two-valued: for any formula ϕ, either $Tr \models \phi$ or $Tr \not\models \phi$. In this paper, MITL and first order logic (FOL) are used for component/system specifications.

Definition 2 (Constraint)**.** *Given a set E of events, a* constraint *is a logical formula defined on E. In details, for MITL, E is the set of atomic propositions; for an event $e \in E$, $(Tr, t) \models e$ if and only if $(e, t) \in Tr$. For FOL, E is the set of logical constants.*

Definition 3 (Component)**.** *A* component *$C = \langle I_C, O_C, \varphi_C \rangle$ is a tuple where the I_C and O_C are its set of input and output, respectively, such that $I_C \cap O_C = \emptyset$, and φ_C is a constraint defined on $I_C \cup O_C$.*

The notion of the component input/output is general. In the GPCA case study, the input and output for each component are the events it could receive and send through the queues, respectively; in [26], the input and output are values passing through component data ports.

Definition 4 (System Definition). *A system definition $S = \langle C_1, \ldots, C_J \rangle$ consists of a set of components.*

The set of all events in the system is defined by $E_S = \bigcup_{j=1}^{J} I_{C_j} \cup O_{C_j}$, where J is the number of components in the system.

Definition 5 (System Property). *A system property φ_S for system definition S is a constraint defined on the set E_S of system events.*

Hypothesis 1. *There must be at least one component violation for a system property violation, or equivalently, $\bigwedge_{j=1}^{J} \varphi_{C_j} \rightarrow \varphi_S$.*

Hypothesis 1 is the basis for the causality analysis. A violation to Hypothesis 1 implies a flawed system design, which is out of the scope of this paper.

Definition 6 (Violation). *We say that a property φ is violated on trace Tr if and only if $Tr \not\models \varphi$. A system property violation is called a system failure. A component property violation is called a component failure; in such cases, the component is called faulty.*

Definition 7 (Faulty Components). *Given an observed trace Tr and a system definition S on which a system property φ_S is violated, we define*

$$\mathcal{F} = \{C \mid C \text{ is a component in } S \text{ and } Tr \not\models \varphi_C\} \tag{2}$$

to be the set of faulty components for the violation of φ_S on Tr.

Consider a *suspected subset* $\mathcal{C} \subseteq \mathcal{F}$ of faulty components. Replacing every component in $\mathcal{C}$ with a correct one would result in an alternative system S'. Let

$$\begin{aligned} TR_{\mathcal{C}} = \{&tr \mid tr \text{ is a trace for } S', \text{ and} \\ &tr \text{ has the same system input as observed on } Tr\} \end{aligned} \tag{3}$$

be the set of possible system traces for S' when rerunning the system S' with the same system input as observed on Tr. The formal characterization of $TR_{\mathcal{C}}$ is a case-by-case analysis, for which we show with the GPCA case study in Section 5. Based on $TR_{\mathcal{C}}$, several notions of causes can be defined.

Definition 8 (Contributory Cause [22]). *A (non-empty) suspected subset $\mathcal{C} \subseteq \mathcal{F}$ of faulty components is a* contributory cause *for the violation of a system property φ_S on an observed trace Tr if and only if $\exists tr \in TR_{\mathcal{C}}.tr \models \varphi_S$.*

Definition 9 (Main Contributory Cause/Necessary Cause [26,11]). *A (non-empty) suspected subset $\mathcal{C} \subseteq \mathcal{F}$ of faulty components is a* main contributory cause *for the violation of a system property φ_S on an observed trace Tr if and only if $\forall tr \in TR_{\mathcal{C}}.tr \models \varphi_S$.*

Definitions 8 and 9 bound the two extremes of defining necessary cause. Definition 8 requires there exists at least one alternative system execution trace on which the system failure disappears while Definition 9 requires so on all alternative system execution traces. In this work, we do not fix a causality definition, but take it as a parameter of the causality analysis problem.

Definition 10 (Causality Analysis Problem Definition). *Given a system definition S, a system property φ_S, and a trace Tr such that $Tr \not\models \varphi_S$, let $\mathcal{F}$ be*

as defined in Equation (2). *The* causality analysis problem *with respect to a causality definition CD, is to identify the set*

$$Culprit = \{\mathcal{C} \in 2^{\mathcal{F}} \mid \mathcal{C} \textit{ is a cause according to causality definition } CD, \textit{ and no proper subset of } \mathcal{C} \textit{ satisfies } CD\}. \quad (4)$$

We call the tuple $\langle S, \varphi_S, Tr, CD \rangle$ an instance of the causality analysis problem. It can be seen from the causality definitions that, the reconstruction of the set $TR_{\mathcal{C}}$ of alternative system execution traces is at the heart of the causality analysis. In [26] we have proposed an approach based on the transformation of a causality analysis problem instance into an unsatisfiability checking problem instance. In this paper we extend the technique to handle real-time systems where a system execution trace is a set of events ordered by their timestamps, and the system/component specifications are based on both the occurrences and timestamps of events. In the following, we first show an overview of the causality analysis framework in Section 3, and detail the techniques for causality analysis in Section 4 with a case study in Section 5.

