A Behavioral Notion of Robustness for Software Systems

Changjian Zhang
Institute for Software Research
School of Computer Science
Carnegie Mellon University
changjiz@andrew.cmu.edu

David Garlan
Institute for Software Research
School of Computer Science
Carnegie Mellon University
garlan@cs.cmu.edu

Eunsuk Kang
Institute for Software Research
School of Computer Science
Carnegie Mellon University
eskang@cmu.edu

ABSTRACT

Software systems are designed and implemented with assumptions about the environment. However, once the system is deployed, the actual environment may deviate from its expected behavior, possibly undermining desired properties of the system. To enable systematic design of systems that are robust against potential environmental deviations, we propose a rigorous notion of robustness for software systems. In particular, the robustness of a system is defined as the largest set of deviating environmental behaviors under which the system is capable of guaranteeing a desired property. We describe a new set of design analysis problems based on our notion of robustness, and a technique for automatically computing robustness of a system given its behavior description. We demonstrate potential applications of our robustness notion on two case studies involving network protocols and safety-critical interfaces.

ACM Reference Format:
Changjian Zhang, David Garlan, and Eunsuk Kang. 2020. A Behavioral Notion of Robustness for Software Systems. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software systems are designed, implemented, and validated with certain assumptions about the environment in which they are deployed. These assumptions include, for example, the expected behavior of a human user, the reliability of the underlying communication network, or the capability of an attacker that may attempt to compromise the security of the system.

Once the system is deployed, however, the actual environment may deviate from its expected behavior, either deliberately or erroneously due to a change in the operating conditions or a fault in one of its parts. For instance, a user interacting with a computer interface may inadvertently perform a sequence of actions in an incorrect order; a network may experience a disruption and fail to deliver a message in time; or an attacker may evolve over time and obtain a wider range of exploits to compromise the system. In these cases, the system may no longer be able to satisfy those requirements that relied on the original assumptions.

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In well-established engineering disciplines such as aerospace, civil, and manufacturing, deviations of the environment from the norm are routinely and explicitly analyzed, and systems are designed to be *robust* against these potential deviations [32]. In software engineering, however, a standard notion of robustness seems to be missing, although a similar concept has been studied in certain domains. For example, in distributed systems and networks, the notion of fault tolerance has been long studied (e.g., [15, 27]), but does not generalize to other types of software systems where environmental deviations are not limited to network failures or delays. In control engineering, a system is said to be robust if small deviations on an input result only in small deviations on an output [40]. This notion of robustness, however, is intended for systems whose behaviors are modeled using continuous dynamics, and not particularly suitable for discrete behaviors observed in software.

In this paper, we propose an approach for designing robust systems based on a mathematically rigorous notion of robustness for software. In particular, we say that a system is *robust* with respect to a *property* and a particular set of *environmental deviations* if the system continues to satisfy the property even if the environment exhibits those deviations. Furthermore, we define the *robustness* of a software system as the set of all deviations under which a system continues to satisfy that property. Based on these definitions, we propose an analysis technique for automatically computing the robustness of a system given its behavioral description.

We argue that robustness itself is a type of software quality that can be rigorously analyzed and designed for. The goal of a typical verification method is to check the following: Given system M, environment E, and property P, does the system satisfy the property under this environment (i.e., $M \| E \models P$)? Our notion of robustness enables formulation of new types of analyses beyond this. For instance, we could ask whether a system is robust against a particular set of environmental deviations; given two alternative system designs (both satisfying P), we could rigorously compare them by generating deviations against which one design is robust but the other is not, and; given multiple system properties (some of them more critical than others), we could compare the environmental deviations under which the system can guarantee them.

We envision that our notion of robustness can be used to support design activities across various domains. In this paper, we demonstrate the application of our approach in two different domains: (1) human-machine interactions, where we adopt the well-studied models of human errors from the industrial engineering and human factors research [6, 35] and show how our method can be used to rigorously evaluate the robustness of safety-critical interfaces against such errors, and (2) computer networks, where our method is used to rigorously compare the robustness of network protocols against different types of failures in the underlying network.

