A Compositional Approach to Diagnosing Faults in Cyber-Physical Systems

Josefine B. Graebener[1], Inigo Incer[2], and Richard M. Murray[1]

[1] California Institute of Technology, Pasadena CA 91125, USA
jgraeben@caltech.edu, murray@cds.caltech.edu
[2] University of Michigan, Ann Arbor MI 48109, USA iir@umich.edu

Abstract. Identifying the cause of a system-level failure in a cyber-physical system (CPS) can be like tracing a needle in a haystack. This paper approaches the problem by assuming that the CPS has been designed compositionally and that each component in the system is associated with an assume-guarantee contract. We exploit recent advances in contract-based design that show how to compute the contract for the entire system using the component-level contracts. When presented with a system-level failure, our approach is able to efficiently identify the components that are responsible for the system-level failure together with the specific predicates in those components' specifications that are involved in the fault. We implemented this approach using Pacti and demonstrate it through illustrative examples inspired by an autonomous vehicle in the DARPA urban challenge.

Keywords: assume-guarantee contracts · diagnostics.

1 Introduction

Any safety-critical system requires efficient detection of system abnormalities and faults to avoid dangerous situations and safety hazards. The rising adoption of autonomy has only increased the need for such diagnostics [5]. Faults are defined as deviations from the correct, expected system behavior, observable in at least one system property or parameter [27]. Diagnostics refers to the process of identifying, analyzing, and resolving problems that become apparent during the operation of a system, product, or process. Diagnosing the cause of a system-level failure in a complex CPS can be a difficult process.

Fault diagnosis consists of three areas: detection, isolation, and identification [12]. Detection refers to identifying when and where a fault occurs from the observable system output. Isolation considers the location of the fault, and identification refers to finding the type, shape, and size of the fault. There are five main fault diagnosis methods: model-based, signal-based, knowledge-based, hybrid, and active. In 1971, Beard introduced model-based fault diagnosis with the intent to replace hardware redundancy by analytical redundancy [1]. Model-based fault diagnosis uses different techniques to monitor the actual system

outputs and compare them to the predicted values. Signal-based fault diagnosis uses measured signals instead of input-output models, and extracts features (or patterns) from a signal to make a diagnostic decision [12]. Knowledge-based fault diagnosis consists of a knowledge base and an inference engine [6]. Hybrid approaches are a combination of the above-mentioned methods [12]. Active fault detection is concerned with designing auxiliary input vectors to reveal faults [22].

Diagnostics has been studied extensively in computer science and engineering. Some early and influential works include [10] and [25]. In [10], a model-based approach to diagnose faults in complex systems by observing symptoms and using reasoning techniques was introduced in 1987 by De Kleer and Williams. In 1987, Reiter developed a formal logical framework to diagnose faults consisting of three main components: a knowledge base, an observation base, and a set of inference rules [25]. In formal methods, the problem of explaining why for certain robot specifications no implementing control strategy exists has been studied in [24], while 'repairing' specifications has been studied in [3]. In [20], assume-guarantee contract operators have been used for specification repair. Recently, TRACE, a tool for requirements analysis was introduced in [28], which uses SMT solvers to analyze system guarantees.

This paper presents a diagnostics approach that utilizes assume-guarantee reasoning and leverages the syntax of specifications to facilitate tracing the causes of violated system-level guarantees to potential subsystems. The use of contracts in diagnostics simplifies the attribution of blame. For example, suppose that we have a trace for a component. If this trace violates the assumption of the component, the component cannot be blamed for any undesired behavior. On the other hand, under satisfied assumptions, the behavior has to satisfy the promised guarantees. If the component does not deliver its guarantees in this case, then it did not satisfy its specification. Now we can further analyze the component and determine whether the implementation was faulty or if anything was missed when defining the specification.

Problem definition. Suppose we implement a system with n interconnected components having contracts $\mathcal{C}_i = (\bigwedge_j a_j^i, \bigwedge_j g_j^i)$ for $i \in \{1, \ldots, n\}$, where a_j^i and g_j^i represent the j-th assumption and guarantee, respectively, of component i. Suppose that the specification of the entire system is given by $\mathcal{C}_S = (\bigwedge_j a_j, \bigwedge_j g_j)$ and that we have access to a log file `Log` containing values for a subset of system variables such that the assumptions of $\mathcal{C}_S$ are satisfied, but there is at least one guarantee of $\mathcal{C}_S$ that is violated. This paper introduces a technique to identify both the contract $\mathcal{C}_k$ and its specific assumptions and guarantees that were responsible for the system-level fault. The technique can indicate exactly the predicates that need to be evaluated to diagnose the failure, as opposed to checking every single assumption and guarantee of all components in the system. While the application of contracts in runtime verification and diagnostics has been pursued—see, e.g., [7,8,9,13,15]—our work is the first to exploit the explicit computation of the contract operation of composition in order to determine not only what component in the system is the likely cause of the system-level fault,

but also the specific predicates in the component's contract that are responsible for the fault.

Our problem setup assumes that we have a log file witnessing a system-level failure and contracts for each component. We may ask, why should not we simply check every assumption and guarantee of all components? We believe that the targeted approach discussed in this paper is preferable for various reasons. First, the log file can contain several errors that are not related to the system-level issue we are interested in debugging. Lacking the means to pinpoint exactly the predicates that are involved in the system-level failure can lead designers to long debugging campaigns of unrelated, but tempting, issues, resulting in distractions and loss of time, which may be costly in time-sensitive projects or when deadlines approach. Second, suppose there is a specific system-level guarantee we want to monitor. The methods discussed in this paper can locate exactly which predicates should be monitored in the system to ascribe blame to a component for the violation of the property we are tracking. Knowledge of these internal predicates can be used to instrument the system to monitor the desired predicates when running a second test, i.e., this methodology can be used in designing test campaigns.

Contributions. We propose a diagnostics methodology based on contracts that enables a systematic search over system variables to trace violated system guarantees back to the responsible component. Our approach reduces the number of predicates and components that need to be evaluated during fault localization by exploiting the explicit computation of the composition operation of contracts. We implement this methodology using Pacti [18], a tool for compositional reasoning over assume-guarantee contracts. Finally, we demonstrate the effectiveness of our approach through illustrative examples and case studies inspired by autonomous vehicle behavior in the DARPA Urban Challenge [4]. A preliminary version of this work appeared in Chapter 5 of [14]. The presentation has been expanded and the current case study has not appeared before.

2 Background

The framework presented in this paper is based on contract-based design, first introduced as a design methodology for modular software systems [11,19,21] and later extended to cyber-physical systems [23,26].