3 The Causality Analysis Framework

In a bird's-eye view, the causality analysis process is conceptually divided into four steps, as shown in Figure 2. The shaded ovals System Definition S, System Property φ_S, observed Trace Tr with system failure, and Causality Definition CD are the input to the analysis; the output is a set $Culprit$ of Minimal Culprits for the violation of φ_S on trace Tr with respect to the causality definition CD. The intermediate artifacts, shown as unshaded ovals, and the four steps of the analysis, shown as solid boxes, are discussed below.

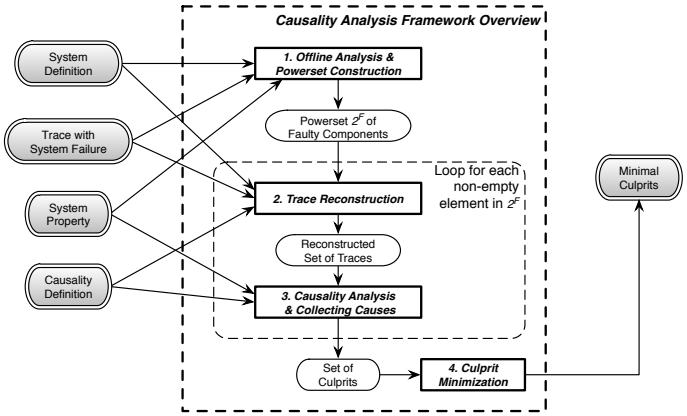

Fig. 2. The Causality Analysis Framework Overview

Step 1. *Offline Analysis & Powerset Construction.* In this step, a sanity check of whether a system property violation occurs is first performed. If not,

then there is no need for the causality analysis. Otherwise, we check Hypothesis 1 defined in Subsection 2.2. When Hypothesis 1 holds, we gather the set $\mathcal{F}$ of faulty components for the violation of φ_S on trace Tr, and construct the powerset $2^{\mathcal{F}}$ of $\mathcal{F}$.

Step 2. *Trace Reconstruction.* The trace reconstruction for the causality analysis is based on the system specification, and parametric to the suspected subset $\mathcal{C} \subseteq \mathcal{F}$ of faulty components and the causality definition. This step is at the core of the causality analysis, and we will discuss it in Section 4.

Step 3. *Causality Analysis & Collecting Causes.* For each suspected subset $\mathcal{C} \subseteq \mathcal{F}$ of faulty components, the causality analysis checks whether $\mathcal{C}$ is a culprit according to the chosen causality definition CD. If yes, it is collected for the subsequent culprit minimization; otherwise $\mathcal{C}$ is not a cause for the violation of system property φ_S according to CD.

Step 4. *Culprit Minimization.* The last step of causality analysis is to check the minimality of each collected culprit, according to Definition 4. Non-minimal culprits are pruned for precise results of causality analysis.

4 Trace Reconstruction and Causality Analysis

The trace reconstruction step in the causality analysis is to identify the set $TR_{\mathcal{C}}$ of traces when the suspected subset $\mathcal{C}$ of faulty tasks in system S are replaced with correct ones and the system is rerun with the same input as observed on trace Tr. The main idea in obtaining $TR_{\mathcal{C}}$ is to specify the logical constraint ψ that exactly the traces in $TR_{\mathcal{C}}$ satisfy. The constraint ψ is composed based on (1) task constraints for correct tasks, (2) tasks constraints for faulty tasks, (3) constraints on values observed on trace Tr, and (4) *trace reconstruction rules*. With the constructed logical constraint ψ, the problem of checking of causality based on Definition 8 (Definition 9, resp.) can be transformed into the problem of satisfiability (unsatisfiability, resp.) checking, for which state-of-the-art solvers exist [26].

In this section, we show the extension of the work presented in [26] to the case where real-time systems are considered, i.e., traces are sequences of timestamped events as in Definition 1, and system/task specifications are given as logical constraints in either temporal logics or first order logic on events.

Given a causality analysis problem instance $\langle S, \varphi_S, Tr, CD \rangle$, Step 1 of the causality analysis framework is to identify the set $\mathcal{F}$ of faulty tasks according to Definition 2. In Step 2, for each non-empty suspected subset $\mathcal{C} \subseteq \mathcal{F}$, a set $TR_{\mathcal{C}}$ of system traces is reconstructed, given that the faulty tasks in $\mathcal{C}$ are replaced with good ones and the system S is rerun with the same input events. Each task's behavior in the system is determined by the trace reconstruction rules, which indicate what constraint must be put on each task C_j in S, according to whether the task is (1) non-faulty, (2) faulty but not suspected, and (3) faulty and suspected. Informally, the three rules are summarized as follows.

(R1) If $C_j \notin \mathcal{F}$, then it is deemed as a good task. In the trace reconstruction, C_j's behavior is constrained by φ_{C_j}, i.e., a correct task's constraint.
(R2) If $C_j \in \mathcal{F} \setminus \mathcal{C}$, i.e., C_j is faulty but not in the consideration of being suspected, then all output events produced by C_j on trace Tr are preserved on any reconstructed traces.
(R3) If $C_j \in \mathcal{C} \subseteq \mathcal{F}$, then C_j is a faulty task that is replaced by a good one. In this case, the trace reconstruction "removes" the events that should not have occurred on the trace Tr, and "adds" those which must be produced by C_j.