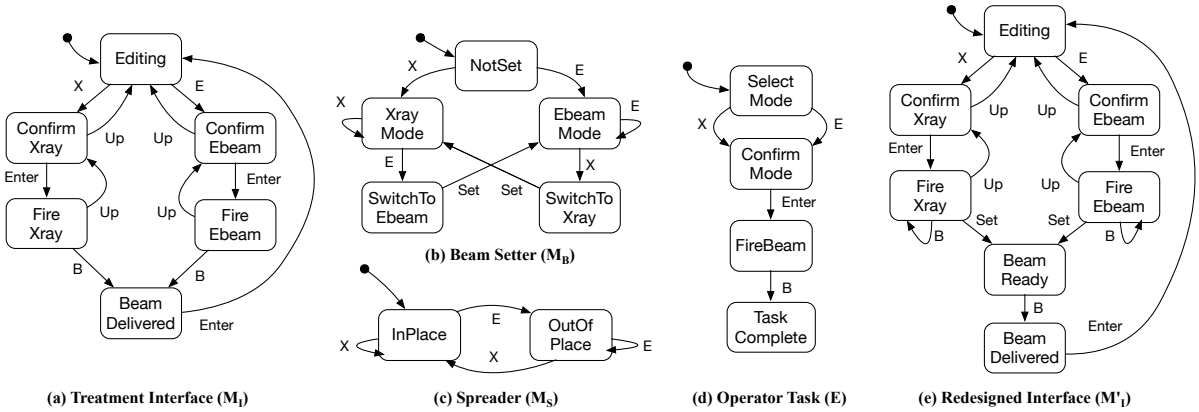

Figure 1: Labelled transition systems for a radiation therapy system.

The contributions of the paper are as follows:

- A systematic approach to designing systems that are robust against potential environmental deviations (Section 2),
- A formal notion of robustness for software systems (Section 3) and a set of analysis problems that evaluate system designs with respect to their robustness (Section 4),
- Algorithmic techniques for automatically computing the robustness of a system and generating succinct representations of robustness (Section 5), and,
- A prototype implementation of the robustness analysis and demonstrate our approach on two case studies involving human-machine interfaces and network protocols (Section 7).

2 MOTIVATING EXAMPLE

This section illustrates how our proposed notion of robustness may be used to support a new type of design analysis and aid a systematic development of systems that are robust against failures or changes in the environment.

(1) Analysis under the normative environment. As a motivating example, consider the design of a radiation therapy system similar to the well-known Therac-25 machine [29]. State machines in Figure 1 describe three components in the system, including the *treatment interface* (M), which allows the operator to control the device by performing interface actions (e.g., X for setting the beam mode to X-ray), the *beam setter* (M), which determines the current mode of radiation therapy (electron beam and X-ray, which delivers roughly 100 times higher level of current than the former), and *spreader* ($M_{(}$), which is put in place during the X-ray mode in order to attenuate the effect of the high-power X-ray beam and limit possible overdose. The overall behavior of the therapy system, as modeled here, is captured by the composition of the state machines, $M = (M \,\|M\, \|M_{(})$.

The radiation therapy system is associated with a number of safety requirements, one of which states that the spreader can be removed only when the beam is delivered in the electron mode. This requirement may formally be stated as the following property in linear-temporal logic (LTL) [33]:

$$\mathbf{G}(\mathsf{BeamDelivered} \wedge \mathsf{OutOfPlace} \Rightarrow \mathsf{EbeamMode})$$

where BeamDelivered is a proposition that holds when M enters the state with the same name (and similarly for other propositions).

During a normal treatment process, a therapist is expected to perform the following tasks: Select the correct therapy mode for the current patient by pressing either X or E, confirm the treatment data by pressing Enter and then finally initiate the beam delivery to the patient by pressing B. This normative behavior of the operator is modeled as state machine E in Figure 1.

Suppose the designer of the machine wishes to check whether the therapy system satisfies its safety requirements, assuming that an operator carries out the tasks as expected. More generally, this can be formulated as the following common type of analysis task:

> Does the system, under the environment that behaves as expected, satisfy a desired property?

To perform this task, one may apply a verification technique such as model checking [12] to check whether the composition of the machine and the environment satisfies a desired property (however, other analysis techniques may be just applicable as long as they can be used to check $M\|E \models P$). Performing this analysis confirms that the system indeed satisfies the safety property that the spreader is always in-place during the X-ray mode.

(2) Analysis of undesirable environmental deviations. In complex systems, the environment may not always behave as expected, and possibly undermine assumptions that the system relies on to fulfill its requirements. For instance, in interactive systems, human operators are far from perfect, and inadvertently make mistakes from time to time while performing a task (e.g., perform a sequence of actions in a wrong order) [35]. In the context of a safety-critical system such as medical devices, some of these operator errors, if permitted by the interface, may result in a safety violation.