Definition 1. (Assume-Guarantee Contract [2,16]) Let T be the language used to express specifications in our system. We assume this language has Boolean semantics. A *contract* is a pair $\mathcal{C} = (a, g) \in T^2$, where a are the assumptions, and g the guarantees. A model $E \models a$ is said to be an *environment* of the contract $\mathcal{C}$. A model $M \models a \rightarrow g$ is said to be an *implementation* of the contract $\mathcal{C}$, meaning that M provides the specified guarantees when it operates in an environment that satisfies the contract's assumptions. There exists a preorder of contracts: we say $\mathcal{C}_1$ is a refinement of $\mathcal{C}_2$, denoted $\mathcal{C}_1 \leq \mathcal{C}_2$, if $(a_2 \leq a_1)$ and $(a_1 \rightarrow g_1 \leq a_2 \rightarrow g_2)$, where the preorder $\phi \leq \phi'$ on formulas ϕ

and ϕ' means that $\phi \rightarrow \phi'$ is a tautology. Contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ are said to be equivalent if $\mathcal{C}_1 \leq \mathcal{C}_2$ and $\mathcal{C}_2 \leq \mathcal{C}_1$.

As shown in [17], the contract algebra is a Stone algebra, but not a Boolean algebra. A contract has three possible evaluations—not two—as requirements normally do (i.e., a requirement is either SAT or UNSAT). A contract can evaluate to either `FAIL`, `ACTIVE`, or `IDLE`. It evaluates to `FAIL` when the assumptions are satisfied but the guarantees are not. It evaluates to `ACTIVE` when both the assumptions and guarantees are satisfied. A contract evaluates to `IDLE` when the assumptions are not satisfied. Given a system-level failure, the diagnostics process corresponds to identifying the component contract that evaluates to `FAIL`. It is key to our diagnostics process to compute the composition operation of contracts. A recent breakthrough in contract-based design showed how to compute this operation efficiently—see [18].

The results of this paper are implemented in Pacti, an open-source Python package for compositional system analysis and design. Components are defined using assume-guarantee contracts, and contract operations can be performed, such as composition, merging, and quotient. Contracts in Pacti are defined over a term algebra T with Boolean semantics. Pacti's basic data structure is the IO contract.

Definition 2. (IO Contract [18]) Let V be a set of variables. An *IO contract* is the tuple $(I, O, \mathfrak{a}, \mathfrak{g})$, where $I, O \in V$ are disjoint sets of input and output variables respectively, $\mathfrak{a} \in T$ a set of assumptions, and $\mathfrak{g} \in T$ a set of guarantees. The terms in the assumptions only refer to input variables, and the terms in the guarantees only refer to input and output variables.

The key contract operation for system-level design is composition, which yields the contract of a system obtained by interconnecting components represented using contracts. The composition of contract $\mathcal{C} = (a, g)$ and contract $\mathcal{C}' = (a', g')$ can be directly computed as

$$\mathcal{C}_c = \mathcal{C} \parallel \mathcal{C}' = ((a \wedge a') \vee (a \wedge \neg g) \vee (a' \wedge \neg g'), (g \vee \neg a) \wedge (g' \vee \neg a')). \tag{1}$$

This yields the most refined contract that a system comprising two components, M and M', will satisfy, provided that M and M' were implemented such that they satisfy their corresponding contracts $\mathcal{C}$, and $\mathcal{C}'$, respectively. Whereas this operation satisfies optimality criteria, the authors of [18] observe that the output of composition should be a contract expressed using only the variables that lie at the interface of the resulting system, which equation (1) does not provide.

To express contract composition using only interface variables, equation (1) can be relaxed by either refining the assumptions, relaxing the guarantees, or both. For two assumptions a and a', we say that a' refines a, denoted $a \geq a'$, if the denotation set corresponding to a' is a subset of the set corresponding to a. Similarly, for guarantees g and g', we say that g' is a relaxation of g, $g \leq g'$, if the denotation set corresponding to g is a subset of the set corresponding to g'. When thinking about assumptions and guarantees each as conjunctions

of terms (or constraints), refining a contract informally corresponds to either assuming *less*, guaranteeing *more*, or both. Pacti makes use of this contract relaxation to eliminate internal variables so that the returned composition is expressed only using the interface variables of the system. Any relaxed contract that is computed in this way will be satisfied by a correct implementation of the system. Nevertheless, for a contract to be useful, we need to compute the relaxation systematically, as in the extreme case a contract that guarantees every possible behavior is a valid, yet pointless refinement in the context of capturing the system's behavior.

When computing a composition of contracts $\mathcal{C}$ and $\mathcal{C}'$, the assumptions of the composed contract are given as follows:

$$a_c = \overbrace{(a \wedge a')}^{\text{stem}} \vee \overbrace{(a \wedge \neg g') \vee (a' \wedge \neg g)}^{\text{failure terms}}. \tag{2}$$

We refer to the first term as the *stem*, as this is where the composed system should operate—where the assumptions of both components are satisfied. The second and third terms are referred to as *failure terms*, where each term refers to one of the components having its assumptions satisfied but not delivering their guarantees. As we want the composition to operate in the stem, Pacti uses the failure terms to eliminate from the stem the variables that are not part of the interface of the resulting system. These failure terms serve as the context for the elimination of variables in the stem—the details about this are contained in [18]. Once the failure terms are no longer needed to eliminate variables in the stem they are discarded, and we define the assumption of the relaxed contract as the transformed stem by refinement. The transformation of the guarantees follows a similar argument, but ensures that the variables are eliminated by relaxing the guarantees. The guarantees of a composition are given as follows:

$$g_c = (g \vee \neg a) \wedge (g' \vee \neg a') = \underbrace{(g \wedge g')}_{\text{stem}} \vee (\neg a \wedge g') \vee (\neg a' \wedge g) \vee (\neg a \wedge \neg a'),$$

where the *stem* again refers to the desired area of operation, when both components satisfy their guarantees. The stem may contain variables that should be eliminated. We can use the remaining terms as a *context* to transform the stem to eliminate these variables. The transformation of either assumptions or guarantees is carried out in Pacti by functions that take as arguments the term to be transformed, the variables that need to be eliminated, and the context that can be used to carry out such elimination. Pacti currently supports linear inequalities and propositional logic as the term algebras for expressing assumptions and guarantees. However, the framework is extensible to other formalisms by implementing the variable elimination routines discussed in [18].

3 Tracing System Guarantees

This section presents a methodological approach to modeling the system and its constituent components to support an effective fault diagnosis mechanism.

Our approach is based on the computation of contract operations in Pacti. We extended this tool to support diagnostics.

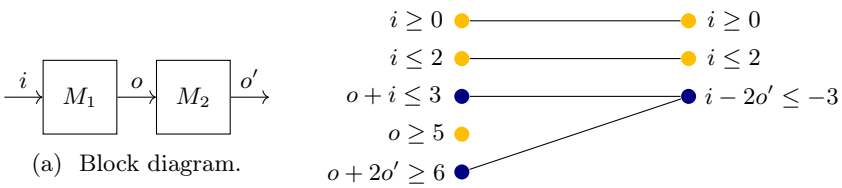
To begin, we assume that each component in the system is modeled as an IO contract. For example, for a component M, we can define the corresponding IO contract $\mathcal{C} = (I, O, \mathfrak{a}, \mathfrak{g})$, where the set $\mathfrak{a}$ contains the term $i \leq 2$ and the guarantee set $\mathfrak{g}$ contains the term $o \leq 2i + 1$, where $I = \{i\}$, and $O = \{o\}$ are the singleton sets of the input and output variables.