The logical constraint to express that an event e is observed at time t on the trace Tr is expressed as

$$on_{Tr}(e,t) := \exists (e',t') \in Tr.e' = e \wedge t' = t. \tag{5}$$

The constraint that all events task C_j produced on Tr are preserved on reconstructed traces is specified with

$$\kappa_{C_j} := \forall e \in O_{C_j}.\forall t \in \mathbb{R}_{\geq 0}.[on_{Tr}(e,t) \rightarrow \exists (e',t').e' = e \wedge t' = t]. \tag{6}$$

The constraint κ_{C_j} means that, any execution trace that satisfies κ_{C_j} must have an event e' which is the same as the e delivered at time $t' = t$, for any timestamped event (e,t) on Tr.

The task constraint of "removing" events from a trace in the trace reconstruction is done by adding more constraints to rule out traces where the events that have to be removed occur. An event e must be removed in the trace reconstruction if (1) e is produced by a suspected faulty task C_j, and (2) there is no other event on the trace that triggers the event e. The task constraint of "adding" events that a faulty task C_j must have produced is specified by augmenting the task constraint φ_{C_j} to specify the allowed time ranges for output events from C_j. The definitions of the "removing" and "adding" constraints are application dependent. We defer the details to Subsection 5.3, and use ρ_{C_j} and α_{C_j} for now to represent the two constraints for "removing" and "adding", respectively.

The conditions for the rules (R1)–(R3) are defined as

$$\xi_{C_j,1} := \neg in(C_j, \mathcal{F}). \tag{7}$$
$$\xi_{C_j,2} := in(C_j, \mathcal{F}) \wedge \neg in(C_j, \mathcal{C}). \tag{8}$$
$$\xi_{C_j,3} := \neg in(C_j, \mathcal{C}). \tag{9}$$

Here the in is the set membership relation defined as $in(C_j, \mathcal{F}) := \bigvee_{C \in \mathcal{F}} C = C_j$. The task constraint for C_j in the trace reconstruction is then specified as

$$\psi_{C_j} := \begin{cases} \varphi_{C_j}, & \text{if } \xi_{C_j,1}, \\ \kappa_{C_j}, & \text{if } \xi_{C_j,2}, \\ \alpha_{C_j} \wedge \rho_{C_j}, & \text{if } \xi_{C_j,3}. \end{cases} \tag{10}$$

Finally, it is required that exactly the set of observed system input events on Tr occur in reconstructed traces. The set I of possible system input events is application dependent. For example, for the GPCA case study, $I = \{$ENQ(Q1, BOLUS),

ENQ(Q1, EMPTY), ENQ(Q1, NON-EMPTY)}. This constraint is defined as

$$\iota := \forall e \in I. \forall t \in \mathbb{R}_{\geq 0}. [on_{Tr}(e,t) \leftrightarrow \exists!(e',t').e' = e \wedge t' = t]. \tag{11}$$

Here, the $\exists!$ quantifier means "there exists one and only one." The behavior of the reconstructed system is then specified with the formula

$$\psi := \iota \wedge \psi_{C_1} \wedge \ldots \psi_{C_J}. \tag{12}$$

Proposition 1. *The formula ψ defined in Equation (12) defines the set $TR_{\mathcal{C}}$ of the possible system behaviors with the same input as observed on Tr, after suspected tasks in $\mathcal{C}$ are replaced with correct ones. That is, $TR_{\mathcal{C}} = \{tr \mid tr \models \psi\}$.*

The construction in this section is a combination of Steps 2 and 3 in the causality analysis framework (cf. Figure 2) for a given suspected faulty subset $\mathcal{C}$. The formula ψ in Equation (12) characterizes the set of reconstructed traces, whereas the satisfiability (unsatisfiability, resp.) result corresponds to whether the subset $\mathcal{C}$ is a cause with respect to Definition 8 (Definition 9, resp.). Due to Proposition 1, to check that the subset $\mathcal{C}$ is a cause according to Definition 8, it suffices to check that $\psi \wedge \varphi_S$ is satisfiable. To check that the subset $\mathcal{C}$ is a cause according to Definition 9, it suffices to check that $\psi \wedge \neg\varphi_S$ is unsatisfiable. State-of-the-art SAT/SMT solvers, e.g., Z3 [7], can be leveraged in solving the causality analysis problem, as shown in our previous work [26].

5 The GPCA Case Study

In this section, we use the GPCA case study to illustrate how the causality analysis problem is solved. We first show a few informal examples, then the formal definitions of the GPCA system, and finally the analysis using the causality analysis framework and trace reconstruction techniques from Sections 3 and 4.

ID	Task	Event	Time (ms)
1	BR	ENQ(Q1, BOLUS)	8500
2	PI	DEQ(Q1, BOLUS)	8502
3	PI	ENQ(Q2, BOLUS)	8503
4	PM	DEQ(Q2, BOLUS)	8701
5	PM	START	8702
6	ER	ENQ(Q1, EMPTY)	17000
7	PI	DEQ(Q1, EMPTY)	17004
8	PI	ENQ(Q3, EMPTY)	17005
9	AC	DEQ(Q3, EMPTY)	17007
10	AC	ALARM	17008
11	AC	STOP	17008
12	PM	STOP	17701

Table 3. A Sample Trace for GPCA

A sample trace we will analyze is shown in Table 3. The ID column is added for the convenience of reference. The Task column indicates which task has produced the corresponding event. The Time column is the timestamp for the corresponding event. On this trace, a bolus request is detected at 8500 ms, and an infusion session starts at 8702 ms. An empty reservoir is detected at 17000 ms, and an alarm is raised at 17008 ms, together with a STOP event from AC which ends the infusion session. The STOP event from PM does not affect the pump operation in this case.