To discover these potential environmental deviations, the designer decides to perform the following analysis task:

> What are possible ways in which the environment may deviate from its expected behavior and cause a violation of the property?

Given the therapy system models (M and E) and property P, the designer can use an existing analysis tool (e.g., LTSA [30]) to check whether $M \models P$. The analyzer may return a counterexample trace that demonstrates how the operator could deviate from its normative behavior (as captured by E) and cause a violation of P.

Suppose that one such trace contains the following sequence of operator actions⟨X-Ray, Up, E-Beam, Up⟩. This trace depicts a scenario in which the operator accidentally selects the X-ray mode, corrects the mistake by pressing up and selecting the electron beam mode, and then carrying on the rest of the treatment as intended (by con- rming the mode and ring the beam). This sequence of operator actions, however, may lead to a violation of the safety property %in the following way: When the operator presses Beam, the beam setter may still be in the process of mode switch (i.e., state SwitchToBeam), causing the beam to be delivered in the X-ray mode while the spreader is out of place. This scenario corresponds to one type of failure that caused fatal overdoses in the Therac-25 system [26].

(3) Robustness analysis. Having discovered how the operator's mistake could lead to a safety violation, the designer modi es the treatment interface to improve its robustness against the possible error. In this redesign, shown in Figure 1(e), the operator can press Beam to re the beam only after the mode switch has been carried out by the beam setter. As the next step, the designer wishes to ensure that the system, as re-designed, is robust against the operator's mistake (i.e, it continues to satisfy the safety property even under the misbehaving operator).

The designer could check "′ ⊨ %where "′ is the redesign, if no errors returned, it means that "′ is robust against the mistake and also "′ can work under any environment. However, it's not always the case. More likely, the analyzer may return another trace representing a new mistake, and it does not necessarily mean that the system is robust against the old one.

Instead, the designer can use our tool to perform the following robustness analysis task:

> What are possible environmental deviations under which the new design satis es the property but the old design does not?

Given the original system model "I , modi ed system model "′, normative environment , and property %, our analysis returns a set of traces (expressed over environmental actions), each trace describes a scenario where system "′ satis es the property but " does not. For example, one of the traces is the sequence of operator actions discussed above⟨X-Ray, Up, E-Beam, Up⟩, con rming that the redesign has correctly addressed the risk of a possible safety violation due to this particular type of mistake by the operator.

The analysis steps (2) and (3) may be repeated to identify potential safety violations due to other types of operator mistakes and further improve the robustness of the system.

3 ROBUSTNESS NOTION

This section describes the underlying formalism used to model systems and environments (namely, labelled transition systems). We then formally de ne the notion of robustness and introduce a new set of analysis problems that leverage this notion to reason about the robustness of systems.

3.1 Preliminaries

In this work, we use labelled transition systems to model the behaviors of machines and environment.

3.1.1 Labelled Transition System.

A labelled transition system is a tuple ⟨(, Σ, ′, 0⟩ where (is a set of states, Σ is a set of actions called the alphabet of , ′ ⊆ (× (Σ ∪ {τ}) × (de nes the state transitions (where τ is a designated action that is unobservable to the system's environment), and 0 ∈ (is the initial state. An LTS is non-deterministic if ∃0 ∈ (. ∃ ∈ Σ ∪ {τ}. ⟨0, , 0′⟩ ∈ ′ ∧ ⟨0, , 0′′⟩ ∈ ′ ∧ 0′ ≠ 0′′; otherwise, it is deterministic.

A trace of ∈ (Σ ∪ {τ}) of an LTS is a sequence of observable actions from the initial state. Then, the behavior of is the set of all the traces generated by , denoted ℒ( ) = { ∈ Σ ∗ | ⟨0, , 0′⟩ ∈ ′}.

3.1.2 Operators.

For an LTS = ⟨(, Σ, ′, 0⟩, the projection operator is used to expose only some subset of the alphabet of . Given = ⟨(, Σ, ′, 0⟩ where for any ⟨0, , 0′⟩ ∈ ′, if ∉ , then ⟨0, , 0′⟩ ∈ ′, i.e., will be hidden by ; otherwise, ⟨0, , 0′⟩ ∈ ′.

The operator can also be applied to traces. We use to denote the trace that results from removing all the occurrences of actions ∉ from .