Definition 3. (Faulty component) Given a component M and the corresponding contract $\mathcal{C} = (I, O, \mathfrak{a}, \mathfrak{g})$, M is *faulty* if it contains a behavior that does not satisfy the guarantees $\mathfrak{g}$, but the assumptions $\mathfrak{a}$ are satisfied.

As discussed in the background section, Pacti uses a filtering procedure to determine the relevant context terms when computing the assumptions and guarantees of the composed contract. These context terms come from the assumptions and guarantees of the contracts being composed. We extended Pacti with an ID system to keep track of which context terms were used to generate a resulting term for the assumptions and guarantees of the composed contract. For each composition operation, we can thus define a composition graph that allows us to map the composed assumption and guarantee terms to the terms that were used in their transformation. A composition graph consists of a set of vertices, where each vertex corresponds to a term in the assumptions or guarantees of the individual contracts, and the composed contract. The edges in the composition graph connect vertices if the corresponding individual contract term was used to generate the composed contract term, shown as the edges from left to right in Figure 1b.

Definition 4. (Composition Graph) Let components M_1 and M_2 have corresponding IO contracts $\mathcal{C}_1 = (I_1, O_1, \mathfrak{a}_1, \mathfrak{g}_1)$ and $\mathcal{C}_2 = (I_2, O_2, \mathfrak{a}_2, \mathfrak{g}_2)$, with their composition given by $\mathcal{C} = (I, O, \mathfrak{a}, \mathfrak{g})$. A *composition graph* is a directed graph $G = (V, E)$, where each vertex in V corresponds to a term from the assumptions or guarantees of $\mathcal{C}_1$, $\mathcal{C}_2$, or $\mathcal{C}$. Specifically, $V_{i,a} \subseteq V$ and $V_{i,g} \subseteq V$ represent the assumptions $\mathfrak{a}_i$ and guarantees $\mathfrak{g}_i$ of component i, while $V_a \subseteq V$ and $V_g \subseteq V$ represent the composed assumptions and guarantees. For simplicity, we use the same symbol s to refer to both a term and its corresponding vertex $s \in V$. An edge $(u, v) \in E$ exists if the term u was used to generate term v. A path from vertex s to t in G, denoted $\texttt{path}(G, s, t)$, exists if there is a sequence of edges connecting the vertices.

Example 1. Given two components M_1 and M_2 and their inputs and outputs as illustrated in Figure 1a and their IO contracts as $\mathcal{C}_1 = (\{i\}, \{o\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \geq 0, i \leq 2\}$, and $\mathfrak{g}_1 = \{o + i \leq 3\}$ and $\mathcal{C}_2 = (\{o\}, \{o'\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_2 = \{o \leq 5\}$, and $\mathfrak{g}_2 = \{o + 2o' \geq 6\}$. The composition results in the contract $\mathcal{C} = (\{i\}, \{o'\}, \mathfrak{a}, \mathfrak{g})$, where $\mathfrak{a} = \{i \geq 0, i \leq 2\}$, and $\mathfrak{g} = \{i - 2o' \leq -3\}$. The composition graph corresponding to the composition $\mathcal{C}_1 \parallel \mathcal{C}_2$ is shown in Figure 1b.



(a) Block diagram.

(b) Composition graph.

Fig. 1: Block diagram and composition graph for Example 1. The yellow vertices correspond to assumption terms and the blue vertices correspond to guarantee terms.

From our experience, the transformation of an assumption or guarantee usually involves few context terms. This means that the resulting composition graph is very sparse. This allows us to trace a composition-level guarantee back to a small set of terms on the component level.

All operations in Pacti are computed under the assumption that the components operate correctly as specified by their contracts. The entire premise of testing lies in the difference between a 'perfectly' specified system and its real-world implementation. In a real-world system, specifications might be incomplete, or implementations might be faulty. A single component failure might present itself in multiple ways; it might result in a system-level guarantee violation, or it might not. If a component failure is latent, meaning it does not show itself in the system-level guarantee violation, our proposed framework cannot detect it. If the fault does show itself in a system-level guarantee violation, we can trace it to the responsible component(s) by tracking which relevant component's guarantees were violated under satisfied assumptions.

At this point, we can create a composition graph for the composition of two components and their corresponding IO contracts. To build the overall system, we need to compose multiple components. Contract composition is a binary operation. Therefore, to compose the entire system, we need to compose two components at a time, and then compose their composition with the next component. There are many ways of composing the same system, and the resulting contract for the composition is dependent on the order of composition. As Pacti hides internal variables, the composition has to be chosen carefully such that the variables necessary for future compositions are kept. Another important aspect that can guide the composition order is the availability of component data. When there is a lack of available information from inside a block of components it might be beneficial to compose these components first and treat them as a meta-component—a grouping of multiple components. If the analysis ends up pointing to this meta-component as the possible cause, a more detailed analysis can still be set up focusing on these components. For this framework, we assume that a composition order has been chosen. This order will be maintained for the remainder of the diagnostics process.

Definition 5. (Composition Order) The component contracts are given as a *composition order* $\texttt{CompOrd} = [\mathcal{C}_1, \ldots, \mathcal{C}_N]$, where N is the number of components. `CompOrd` specifies the order in which contracts are composed to compute the system specification. The composition up to the k^{th} contract is denoted $\mathcal{C}_{\text{comp},k} \coloneqq \mathcal{C}_1 \parallel \ldots \parallel \mathcal{C}_k$, where the composition operator $\parallel$ binds left-associatively.

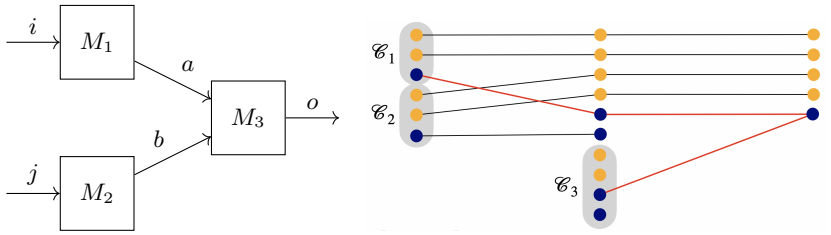
This composition order requires composing the system starting from a single component and building the system up from there. If it is desired to start by composing certain regions of the system first, this framework can easily be extended to include this approach. Otherwise, the component contracts in the composition order need to be provided at the level of granularity such that they can be composed according to a composition order defined in Definition 5. We will now define the diagnostics graph that corresponds to the composition of multiple components and outline how it is constructed. The union of two graphs G_1 and G_2 is defined as $G = (V_1 \cup V_2, E_1 \cup E_2)$, and we will denote it by $G = G_1 \cup G_2$. Simply stated, the diagnostics graph consists of multiple composition graphs. The system is composed step-by-step according to the composition order, and for every composition, the diagnostics graph is extended by the composition graph for this composition. An example is shown in Figure 2b, where we can see the union of two composition graphs. Each 'column' of vertices corresponds to the individual contract terms in a composition, where the top nodes correspond to the already composed system (or the first component for the first composition), and the bottom vertices correspond to the next contract in the composition order.