5.1 Informal Causality Analysis Examples

It can be easily verified that the trace shown in Table 3 satisfies the GPCA safety property in Equation (1). Now we show the causality analyses via a series of examples, based on variants of the trace observed in Table 3.

Example 1 (Faulty tasks, no system failure). Given a trace as observed in Table 3, with Event 12 missing. In this case, PM is faulty by not sending the STOP event. However the system property is not violated since the AC task detects an empty reservoir message and alarms and stops the pump motor. □

According to Step 1 of the causality analysis framework, there is no need for subsequent causality analysis.

Example 2 (Single faulty task caused system failure). Consider a trace where only Event 1 and Event 2 in Table 3 are observed. In this case, the PI task fails to move the bolus request event from Q1 to Q2, read by the PM task. Subsequently the PM task does not perform any actions, since it does not know there is a bolus request. In this case the PI task is faulty while PM is not. □

Example 3 (Multiple faulty tasks jointly caused system failure). Consider the trace in Table 3 with Events 3–5 and Events 10–12 missing. In this case, the PI task is faulty by not moving the bolus request message to Q2. The AC task is faulty by not delivering the ALARM and STOP events. However, replacing neither the PI nor the AC task individually could make the system failure disappear. Both PI and AC must be replaced with good ones for the system failure to disappear. □

Example 4 (Multiple faulty tasks, but only one caused system failure). In the example in Subsection 2.1, a trace with six events $\{(\text{ENQ}(\text{Q1, BOLUS}), 8500), (\text{DEQ}(\text{Q1, BOLUS}), 8502), (\text{ENQ}(\text{Q3, EMPTY}), 8503), (\text{DEQ}(\text{Q3, EMPTY}), 8701), (\text{ALARM}, 9760), (\text{STOP}, 9760)\}$ is shown. In this example, both PI and AC are faulty. However AC's faulty behavior would not have been triggered in the first place if PI were not faulty, and thus should not be considered as a cause for system failure. In the meanwhile, although it is the PM task's job to send the START event, it should not be the cause of system failure in this case either since it is not a faulty task: it does not receive the bolus request message in the first place, due to PI's fault. □

Example 5 (Multiple faulty tasks, but only one caused system property violation). Consider the trace in Table 3 with only Events 1–9 observed. In this case, both the AC and the PM tasks are faulty by not delivering the corresponding events. In this case, if the AC task were not faulty, the system failure would disappear. □

Examples 4 and 5 show the improvement in precision that we have achieved using causality analysis: not all of the identified faulty tasks are the culprits for the system failure. By ruling out the tasks which are not culprits, the subsequent analysis for the system failure can be focused on the identified minimal culprits.

5.2 Formal Definitions for GPCA System

We first define constraints φ_{AC}, φ_{PI}, and φ_{PM} for the three tasks that can fail. We do not consider faults in BR and ER in this paper. The constraint for each task consists of two parts:
(1) what would a task do when it reads a message from a queue, and
(2) when would a task read a message if there is one in the corresponding queue.

For Part (1), the AC task's constraint is specified with

$$\begin{aligned}\tau_{AC} := \forall(e,t).[e = \text{DEQ(Q3, EMPTY)} \rightarrow &\exists!(e',t').[e' = \text{STOP} \wedge t' \leq t+10] \wedge \\ &\exists!(e'',t'').[e'' = \text{ALARM} \wedge t'' \leq t+10]].\end{aligned} \tag{13}$$

It is interpreted as, as long as there is a DEQ(Q3, EMPTY) event on the trace, there must be a STOP event within 10 ms and an ALARM event within 10 ms. Similarly, the PI task's constraint is specified with

$$\begin{aligned}\tau_{PI} := \forall(e,t).[&[e = \text{DEQ(Q1,BOLUS)} \rightarrow \exists!(e',t').[e' = \text{ENQ(Q2, BOLUS)} \wedge t' \leq t+10]] \wedge \\ &[e = \text{DEQ(Q1, EMPTY)} \rightarrow \exists!(e',t').[e' = \text{ENQ(Q3, EMPTY)} \wedge t' \leq t+10]] \wedge \\ &[e = \text{DEQ(Q1, NON-EMPTY)} \rightarrow \exists!(e',t').[e' = \text{ENQ(Q3, NON-EMPTY)} \wedge t' \leq t+10]]].\end{aligned} \tag{14}$$

The PM task's constraint is specified with

$$\begin{aligned}\tau_{PM} := \forall(e,t).[e = \text{DEQ(Q2,BOLUS)} \rightarrow &\exists!(e',t').[e' = \text{START} \wedge t' \leq t+10 \wedge \\ &\exists!(e'',t'').[e'' = \text{STOP} \wedge 8990 \leq t''-t' \leq 9010]]].\end{aligned} \tag{15}$$