The parallel composition || is a commutative and associative operator which combines two LTSs by synchronizing their common actions and interleaving the remaining actions. Let = ⟨(1, Σ1, ′1, 01⟩ and) 2 = ⟨(2, Σ2, ′2, 02⟩, 1 || 2 is an LTS = ⟨(, Σ, ′, 0⟩ where (= (1 × (2, Σ = Σ1 ∪ Σ2, 0 = ⟨01, 02⟩, and ′ is de ned as: For any ⟨0, , 0′⟩ ∈ ′1 and ∉ Σ2, we have ⟨⟨0, 2⟩, , ⟨0′, 2⟩⟩ ∈ ′; for any ⟨0, , 0′⟩ ∈ ′2 and ∉ Σ1, we have ⟨⟨1, 0⟩, , ⟨1, 0′⟩⟩ ∈ ′; and for ⟨0, , 0′⟩ ∈ ′1 and ⟨0, , 0′⟩ ∈ ′2, we have ⟨⟨0, 0⟩, , ⟨0′, 0′⟩⟩ ∈ ′.

3.1.3 Properties.

In this work, we consider a class of properties called safety properties [26]. In particular, a safety property % can be represented as a deterministic LTS that contains sink transitions. It de nes the acceptable behaviors of a system over Σ% and we say that an LTS satis es % (denoted ⊨ %) if and only if ℒ( ) ∩ Σ% ⊆ ℒ(%).

We check whether an LTS satis es a safety property % = ⟨(%, Σ%, ′%, 0%⟩ by automatically deriving an error LTS ⊗% = ⟨(⊗, Σ⊗, ′⊗, 0⊗⟩ where (⊗ = {0 × 0% | 0 ∈ (} ∪ {c}, Σ⊗ = Σ ∪ Σ%, and ′⊗ = {⟨⟨0, 0%⟩, , ⟨0′, 0%′⟩⟩ | ⟨0, , 0′⟩ ∈ ′ ∧ ⟨0%, , 0%′⟩ ∈ ′% ∧ ∈ Σ%} ∪ {⟨⟨0, 0%⟩, , c⟩ | ∈ Σ% ∧ ∄0′. ⟨0%, , 0%′⟩ ∈ ′%}. With this ⊗% LTS, we test whether the error state c is reachable in || ⊗%. If c is not reachable, then we can conclude that ⊨ %.

3.2 Robustness De nition

Let " be the LTS of a machine, the LTS of the environment, and Σ" = Σ" ∩ Σ the common actions between the machine and the environment. Then, we say Σ" represents the set of all environmental behaviors that are permitted by machine " .

Machine " is said to be robust against a set of traces ⊆ (Σ ∪ Σ") ∗ if and only if the system satis es a desired property under a new environment (′) that is capable of additional behaviors in compared to the original environment ( ):

De nition 3.1. Machine " is robust against a set of traces with respect to environment and property % if and only if ∩ (Σ ∪ Σ")∗ \ ℒ( ) ∩ Σ" ∗ = ∅, and for every ′ such that ℒ(′) ∩ Σ" ∗ = ℒ( ) ∩ Σ" ∗ ∩ , " || ′ ⊨ %.

The set of traces in are also called deviations of ′ from over Σ" . Then, the robustness of a machine is de ned as the largest set

of environmental deviations under which the system continues to satisfy a desired property:

De nition 3.2. The robustness of machine" with respect to environment and property%, denoted ¹"• •%º, is the set of traces X such that" is robust against X with respect to and %, and there exists no X′ such that X X′ and" is also robust against X′.

Figure 2: Behavioral relationships between possible environments.

Figure 2 illustrates the relationships between the behaviors of possible environments that interact with a machine through shared actions U \ U" . The outermost circle represents the set of all environmental behaviors that are permitted by the machine; the innermost circle represents the normative behaviors of the environment. The deviations of the environment could be classi ed into two sets: those under which the machine still maintains a desired property %(i.e., its robustness), and the others that lead to its violation (the area shaded red in Figure 2).

4 ANALYSIS PROBLEMS

This section de nes a set of analysis problems for evaluating system designs with respect to their robustness.

Problem 4.1 (Robustness analysis). Given machine" , environment , and property %, compute ¹"• •%º.