Definition 6. (Diagnostics Graph) Given component contracts in a composition order $\texttt{CompOrd} = [\mathcal{C}_1, \ldots, \mathcal{C}_N]$, the *diagnostics graph* $G = (V, E)$ is constructed as follows. For each i, $2 \leq i \leq N$, we compute the composition $\mathcal{C}_{\text{comp,i-1}} \parallel \mathcal{C}_i$ and the corresponding graph G_i. Then the diagnostics graph G is defined as $G = \bigcup_{i=2}^{N} G_i$.

Definition 7. (Diagnostics Map) Let `CompOrd` be the composition order for N components and their contracts. Let $\mathcal{C} = \|_{i=1}^{N} \mathcal{C}_i = (\mathfrak{a}, \mathfrak{g})$ be the system-level composition according to the composition order, and let the corresponding diagnostics graph be G. Then, we can define the *diagnostics map* $\texttt{CM} : \mathfrak{g} \to 2^{(\bigcup_{i=1}^{N}(\mathfrak{a}_i \cup \mathfrak{g}_i)) \times \{\mathcal{C}_i\}_{i=1}^{N}}$, that maps each composed assumption and guarantee term to a set of component level assumption or guarantee terms through the diagnostics graph. That is, for system-level term s and the corresponding vertex $s \in V$, we have

$$\begin{aligned} \texttt{CM}(s) = \{(t, \mathcal{C}_i) \mid & \forall \mathcal{C}_i \in \texttt{CompOrd},\ t \in \mathfrak{g}_i \cup \mathfrak{a}_i,\ \text{if} \\ & \exists \texttt{path}(G, t, s) \text{ and } \forall u \in V, u \neq t \implies \nexists (u, t) \in E\}, \end{aligned} \tag{3}$$

where $t \in V$ corresponds to a component-level term used to generate s, and i is the index of the component the term belongs to.



(a) Block diagram. (b) Diagnostics graph.

Fig. 2: Block diagram and diagnostics graph for composition in Example 2.

The diagnostics map finds the leaf nodes in the diagnostics graph that have a path to the vertex corresponding to the violated guarantee. For each leaf node, it returns the term corresponding to the leaf node and the contract that this term belongs to in the form of a tuple. With this information, we can now focus our attention on these terms first and start the diagnostics process by evaluating these terms using the available test data.

Using the diagnostics graph and the system block diagram, we can identify the component assumptions and guarantees that were relevant to the generation of the top-level system guarantee that was violated. We can then focus on the relevant components of the system block diagram and analyze whether the guarantee was satisfied or violated. Once we find a component where the guarantee was violated we can shift our focus to the assumptions of this term. From then on two different scenarios can occur: i) the assumptions are satisfied, or ii) the assumptions are violated.

Case i) When a contract's assumptions are satisfied, but the component does not deliver the contract's guarantees, it either means that the component failed or that the contract did not adequately characterize the context of operation of the component. The analysis at this level terminates, and the component designer needs to analyze the behavior of this component.

Case ii) In the case of violated assumptions, the component is likely not the cause of the failure, as another component's behavior resulted in the violation of the assumptions. To diagnose this fault, we need to trace the violated assumption back to the guarantees of one or more components. We do this by searching over the composition order for the instance of a composition of the component whose assumptions were violated. As this component's contract was composed with another contract, we identify the guarantees of this second contract that are relevant for the satisfaction of the violated assumption. The function ElimVarsbyRefinement in Pacti identifies a subset of terms that participate in the satisfaction of formulas—see [18] for details about this function. After we identify the assumptions and guarantees of the second contract, we refer to the diagnostics graph again to trace these newly identified guarantees back.

It is important to note that this process of tracking assumptions only applies to components whose assumptions are not solely dependent on the overall system

input variables. A system-level failure is defined as having satisfied the system-level assumptions, but failing to satisfy the guarantees of the system. Thus, the system-level assumptions are satisfied—this will ensure that component-level assumptions that are only dependent on the system-level input variables are also satisfied.

Identifying Causes for Violated Assumptions. Assume we are given the component M, its corresponding contract $\mathcal{C} = (I, O, \mathfrak{a}, \mathfrak{g})$, the component M_{other} in the composition order that is composed with component M, and the contract $\mathcal{C}_{\text{other}} = (I_{\text{other}}, O_{\text{other}}, \mathfrak{a}_{\text{other}}, \mathfrak{g}_{\text{other}})$ corresponding to M_{other}. The component assumption that was violated is denoted $a_v \in \mathfrak{a}$. We will use Pacti to find the relevant context used to refine this assumption, referred to as the function FindCauseForAssumption$(a_v, \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}})$. This function exploits the function call ElimVarsbyRefinement in Pacti, which transforms the assumption a_v with the use of $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$ as the context to eliminate any unwanted variables. We can make use of the same function augmentation that we created to compute the diagnostics graph to analyze the transformation at this level. The instrumentation of the filtering step will return the relevant context terms $\mathfrak{c}_r \subseteq \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$ in the assumptions and guarantees of $\mathcal{C}_{\text{other}}$. Once we have determined $\mathfrak{c}_r$, the diagnostics map allows us to trace back the terms in $\texttt{CM}(c)$ for each $c \in \mathfrak{c}_r$ to the responsible component level terms. The entire diagnostics procedure is outlined in Algorithm 1.

This approach can identify multiple component faults under certain conditions. As discussed above, we can only find the faulty component if a system-level guarantee is violated. If two component faults end up cancelling each other out (i.e., are not observable at the system level), then this approach cannot identify them as no system-level guarantee was violated. If a faulty component results in violated assumptions for another component, we cannot determine whether the component with the violated assumption also failed. This is due to the fact that for a contract with violated assumptions, any behavior is allowed. Under the condition that all faulty components are independent (i.e., a faulty component does not lead to violated assumptions of another faulty component), this procedure is able to identify all faulty components.