For Part (2), the aperiodic task PI and the two periodic tasks PM and AC have different behaviors when a message the task should read is present in the corresponding queue. In the GPCA implementation, PI is set to preempt lower priority tasks to process the message and put it to the corresponding queues. This behavior is specified with the constraint γ_{PI} in Equation (16). The value 20 ms is due to the possible preemption of PI by the even higher priority PM.

$$\begin{aligned}\gamma_{PI} := \forall(e,t).[&[e = \text{ENQ(Q1,BOLUS)} \rightarrow \exists!(e',t').[e' = \text{DEQ(Q1, BOLUS)} \wedge t' \leq t+20]] \wedge \\ &[e = \text{ENQ(Q1, EMPTY)} \rightarrow \exists!(e',t').[e' = \text{DEQ(Q1, EMPTY)} \wedge t' \leq t+20]] \wedge \\ &[e = \text{ENQ(Q1, NON-EMPTY)} \rightarrow \exists!(e',t').[e' = \text{DEQ(Q1, NON-EMPTY)} \wedge t' \leq t+20]]].\end{aligned} \tag{16}$$

For periodic tasks AC and PM, when a message is present in a queue, it may be processed nearly two periods later as illustrated in Figure 3 (for AC). AC is dispatched at the beginning of each period, but can only execute in boxed durations where the FreeRTOS schedules AC by switching it IN and OUT. Suppose there is a check function in the AC's implementation to check if there is an EMPTY message in Q3. It may happen that the check function is executed at the very beginning of one period, slightly before the message EMPTY is put to Q3 (in which case AC does not know there is a message EMPTY during the rest of the period); in the following period, the check function may be executed at the very end of the period.

This scenario is the worst case for a periodic task to detect a message in a queue. Therefore, the constraints for AC and PM are respectively:

$$\begin{aligned}\gamma_{AC} := \forall(e,t).[&[e = \text{ENQ(Q3, EMPTY)} \rightarrow \exists!(e',t').[e' = \text{DEQ(Q3, EMPTY)} \wedge t' < t+1000]] \wedge \\ &[e = \text{ENQ(Q3, NON-EMPTY)} \rightarrow \exists!(e',t').[e' = \text{DEQ(Q3, NON-EMPTY)} \wedge t' < t+1000]]].\end{aligned} \tag{17}$$

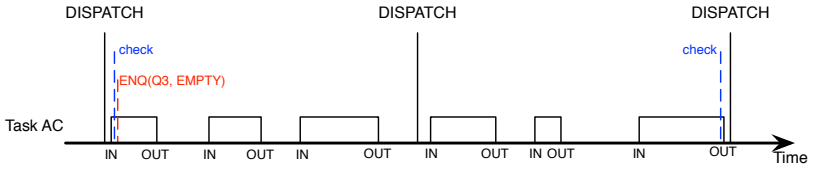

Fig. 3. Illustration of Message Processing for Periodic Tasks in FreeRTOS

$$\gamma_{PM} := \forall(e,t).[e = \text{ENQ(Q2, BOLUS)} \rightarrow \exists!(e',t').[e' = \text{DEQ(Q2, BOLUS)} \wedge t' < t + 600]]. \quad (18)$$

Notice that we have asymmetric treatments on the response time requirements for Part (1) and Part (2) of a task's constraint. For Part (1), a hard response time of 10 ms is imposed; for Part (2), the task scheduling in FreeRTOS is considered. This is due to the view that for Part (1), a task knows that a message has arrived and is thus required to deliver the corresponding event within the imposed response time; Part (2) of the constraint reflects the fact that a task's worst case delay to detect a message in a queue.

With Equations (13)–(18), the complete task constraints are defined as:

$$\varphi_{AC} := \tau_{AC} \wedge \gamma_{AC}, \quad (19)$$

$$\varphi_{PI} := \tau_{PI} \wedge \gamma_{PI}, \quad (20)$$

$$\varphi_{PM} := \tau_{PM} \wedge \gamma_{PM}. \quad (21)$$

The GPCA system is then defined as $S = \langle BR, ER, AC, PI, PM \rangle$. For each task C, its sets I_C and O_C of input/output events, as well as its constraint φ_C, are shown in Table 4.

Task C	Input I_C	Output O_C	φ_C
BR		{ENQ(Q1, BOLUS)}	True
ER		{ENQ(Q1, EMPTY), ENQ(Q1, NON-EMPTY)}	True
AC	{DEQ(Q3, EMPTY), DEQ(Q3, NON-EMPTY)}	{ALARM, STOP}	φ_{AC} (19)
PI	{DEQ(Q1, *)}	{ENQ(Q2, BOLUS), ENQ(Q3, EMPTY), ENQ(Q3, NON-EMPTY)}	φ_{PI} (20)
PM	{DEQ(Q2, BOLUS)}	{START, STOP}	φ_{PM} (21)

Table 4. Tasks in GPCA Case Study

5.3 Formal Causality Analysis

An instance of the causality analysis problem is defined by a tuple $\langle S, \varphi_S, Tr, CD \rangle$. Now we show the application of the causality analysis framework and trace reconstruction technique to solve the causality analysis problem in Example 4.