Given a method for computing the robustness of a machine (described in Section 5), we can also perform the following analyses:

Problem 4.2 (Design comparison). Given machines" 1 and" 2, environment , and property %such that U" 1 \ U = U" 2 \ U , compute set = ¹" 2• •%º ¹" 1• •%º.

This analysis allows us to compare a pair of machines (representing alternative designs of a system) on their robustness against the given environment and property. " 2, for example, may be an evolution of" 1; the result of the analysis would describe precisely the environmental deviations under which" 2 is more robust than " 1. Note that" 1 and" 2 may overlap, not necessarily subsume, in terms of their robustness.

Another type of analysis can be used to reason about how the robustness of a machine changes depending on the property that it attempts to establish:

Problem 4.3 (Property comparison). Given machines" , environment , and properties %1 and %2, compute set = ¹"• •%2º ¹"• •%1º.

For instance, suppose that %1 says that the radiation therapy system should always deliver the correct amount of dose to each patient , while %2 states that the system never overdoses patients by delivering X-ray while the spreader is out of place (similar to property %from Section 2). The result of this analysis could tell us, for example, that the system is capable of guaranteeing %2 (weaker and arguably more critical of the two) even under certain operator errors, while %1 may be violated under similar deviations.

In general, since improving robustness might introduce additional complexity into the system, it may be a cost-e ective strategy to design the system to be robust for most critical of the system requirements [24]; our analysis could be used to support this approach to design.

5 ROBUSTNESS COMPUTATION

This section describes a method for automatically computing the robustness of the machine with respect to a given environment and a desired property (Problem 4.1 in Section 4).

5.1 Overview

Figure 3 shows the overall process of our approach to compute the robustness of a machine with respect to environment and property %. The input of our tool is the LTS of , , and %. We rst generate the weakest assumption of" (Section 5.2) to compute ¹"• •%º. Since may be in nite, we then generate a succinct representation of it. We compute the representative model of (Section 5.3.1), group the traces into equivalence classes, and generate a nite set of representative traces (Section 5.3.2). Finally, we take an external deviation model as input to generate explanations for those representative traces (Section 5.4). The nal output is a set of pairs of a representative trace and its explanation.

5.2 Weakest Assumption

In assume-guarantee style of reasoning [25], a machine is considered capable of establishing a property under some assumption about the behavior of the environment. In our modeling approach, an assumption is represented as some subset of all permitted environmental behaviors; the largest such subset is called the weakest assumption (the second largest circle in Figure 2). More formally:

De nition 5.1. The weakest assumption ,•• % of a machine" with respect to environment and property %is an LTS which de nes the largest subset of the permitted environment behaviors of" which satisfy property %, i.e.,

$$14\ ^1,\ \text{"}\bullet\bullet\%\ ^o\ \ 14\ ^1\text{"}\quad \mathrm{U}\ \text{"}\ ^o\ \wedge\ \text{"}\ jj,\ \text{"}\bullet\bullet\%\ j\text{=}\ \%\wedge$$
$$8\ ^0\text{:"}\ jj\ ^0 j\text{=}\ \%\$\ \ ^0 j\text{=},\ \text{"}\bullet\bullet\%$$

If stated otherwise, we will simply write , to mean , "•• % for the rest of the paper.

Then, the robustness of a machine is equivalent to its weakest assumption minus the behaviors of the original environment. More formally, we can compute the robustness of machine" with respect to environment and property %by constructing the following set:

$$^1\text{"}\bullet\bullet\%\ ^o\text{=}\ \text{ff}\ 2\,14\ ^1,\ ^o j\text{f}\ 8\,14\ ^1\quad \mathrm{U}\ \text{"}\ ^o\text{g}\qquad (1)$$

We use the approach by Giannakopoulou et al. [17] to generate the weakest assumption of a system to satisfy a certain safety

Figure 3: Overview of the process for robustness computation. The input is the LTS of machine " , environment , and property % Section 5.2 describes weakest assumption generation for computing . Section 5.3 describes generating robustness representation (i.e., a set of representative traces). Finally, Section 5.4 describes explanation generation for the representative traces.

property. We brie y describe the approach: Given the system LTS " , the environment LTS , and the safety property %,

(1) Compose system " with the error LTS of property % (as de ned in Section 3.1.3) and project its alphabet to the common actions between " and , i.e., let U " = U" \ U , we compute ¹" jj% 4AA º U " .
(2) Perform backward propagation of the error state over g transitions in the LTS obtained from Step 1. We prune all states that are backward reachable from the error state via one or more g steps. The rationale is that if the system is in a state which can enter the error state with some internal actions, then no environment can prevent the property violation.
(3) Determinize the LTS obtained from step 2 by applying g elimination and subset construction [22].
(4) Remove the error state and all of its incoming transitions to obtain the LTS that corresponds to the weakest assumption.