Example 2. Let there be a system consisting of three component contracts in the composition order $\texttt{CompOrd} = [\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3]$ with their inputs and outputs as illustrated in Figure 2a. The IO contracts are given as $\mathcal{C}_1 = (\{i\}, \{a\}, \mathfrak{a}_1, \mathfrak{g}_1)$, where $\mathfrak{a}_1 = \{i \leq 2, i \geq 0\}$, and $\mathfrak{g}_1 = \{a \leq 2\}$ and $\mathcal{C}_2 = (\{j\}, \{b\}, \mathfrak{a}_2, \mathfrak{g}_2)$, where $\mathfrak{a}_2 = \{j \leq 2, j \geq 0\}$, and $\mathfrak{g}_2 = \{b \leq 3\}$ and $\mathcal{C}_3 = (\{a, b\}, \{o\}, \mathfrak{a}_3, \mathfrak{g}_3)$, where $\mathfrak{a}_3 = \{a \leq 5, b \leq 5\}$, and $\mathfrak{g}_3 = \{o \leq a, o \leq b\}$. The system-level contract is computed as $\mathcal{C} = (\{i, j\}, \{o\}, \mathfrak{a}, \mathfrak{g})$, where $\mathfrak{a} = \{i \leq 2, i \geq 0, j \leq 2, j \geq 0\}$, and $\mathfrak{g} = \{o \leq 2\}$. Suppose the following trace is observed during execution: $i = 1$, $j = 1$, $a = 2$, $b = 7$, $o = 3$. The system-level guarantee $g_v \coloneqq o \leq 2$ is violated while the assumptions are satisfied. The diagnostics map is given by $\texttt{CM}(g_v) = \{(a \leq 2, \mathcal{C}_1), (o \leq a, \mathcal{C}_3)\}$. Applying Algorithm 1, we check the guarantee of component M_1 first and see that $a \leq 2$ is satisfied. Next, we check $o \leq a$, which is

Algorithm 1 Diagnosing Violated Guarantee g_v

procedure DIAGNOSE(g_v, `CompOrd`, `Log`)
  Input: failed guarantee g_v, composition order `CompOrd` $= [\mathcal{C}_1, \ldots, \mathcal{C}_N]$, log data `Log`
  Output: set of failed components C_f
  G $\leftarrow (\emptyset, \emptyset)$ ▷ Initialize empty diagnostics graph
  for $\mathcal{C}_i \in$ `CompOrd` **do**
    $\mathcal{C}_{\text{comp},i-1} \leftarrow \mathcal{C}_1 \parallel \ldots \parallel \mathcal{C}_{i-1}$
    $G_i \leftarrow$ COMPOSITIONGRAPH($\mathcal{C}_{\text{comp},i-1}, \mathcal{C}_i$)
    $G \leftarrow G \cup G_i$ ▷ Add composition graph to diagnostics graph
  `CM` $\leftarrow$ define diagnostics map according to equation (3)
  $C_f \leftarrow$ TRACE(g_v, `CompOrd`, `CM`, `Log`) ▷ Find set of failed components
  return C_f

procedure TRACE(g_v, `CompOrd`, `CM`, `Log`)
  Input: guarantee to trace g_v, composition order `CompOrd` $= [\mathcal{C}_1, \ldots, \mathcal{C}_N]$, diagnostics map `CM`, log data `Log`, components $M_1, \ldots, M_N$
  Output: set of failed components C_f
  $C_f \leftarrow \emptyset$ ▷ Initialize empty set of failed components
  for $(t, \mathcal{C}_i) \in$ `CM`(g_v) **do** ▷ Component-level term t, component index i
    if $t \in \mathfrak{g}_i$ **then** ▷ $\mathfrak{g}_i$ are the guarantees of $\mathcal{C}_i$
      if NOTSATISFIED(t, `Log`) **then** ▷ Check if t is satisfied in log data
        `AssumptionsSatisfied` $\leftarrow$ True ▷ Initialize flag as True
        for $a_i \in \mathfrak{a}_i$ **do**
          if NOTSATISFIED(a_i, `Log`) **then** ▷ Check a_i in log data
            `AssumptionsSatisfied` $\leftarrow$ False
            $\mathcal{C}_{\text{other}} \leftarrow \mathcal{C}_1 \parallel \ldots \parallel \mathcal{C}_{i-1}$
            $\mathfrak{c} \leftarrow$FINDCAUSEFORASSUMPTION(a_i, $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$)
            for $c_k \in \mathfrak{c}$ **do**
              $C_f \leftarrow C_f \cup$ TRACE(c_k, `CompOrd`, `CM`)
        if `AssumptionsSatisfied` **then**
          $C_f \leftarrow C_f \cup M_i$ ▷ Add component i to the list
  return C_f

procedure FINDCAUSEFORASSUMPTION(a_i, $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$)
  Input: assumption to trace a_i, context terms $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$
  Output: list of relevant context terms $\mathfrak{c}$
  _, $\mathfrak{c} \leftarrow$ ELIMVARSBYREFINEMENT($a_i, \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$) ▷ Refine the assumption a_i using the context terms $\mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}}$ and return the relevant context terms
  return $\mathfrak{c}$

not satisfied, as $3 \not\leq 2$. This narrows our analysis on component M_3. Evaluating the assumptions of contract $\mathcal{C}_3$, we find that $a \leq 5$ is satisfied, but $b \leq 5$ is not. Therefore, M_3 is not responsible for the violation. We can now trace which terms were used to transform $b \leq 5$ to find which terms to evaluate next in our search for the failed component. For this, we compute $\mathcal{C}_{\text{other}} = \mathcal{C}_1 \parallel \mathcal{C}_2$ from the composition order and evaluate FindCauseForAssumption$(b \leq 5, \mathfrak{a}_{\text{other}} \cup \mathfrak{g}_{\text{other}})$, which returns the relevant context term as the following guarantee from component contract $\mathcal{C}_2$, $b \leq 3$. This guarantee is not satisfied, as $7 \not\leq 3$. Subsequently, we check the assumptions for $\mathcal{C}_2$, $0 \leq j \leq 2$, which are satisfied, leading to the identification of M_2 as the component responsible for the violation. In this example, only 6 terms were checked instead of all 10, showing that tracing the cause of a violated assumption can require fewer checks than evaluating all component-level terms from the log.

Theorem 1. *Suppose we have a list of components $M_1, \ldots, M_N$, their contracts in a composition order* `CompOrd` $= [\mathcal{C}_1, \ldots \mathcal{C}_N]$*, a violated system-level guarantee g_v, and the complete log data of a failing trace* `Log`*. If g_v is a guarantee of the composed system, we can identify the faulty component(s) using Algorithm 1.*

Proof. Let us denote the composed contract as $\mathcal{C} = (I, O, \mathfrak{a}, \mathfrak{g})$. For a given composition and the corresponding contract $\mathcal{C}$, under satisfied system-level assumptions $\mathfrak{a}$ and a violated system-level guarantee $g_v \in \mathfrak{g}$, by construction of the composition, there exists at least one faulty component. From `CompOrd`, we can construct a diagnostics map `CM`. If $g_v \in \mathfrak{g}$, `CM`(g_v) is guaranteed to contain at least one component-level guarantee $g_k \in \mathfrak{g}_k$, a guarantee of contract $\mathcal{C}_k$, where $1 \leq k \leq N$. For each $g \in$ `CM`(g_v), we evaluate from the trace `Log` whether it is satisfied or violated. If g_k is violated, we have two different cases: i) if the assumptions $\mathfrak{a}_k$ of contract $\mathcal{C}_k$, are satisfied, then M_k is added to the list of responsible components; in case ii), if the assumptions of $\mathcal{C}_k$ are violated, from the composition operation in Pacti, we can identify which component-level terms were used to refine this assumption and identify which terms to evaluate next. By definition of assume-guarantee contracts, an implementation of a contract where the assumptions are satisfied and whose guarantees are violated is faulty. Any component in the analysis that violated its guarantees under satisfied assumptions is faulty.