The system property is defined as in Equation (1). The trace is Tr = {(ENQ(Q1, BOLUS), 8500), (DEQ(Q1, BOLUS), 8502), (ENQ(Q3, EMPTY), 8503), (DEQ(Q3, EMPTY), 8701), (ALARM, 9760), (STOP, 9760)}. The causality definition CD is the main contributory cause in Definition 9. In Step 1 of the causality analysis frame work, the set $\mathcal{F} = \{PI, AC\}$ of faulty tasks is identified. In

Step 2, for each non-empty subset $\mathcal{C} \subseteq \mathcal{F}$, the formula ψ in Equation (12) is constructed according to the Equations (5) through (12) and information from S, Tr, $\mathcal{C}$ and $\mathcal{F}$. We now discuss the construction for the two application dependent constraints ρ_{C_j} and α_{C_j} not discussed in Section 4.

Defining "Removing" and "Adding" Constraints. As discussed in Section 4, one condition for an event to be removed from a trace is that it is not triggered by any other events. The trigger relation between timestamped input and output for a task is derived from the task's constraint. For example, the constraint for AC specifies that the STOP event and ALARM event must be delivered within 10 ms once the ENQ(Q3, EMPTY) event is read from queue Q3, as defined in Equation (13). In this case, the trigger relation is expressed as

$$\begin{aligned} trig_{AC} := \{((\text{DEQ}(\text{Q3, EMPTY}), t), (\text{STOP}, t')) \mid \text{ for all } t' \leq t + 10\} \\ \cup \{((\text{DEQ}(\text{Q3, EMPTY}), t), (\text{ALARM}, t')) \mid \text{ for all } t' \leq t + 10\}. \end{aligned} \tag{22}$$

Similarly, the trigger relations for PI and PM are defined as follows.

$$\begin{aligned} trig_{PI} := \{((\text{DEQ}(\text{Q1, BOLUS}), t), (\text{ENQ}(\text{Q2, BOLUS}), t')) \mid \forall t' \leq t + 10\} \\ \cup \{((\text{DEQ}(\text{Q1, EMPTY}), t), (\text{ENQ}(\text{Q3, EMPTY}), t')) \mid \forall t' \leq t + 10\} \\ \cup \{((\text{DEQ}(\text{Q1, NON-EMPTY}), t), (\text{ENQ}(\text{Q3, NON-EMPTY}), t')) \mid \forall t' \leq t + 10\}. \end{aligned} \tag{23}$$

$$trig_{PM} := \{((\text{DEQ}(\text{Q3, EMPTY}), t), (\text{START}, t')) \mid \forall t' \leq t + 10\}. \tag{24}$$

The constraint for traces where the events produced by a faulty task C_j are removed is specified with

$$\begin{aligned} \rho_{C_j} := \forall e \in O_{C_j}. \forall t \in \mathbb{R}_{\geq 0}. [[\neg \exists (e', t'). ((e', t'), (e, t)) \in trig_{C_j}] \rightarrow \\ \neg \exists (e'', t''). e'' = t \wedge t'' = t]. \end{aligned} \tag{25}$$

Informally, this constraint means that if there is no trigger for event e at time t, then it should not occur on any reconstructed traces.

For "adding" events to a trace, a task must only deliver output events when it is activated by the FreeRTOS scheduler. This piece of information is unavailable to offline analysis. We assume that the FreeRTOS scheduler would schedule each task the same as on the observed trace Tr. The instance at which an event can be produced on a reconstructed trace is then limited both by the task response time and its activation time. For example, if AC reads the EMPTY message at time 8701 ms, and it is observed on Tr that AC is active in time ranges [8700 ms, 8703 ms], [8709 ms, 8712 ms], etc., then in addition to the 10 ms deadline to deliver the events ALARM and STOP in the range [8701 ms, 8711 ms], AC can only produce the events during the ranges of [8701 ms, 8703 ms] or [8709 ms, 8711 ms]. The constraint for this requirement is obtained by augmenting the task constraint with the time information. For example, for AC, the specification is

$$\begin{aligned} &\alpha_{AC} := \ \gamma_{AC} \wedge \forall (e, t). [e = \text{DEQ(Q3, EMPTY)} \rightarrow \exists (e_i^1, t_i^1), (e_o^1, t_o^1), (e_i^2, t_i^2), (e_o^2, t_o^2) \in Tr. \\ &[e_i^1 = e_i^2 = \text{IN}(AC) \wedge e_o^1 = e_o^2 = \text{OUT}(AC) \wedge t \leq t_o^1 \wedge t \leq t_o^2 \wedge \\ &\neg \exists (e_i', t_i'), (e_o', t_o') \in Tr. [[e_i' = \text{IN}(AC) \wedge t_i^1 < t_i' \leq t_o^1] \ \vee [e_o' = \text{OUT}(AC) \wedge t_i^1 \leq t_o' < t_o^1]] \wedge \\ &\neg \exists (e_i', t_i'), (e_o', t_o') \in Tr. [[e_i' = \text{IN}(AC) \wedge t_i^2 < t_i' \leq t_o^2] \ \vee [e_o' = \text{OUT}(AC) \wedge t_i^2 \leq t_o' < t_o^2]]] \rightarrow \\ &\exists ! (e^1, t^1). [e^1 = \text{STOP} \wedge [\max(t, t_i^1) \leq t^1 \leq \min(t + 10, t_o^1)]] \wedge \\ &\exists ! (e^2, t^2). [e^1 = \text{ALARM} \wedge [\max(t, t_i^2) \leq t^2 \leq \min(t + 10, t_o^2)]]]. \end{aligned} \tag{26}$$

The third and fourth lines of Equation (26) constraint the pairs (e_i^1, e_o^1) and (e_i^2, e_o^2) to bound single time chunks of execution for task AC (i.e., a single box in Figure 3). The constraint α_{AC} means, if an event $e = \text{DEQ}(\text{Q3}, \text{EMPTY})$ at time t is on the reconstructed trace, then its corresponding events (STOP and ALARM) must be produced by AC within the 10 ms deadline, as well as when AC is active. α_{PM} and α_{PI} can be similarly defined.

Causality Analysis Result. For the constructed constraint ψ for each case, we have manually proved that $\psi \wedge \neg\varphi_S$ is unsatisfiable for the cases when $\mathcal{C} = \{PI\}$ or $\mathcal{C} = \{PI, AC\}$, while it is satisfiable for $\mathcal{C} = \{AC\}$. This result shows that both $\{PI\}$ and $\{PI, AC\}$ are culprits, according to the main contributory cause definition (Definition 9). These two subsets are collected as the set of culprits in Step 3 of the causality analysis framework. In Step 4, the two culprits $\{PI\}$ and $\{PI, AC\}$ are minimized to be $\{PI\}$ only. This result is consistent with our intuition in that it is the PI task's fault in the first place to put a bogus empty reservoir message to Q3, which triggers AC's fault.

6 Discussion

FreeRTOS Scheduling in Trace Reconstruction. When defining the "adding" constraint, we have assumed that the FreeRTOS scheduler would schedule all the tasks the same as on the observed trace during trace reconstruction. This assumption must be made due to the unavailability of FreeRTOS scheduling information should the system be rerun. Without this assumption, the analysis would have to include the FreeRTOS scheduler as part of the system and model it (or even the entire FreeRTOS operating system) as a component too. This is by itself a challenging task and is beyond the scope of this paper.

Causality Analysis vs. Fault Diagnosis. Unlike many approaches to fault diagnosis, we address the case of black-box components [25], in which internal flows of information between component input and output are unknown. In this case, techniques based on computing fault propagation paths lead to an over-approximation of cause-effect chains. The causality analysis we proposed in the paper improves the precision of this over-approximation.

Alternative Ways to Trace Reconstruction. Our causality analysis is based on counterfactual reasoning [16], where the system behavior is reevaluated on the possible alternative traces. A commonly used criterion for constructing alternative traces is to measure the *similarity* between the reconstructed traces and the actual observed one. Causality analysis is only performed on alternative traces which are *similar* to the observed trace. However, the notion of *similarity* is subjective, reflected by the rules used for the trace reconstruction.

In our approach, the trace reconstruction rules (R1)–(R3) represent a view at the *task level*: a faulty task is replaced with a good one, and all its events, except for system inputs, are reconstructed via the "removing" and "adding" operations. In contrast, one could perform trace reconstruction at the finer grained *event level*: the trace under analysis is scanned through until the first occurrence e_f of an event that leads to task failure is found, and trace reconstruction is started

only from that particular occurrence; every event that happens before e_f is kept the same as observed.

Compared to the set $TR^t_{\mathcal{C}}$ of reconstructed traces produced by task-level trace reconstruction, the event-level trace reconstruction could produce a smaller set $TR^e_{\mathcal{C}} \subseteq TR^t_{\mathcal{C}}$ of traces *more similar* to the observed trace. Using the finer-grained event-level trace reconstruction, it is comparatively more likely to establish a necessary cause (Definition 9), since less traces have to be examined for the "for all" quantification to be satisfied. On the other hand, it is comparatively less likely to establish a contributory cause (Definition 8), since less possible alternative traces can be used to satisfy the "exists" quantification.

Full Observability. *Full observability* involves two assumptions: (1) we are able to put probes at the interfaces of components so that each event is observable, and (2) the recording facility is capable of capturing all events at component interfaces. The first assumption is by our consideration of black-box components, where internal events within a component is not observable, but the events at its interface are observable. Violations to the second assumption may lead to undetected faulty components, yielding a smaller set $\mathcal{F}$ of faulty components. This may possibly lead to spuriously identified culprits.

Causality Definitions. Several causality definitions have been discussed in previous work [13,24,11,15,4,26], all based on the notion of counterfactual reasoning [16]. We in this work used the main contributory cause (Definition 9), but showed that the causality analysis framework is parametric to the causality definition of choice. The capability of using different causality definitions in the analysis increases the flexibility for the investigator to make reasonable arguments.

The definitions of contributory and main contributory causes express different levels of necessity needed to judge for the cause. If the sufficiency of causality definition is of concern, one could use alternative trace reconstruction rules and causality definitions.

Scalability. While we are working on larger case studies to gain empirical results on the scalability of our approach, we foresee two limitations. First, for a given subset $\mathcal{C}$ of suspected faulty components, the complexity of computing whether $\mathcal{C}$ is a necessary cause is coNP-complete for propositional logic [26] and undecidable in general for first order logic [18]. This limits the scalability of our approach to the capability of state-of-the-art SAT/SMT solvers, such as Z3 [7]. Second, we have shown in the paper the direct computation of the minimal culprit, which requires the explicit generation of the powerset of $\mathcal{F}$, limiting the possible number of faulty components that can be analyzed practically. Further studies on algorithms exploiting the underlying structure of the sets of reconstructed traces could potentially speed up the explicit computation.