4 Examples

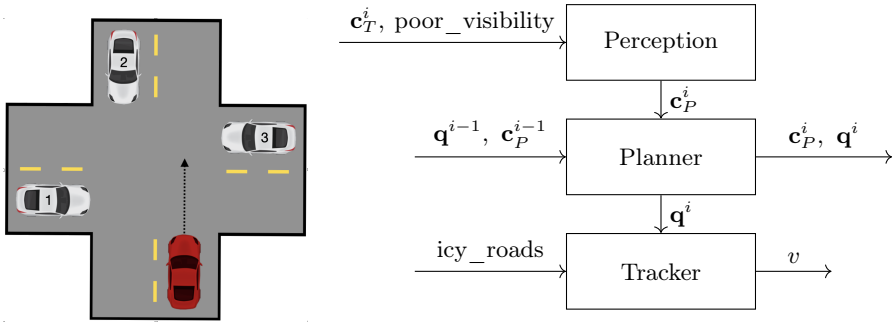
The following example was inspired by Alice, Caltech's entry in the 2007 DARPA Urban Challenge. While conducting the pre-challenge testing campaign, Alice faced scenarios where it failed to accomplish its objective because of an unforeseen behavior arising from the interaction of various subsystems in that particular situation. In this section, we will illustrate a scenario that is loosely based on the real-world scenarios that Alice faced in the DARPA Urban Challenge, which are described in detail in [4].

Alice at Intersection using Propositional Logic In this example, the test was set up such that Alice was approaching an intersection with multiple cars already waiting at the intersection as shown in Figure 3a. While Alice was approaching, its sensors detected the other cars in the intersection and commanded Alice to stop and give way to the other cars. The unforeseen circumstance was that the deceleration tilted the LIDARs forward and towards the ground such that Alice lost sight of the other cars momentarily. Once Alice came to a full stop, the line of sight of the LIDARs tilted back up and detected the cars again, but now Alice was under the impression that the cars just arrived, leading to the control system commanding Alice to drive into the intersection and resulting in unsafe behavior.

We model the components in Alice's control architecture as shown in Figure 3b. Alice's system consists of three components, the *Perception*, the *Planner* and the *Tracker* (highlighted in the red dashed box). For each component in Alice's system, we can define an IO contract that describes the correct component behavior. The perception component is modeled as follows

$$\mathcal{C}_{\text{perception}} = \{I_P, O_P, \mathfrak{a}_{\text{perception}}, \mathfrak{g}_{\text{perception}}\},$$

with the input variables $I_{\text{perception}} = \{c^i_{T_1}, c^i_{T_2}, c^i_{T_3}, \text{poor\_visibility}\}$, the output variables are $O_{\text{perception}} = \{c^i_{P_1}, c^i_{P_2}, c^i_{P_3}\}$, the assumptions $\mathfrak{a}_{\text{perception}} = \{\neg\text{poor\_visibility}\}$, and guarantees $\mathfrak{g}_{\text{perception}} = \{c^i_{T_1} \Leftrightarrow c^i_{P_1}, c^i_{T_2} \Leftrightarrow c^i_{P_2}, c^i_{T_3} \Leftrightarrow c^i_{P_3}\}$. The variables $c^i_{T_1}, c^i_{T_2}, c^i_{T_3}$ correspond to whether there is a car in the 1^{st}, 2^{nd}, and 3^{rd} position in the intersection, and $c^i_{P_1}, c^i_{P_2}, c^i_{P_3}$ corresponds to the perceived state of the cars in the intersection; the variable poor_visibility represents to the visibility conditions. This contract describes that the perception component guarantees that the cars in the intersection will be detected correctly if there is no poor visibility.



(a) Intersection layout.

(b) Diagram of Alice's components and interfaces for timestep i.

Fig. 3: Layout of the intersection and Alice's component block diagram.

The planner component is tasked with determining Alice's spot in the queue of arriving cars to determine whether Alice has the right of way or needs to stop for other cars to take their turn first. For this, the Planner needs to keep track

of the arrival order of the cars at the intersection. The planner inputs are

$$I_{\text{planner}} = \{c^i_{P_1}, c^i_{P_2}, c^i_{P_3}, c^{i-1}_{P_1}, c^{i-1}_{P_2}, c^{i-1}_{P_3}, q^{i-1}_1, q^{i-1}_2, q^{i-1}_3, q^{i-1}_4\},$$

where $c^i_{P_1}, c^i_{P_2}, c^i_{P_3}$ are the detected cars at the intersection at the current timestep i, $c^{i-1}_{P_1}, c^{i-1}_{P_2}, c^{i-1}_{P_3}$ are the detected cars in the previous timestep $i-1$, and $q^{i-1}_1, q^{i-1}_2, q^{i-1}_3, q^{i-1}_4$ corresponds to Alice's position in the queue from the previous timestep. The planner output is given as $O_{\text{Planner}} = \{q^i_1, q^i_2, q^i_3, q^i_4\}$, which corresponds to Alice's updated position in the queue for this timestep. The Planner contract is given as

$$\mathcal{C}_{\text{planner}} = \{I_{\text{planner}}, O_{\text{planner}}, \mathfrak{a}_{\text{planner}}, \mathfrak{g}_{\text{planner}}\},$$

where the assumptions are $\mathfrak{a}_{\text{planner}} = \{(q^{i-1}_1 \wedge \neg q^{i-1}_2 \wedge \neg q^{i-1}_3 \wedge \neg q^{i-1}_4) \vee (\neg q^{i-1}_1 \wedge q^{i-1}_2 \wedge \neg q^{i-1}_3 \wedge \neg q^{i-1}_4) \vee (\neg q^{i-1}_1 \wedge \neg q^{i-1}_2 \wedge q^{i-1}_3 \wedge \neg q^{i-1}_4) \vee (\neg q^{i-1}_1 \wedge \neg q^{i-1}_2 \wedge \neg q^{i-1}_3 \wedge q^{i-1}_4)\}$, which ensures that Alice can only be in one position in the queue in the previous timestep. The guarantees $\mathfrak{g}_{\text{planner}}$ describe how Alice's updated position in the queue will be determined. If none of the other cars leave the intersection, Alice will stay in the same position in the queue,

$$\begin{aligned}
\mathfrak{g}_0 = \{&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_4 \Rightarrow q^i_4,\\
&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_3 \Rightarrow q^i_3,\\
&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_2 \Rightarrow q^i_2,\\
&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_1 \Rightarrow q^i_1\}.
\end{aligned}$$