7 Related Work

Halpern and Pearl [13] were among the first to introduce the counterfactual reasoning for causes into the engineering domain. Some later development [15,4] is

based on the notions in [13]. In this work, we formally characterized the set of reconstructed traces, and showed that causality can be defined based on the set of reconstructed traces. One advantage of our work is the explicit treatment of real-time systems, which is not presented in previous work on causality analysis. Timestamps are considered as variables so that constraints on timestamped events symbolically characterize sets of traces that satisfy the constraints.

The treatment of trace reconstruction is another difference between our work and previous ones [15,11]. In [15], each occurrence of an event on a trace is represented by a boolean variable e, indicating whether the event is present on the trace (e is true) or not (e is false). The underlying component behaviors are not considered in [15]. Similarly, in [11], the trace reconstruction rules place a more rigid requirement than in this paper, which may occasionally lead to undesired analysis result, as we have discussed in [26]. On the other hand, the work in [11] in addition defines *horizontal* causality between one component's failure and another's, which is not discussed in any other work in causality analysis. Also, our Hypothesis 1 is due to [11].

The result of causality analysis naturally provides an *explanation* to the system failure: which components' faulty behaviors are the causes to the system property violation. The work in [4] provides an application in explaining counterexamples from formal verification of system properties specified in linear temporal logics (LTL) [20]. We believe the approach in [4] can be extended to the setting in this paper.

8 Conclusion

We proposed the causality analysis problem for black-box component-based systems. By using causality analysis we are able to establish causal relationship between component failures and system failure. We provided a formal analysis framework to solve the causality analysis problem, and detailed the trace reconstruction rules for the analysis for real-time systems. We illustrated our approach with the GPCA case study. In the future, we are planning to enhance the application of the analysis by providing tool support for safety-critical systems in the medical device domain.

Acknowledgement. We would like to thank FDA researchers Paul L. Jones and Yi Zhang for their motivating discussions on the causality analysis problem and help in explaining infusion pumps and the GPCA safety requirement document [12].

References

1. R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, Jan. 1996.
2. Atmel Corporation. AT91SAM7S-EK Evaluation Board User Guide. `http://www.atmel.com/Images/doc6112.pdf`, 2007.
3. R. Barry. FreeRTOS User Manual. `http://www.freertos.org`.

4. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. In *CAV'09*, pages 94–108. Springer, 2009.
5. S. Bhattacharyya, Z. Huang, V. Chandra, and R. Kumar. A discrete event systems approach to network fault management: detection and diagnosis of faults. In *American Control Conference*, volume 6, pages 5108–5113, 2004.
6. J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
7. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*. Springer, 2008.
8. A. Dubey, G. Karsai, R. Kereskenyi, and N. Mahadevan. Towards a real-time component framework for software health management. Technical Report ISIS-09-111, Vanderbilt University, 2009.
9. FDA. FDA MAUDE Database. `http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfmaude/search.cfm`.
10. Generic PCA Infusion Pump Reference Implementation. `http://rtg.cis.upenn.edu/medical/gpca/gpca.html`.
11. G. Gössler, D. L. Métayer, and J.-B. Raclet. Causality analysis in contract violation. In *Runtime Verification*, 2010.
12. Safety Requirements for the Generic PCA Pump. `http://rtg.cis.upenn.edu/gip-docs/Safety_Requirements_GPCA.doc`.
13. J. Y. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 56(4):843–887, December 2005.
14. M. Kuntz, F. Leitner-Fischer, and S. Leue. From probabilistic counterexamples via causality to fault trees. In *Computer Safety, Reliability, and Security*. Springer Berlin Heidelberg, 2011.
15. F. Leitner-Fischer and S. Leue. Causality checking for complex system models. Technical Report soft-12-02, University of Konstanz, 2012.
16. D. Lewis. *Counterfactuals*. Wiley-Blackwell, 2nd edition, 2001.
17. N. Mahadevan, S. Abdelwahed, A. Dubey, and G. Karsai. Distributed diagnosis of complex systems using timed failure propagation graph models. In *the IEEE Systems Readiness Technology Conference*, pages 1–6, 2010.
18. E. Mendelson. *Introduction to Mathematical Logic*. Chapman and Hall/CRC, 4th edition, 1997.
19. J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2009.
20. A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS '77*, pages 46–57, 1977.
21. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
22. R. Riegelman et al. Contributory cause: unnecessary and insufficient. *Postgrad Med*, 66(2):177, 1979.
23. M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, 1996.
24. J. Tian and J. Pearl. Probabilities of causation: Bounds and identification. *Annals of Mathematics and Artificial Intelligence*, 28:287–313, 2000.
25. S. Tripakis. A combined on-line/off-line framework for black-box fault diagnosis. In *Runtime Verification*, volume 5779 of *LNCS*, pages 152–167, 2009.
26. S. Wang, A. Ayoub, R. Ivanov, O. Sokolsky, and I. Lee. Contract-based blame assignment by trace analysis. In *HiCoNS*, pages 117–125, 2013.