In the case of a single car leaving the intersection, Alice would advance by one in the queue, described as follows

$$\begin{aligned}
\mathfrak{g}_1 = \{&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q^{i-1}_4 \Rightarrow q_3,\\
&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q^{i-1}_3 \Rightarrow q_2,\\
&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q^{i-1}_2 \Rightarrow q_1,\\
&(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q^{i-1}_4 \Rightarrow q_3,\\
&(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q^{i-1}_3 \Rightarrow q_2,\\
&(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q^{i-1}_2 \Rightarrow q_1,\\
&(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_4 \Rightarrow q_3,\\
&(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_3 \Rightarrow q_2,\\
&(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q^{i-1}_2 \Rightarrow q_1\}.
\end{aligned}$$

If any two cars leave the intersection, Alice would advance in the queue by two steps, given by the following

$$\begin{aligned}\mathfrak{g}_2 =\{&(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q_4^{i-1} \Rightarrow q_3,\\ &(c^i_{P_1} \Leftrightarrow c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q_3^{i-1} \Rightarrow q_1,\\ &(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q_4^{i-1} \Rightarrow q_3,\\ &(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (c^i_{P_2} \Leftrightarrow c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q_3^{i-1} \Rightarrow q_1,\\ &(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q_4^{i-1} \Rightarrow q_3,\\ &(\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (c^i_{P_3} \Leftrightarrow c^{i-1}_{P_3}) \wedge q_3^{i-1} \Rightarrow q_1\}.\end{aligned}$$

If all three cars left the intersection in this timestep, Alice would move from 4^{th} position to the 1^{st} position in the queue, this is given by

$$\mathfrak{g}_3 = \{((\neg c^i_{P_1} \wedge c^{i-1}_{P_1}) \wedge (\neg c^i_{P_2} \wedge c^{i-1}_{P_2}) \wedge (\neg c^i_{P_3} \wedge c^{i-1}_{P_3}) \wedge q_4^{i-1} \Rightarrow q_1)\}.$$

Additionally, if Alice is in the first spot, it will stay in this spot until it takes its turn, $\mathfrak{g}_4 = \{q_1^{i-1} \Rightarrow q_1\}$.The Planner guarantees are thus given as

$$\mathfrak{g}_{\text{planner}} = \mathfrak{g}_0 \cup \mathfrak{g}_1 \cup \mathfrak{g}_2 \cup \mathfrak{g}_3 \cup \mathfrak{g}_4.$$

The Tracker component ensures that Alice will only move into the intersection if it has the right of way. The contract is given as

$$\mathcal{C}_{\text{tracker}} = \{\{q_1^i, \text{icy\_roads}\}, \{v^i\}, \mathfrak{a}_{\text{tracker}}, \mathfrak{g}_{\text{tracker}}\},$$

with the input variables whether it is Alice's turn (q_1^i provided by the Planner, and the output variable being Alice's speed v^i (where v evaluating to True corresponds to a positive speed). The assumptions are $\mathfrak{a}_{Tracker} = \{\neg\text{icy\_roads}\}$, and the guarantees are $\mathfrak{g} = \{q_1^i \Leftrightarrow v^i\}$. This ensures that only when q_1^i is True, Alice will move into the intersection with a positive speed. In addition to the above explained component assumptions and guarantees, the three components also contain additional assumptions and guarantees that are unrelated to the process of determining whether it is Alice's turn. These other guarantees represent different viewpoints of the components, and we will show that our framework accurately pinpoints the potential causes within the relevant terms outlined above. For each component, we added 100 input and output variables, and 100 component guarantees each, which result in 100 system observations per timestep (tracker output variables), as shown in gray in Figure 4. These variables and their interactions correspond to other component functionalities of Alice's system.
Next, we compose Alice's system for one timestep by composing the contracts in their provided composition order $\texttt{CompOrd} = [\mathcal{C}_{\text{perception}}, \mathcal{C}_{\text{planner}}, \mathcal{C}_{\text{tracker}}]$. We can then compose the system for a sequence of time steps, which allows us to evaluate the system behavior using the system-level variables. This composition is shown in Figure 4. The resulting system contract comprises 212 guarantee terms, each involving a logical statement with some being lengthy and complex. We are given the test trace as valuations of the variables for a series of time

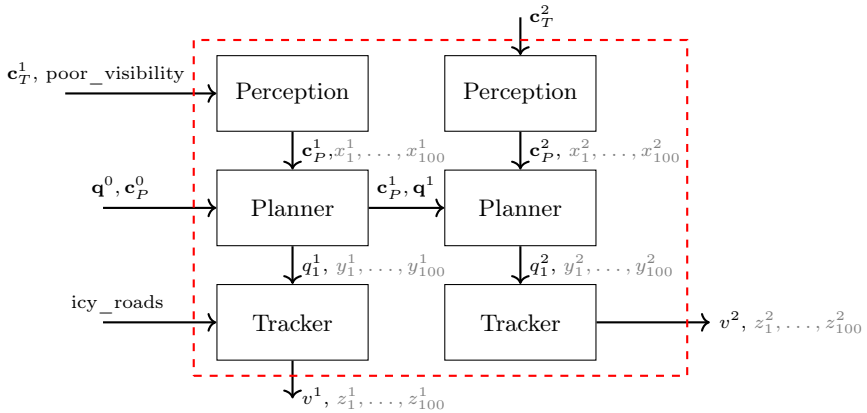


Fig. 4: Two-step execution of a perception–planner–tracker pipeline with intermediate signals. The bold variables denote vectors of the indicator variables.

steps shown in Table 1. The trace contains the observations of the other cars in the intersection, and the visibility and road conditions. Alice's system-level output is its speed v^i and the additional tracker output variables (not shown in the table). The internal variables are not accessible from the system level, except for the initial timestep as we require an initial condition as input for our contracts. We can now analyze the given trace and we notice that five of

Table 1: System-level violating trace (relevant variables)

Time step i	poor_visibility	icy_roads	$c_{T_1}^i$	$c_{T_2}^i$	$c_{T_3}^i$	$c_{P_1}^i$	$c_{P_2}^i$	$c_{P_3}^i$	q_1^i	q_2^i	q_3^i	q_4^i	v^i
0	0	0	1	1	1	1	1	1	0	0	0	1	0
1	0	0	1	1	1	–	–	–	–	–	–	–	1
2	0	0	1	1	1	–	–	–	–	–	–	–	1

the 212 system-level guarantees are violated. We use the diagnostics process to track these guarantees to the possible component causes, which requires us to check 50 component-level statements (out of 656 total). Using this framework, we only have to check 7.6% of the component level terms. This now guides the the debugging process to access the relevant internal variables to evaluate the component-level statements and we will identify that the perception component did not provide its guarantees and did not detect the cars in time step 1. For implementation details, please refer to the notebook in this repository[3].

5 Conclusion and Future Work

In this paper, we proposed a diagnostics methodology based on assume-guarantee contracts using Pacti, a tool for compositional system analysis and design. We

[3] `https://github.com/pacti-org/cs-selfdriving-diagnostics`

formally characterized when a system-level guarantee failure can be traced to a component, defined the required structure for contract composition, and identified the information that must be stored to enable diagnostics.

The diagnostics framework involves composing the system, generating a diagnostics map, and systematically checking component-level guarantees and assumptions using log data. We show that if the log contains valuations of the full system state, the method reliably identifies the responsible component. By focusing on the most relevant contract terms first, the approach often reduces the number of evaluated statements. In the worst case, all component-level contracts must be checked—but the framework prioritizes likely culprits and can significantly reduce effort in practice.

We demonstrated the methodology on abstract examples and two simplified scenarios inspired by real-world autonomous system tests. As future work, we plan to explore connections to behavior explainability in robotics. While our current method uses the contract composition operator, a similar approach could be developed using the contract quotient to identify missing components and trace the origin of contract terms, offering additional insight into system behavior.

Acknowledgments. This work was funded by the Air Force Office of Scientific Research (grant number FA9550-22-1-0333).

References

1. Beard, R.V.: Failure accomodation in linear systems through self-reorganization. Ph.D. thesis, Massachusetts Institute of Technology (1971)
2. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures. pp. 200–225. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92188-2_9, https://doi.org/10.1007/978-3-540-92188-2_9
3. Boteanu, A., Arkin, J., Patki, S., Howard, T., Kress-Gazit, H.: Robot-initiated specification repair through grounded language interaction. arXiv preprint arXiv:1710.01417 (2017)
4. Burdick, J.W., DuToit, N., Howard, A., Looman, C., Ma, J., Murray, R.M., Wongpiromsarn, T.: Sensing, navigation and reasoning technologies for the DARPA urban challenge. DARPA Urban Challenge Final Report, Tech. Rep (2007)
5. Chen, J., Patton, R.J.: Robust model-based fault diagnosis for dynamic systems, vol. 3. Springer Science & Business Media (2012)
6. Chi, Y., Dong, Y., Wang, Z.J., Yu, F.R., Leung, V.C.: Knowledge-based fault diagnosis in industrial internet of things: a survey. IEEE Internet of Things Journal **9**(15), 12886–12900 (2022)
7. Cimatti, A., Grosen, T.M., Larsen, K.G., Tonetta, S., Zimmermann, M.: Exploiting assumptions for effective monitoring of real-time properties under partial observability. In: Madeira, A., Knapp, A. (eds.) Software Engineering and Formal Methods. pp. 70–88. Springer Nature Switzerland, Cham (2025)

8. Cimatti, A., Tian, C., Tonetta, S.: Assumption-based runtime verification with partial observability and resets. In: Finkbeiner, B., Mariani, L. (eds.) Runtime Verification. pp. 165–184. Springer International Publishing, Cham (2019)
9. Cobleigh, J.M., Giannakopoulou, D., PĂsĂreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 331–346. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
10. De Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial intelligence **32**(1), 97–130 (1987)
11. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8), 453–457 (1975)
12. Gao, Z., Cecati, C., Ding, S.X.: A survey of fault diagnosis and fault-tolerant techniques—part i: Fault diagnosis with model-based and signal-based approaches. IEEE transactions on industrial electronics **62**(6), 3757–3767 (2015)
13. Goyal, S., Griggio, A., Tonetta, S.: Leveraging contracts for failure monitoring and identification in automated driving systems. In: Madeira, A., Knapp, A. (eds.) Software Engineering and Formal Methods. pp. 441–460. Springer Nature Switzerland, Cham (2025)
14. Graebener, J.B.M.: Formal Methods for Test and Evaluation: Reasoning over Tests, Automated Test Synthesis, and System Diagnostics. Ph.D. thesis, California Institute of Technology (2024)
15. Henzinger, T.A., Saraç, N.E.: Monitorability under assumptions. In: Deshmukh, J., Ničković, D. (eds.) Runtime Verification. pp. 3–18. Springer International Publishing, Cham (2020)
16. Incer, I.: The Algebra of Contracts. Ph.D. thesis, EECS Department, University of California, Berkeley (2022)
17. Incer, I.: An adjunction between Boolean algebras and a subcategory of stone algebras. Theory Appl. Categ. **41**, Paper No. 57, 2041–2057 (2024)
18. Incer, I., Badithela, A., Graebener, J.B., Mallozzi, P., Pandey, A., Rouquette, N., Yu, S.J., Benveniste, A., Caillaud, B., Murray, R.M., et al.: Pacti: Assume-guarantee contracts for efficient compositional analysis and design. ACM Transactions on Cyber-Physical Systems **9**(1), 1–35 (2025)
19. Lamport, L.: win and sin: Predicate transformers for concurrency. ACM Transactions on Programming Languages and Systems (TOPLAS) **12**(3), 396–428 (1990)
20. Mallozzi, P., Incer, I., Nuzzo, P., Sangiovanni-Vincentelli, A.: Contract-based specification refinement and repair for mission planning. In: 2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 29–38. IEEE (2023)
21. Meyer, B.: Applying 'design by contract'. Computer **25**(10), 40–51 (1992)
22. Niemann, H.: A setup for active fault diagnosis. IEEE Transactions on Automatic Control **51**(9), 1572–1578 (2006)
23. Nuzzo, P., Sangiovanni-Vincentelli, A.L., Bresolin, D., Geretti, L., Villa, T.: A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. Proceedings of the IEEE **103**(11), 2104–2132 (2015)
24. Raman, V., Kress-Gazit, H.: Explaining impossible high-level robot behaviors. IEEE Transactions on Robotics **29**(1), 94–104 (2012)
25. Reiter, R.: A theory of diagnosis from first principles. Artificial intelligence **32**(1), 57–95 (1987)
26. Sangiovanni-Vincentelli, A.L., Damm, W., Passerone, R.: Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. Eur. J. Control **18**(3),

217–238 (2012). https://doi.org/10.3166/ejc.18.217-238, https://doi.org/10.3166/ejc.18.217-238
27. van Schrick, D.: Remarks on terminology in the field of supervision, fault detection and diagnosis. IFAC Proceedings Volumes **30**(18), 959–964 (1997)
28. Varanasi, S.C., Meng, B., Lorch, R., Moitra, A., Siu, K., Paul, S., Durling, M., Beniwal, N., Visnevski, N.: Trace: Toolkit for requirements analysis, capture, and elicitation. In: Dutle, A., Humphrey, L., Titolo, L. (eds.) NASA Formal Methods. pp. 380–399. Springer Nature Switzerland, Cham (2025)