5.3 Representation of Robustness

In general, the set of environmental traces that represent robustness in Equation (1) may be in nite. Since simply enumerating this set may not be an e ective way to present this information to the system designer, we propose a succinct, nite representation of the robustness. The key idea behind our approach is that many of the traces in ¹"• •%º capture a similar type of deviation (e.g., a human operator erroneously skipping an action) and can be grouped into the same equivalence class with a single representative trace that describes the deviation. Based on this idea, we describe a method for automatically converting into a nite number of such equivalence classes (and thus, a nite set of representative traces).

5.3.1 Representative Model of Robustness.

Recall from Equation (1) that contains traces that are in the weakest assumption but not in the original normative environment . To construct an LTS that represents , we take advantage of the method to check safety properties (described at the end of Section 3.1.3). In particular, we treat the original environment projected over U " as a safety property, and compute traces in that lead to a violation of this property; any such trace represents a pre x of the traces in ¹"• •%º.

To illustrate our approach, consider a simple example in Figure 4, where, is the weakest assumption generated from some machine " and is the original environment. To compute the representation of ¹"• •%º, we rst test whether, ⊨ ¹ U " º, which is

Figure 4: LTS's for a simple example illustrating the construction of robustness.

equivalent to testing whether the error state is reachable in , jj¹ U " º4AA as shown in Figure 4(c). We say , jj¹ U " º4AA is the representative model of ¹"• •%º, and let ¹,• º be the set of all the error traces in it. Then,

$$¹"• •%º = f f 2 14 ¹, º j 9f ^0 2 ¹,• º : ?A45 8G• f º g \quad (2)$$

where ?A45 8G• f₁• f₂º means f₁ is the pre x of f₂. Thus, a trace in ¹,• º can represent a set of traces in ¹"• •%º that share this pre x. For our example, trace h0•2i in ¹,• º can represent, e.g., h0•2•0•1•1i and h0•2•0•2•1i in ¹"• •%º.

5.3.2 Representative Traces of Robustness.

Nevertheless, ¹,• º may also be in nite due to possible cycles. For example, in Figure 4(c), h0•1•1• " " "•0i would result in in nite number of error traces. Therefore, we further divide the traces into equivalence classes:

Let LTS ,• = h(,• •U " •',• •B0i be the composition , k¹ U " º4AA. Then,

$$¹,• º = \bigcup_{B2(,• \atop 02U "} B•0¹,• º \text{ where } ¹B•0•cº 2 ',•$$

i.e., B•0¹,• º denotes a subset of traces in ¹,• º that all end with transition ¹B•0•cº. Then, we have

$$¹"• •%º = f f 2 14 ¹, º j 9 B•0¹,• º•9f ^0 2 B•0¹,• º : ?A45 8G• f º g \quad (3)$$

We say that B•0¹,• º is an equivalence class of ¹,• º. In our example, we have two equivalence classes, 1•2¹,• º and 2•0¹,• º. Traces like h0•2i and h0•1•2•0•2i all belong to class 1•2¹,• º; and traces like h0•1•0i and h0•1•1•1•0i all belong to class 2•0¹,• º.

The rationale is that B is the last state by following the normative behaviors of the original environment, and 0 is the rst deviated

action. Thus, $B(s_0, a)$ describes a class of traces that deviate from the original environment from the same normative state s_0 and by the same action a.

Since S_e and Σ_e are finite, so we have a finite number of equivalence classes. We can simply generate them by enumerating all the transitions leading to the error state. Then, we can pick one of the traces in each equivalence class to represent it. Because we may not be interested in how the environment reaches the last normative state, here we simply choose the shortest one. Finally, we define:

Definition 5.2. The representation of $\mathcal{L}(M_d \| M_n)$, denoted by $\mathcal{R}(\mathcal{L}(M_d \| M_n))$, is a finite set of traces such that each trace in it is the shortest trace of one of the equivalence classes of $B(s, a)$.

Therefore, for our conceptual example, $\mathcal{L}(M_d \| M_n)$ can be represented by: $\langle s_0, a, s_1 \rangle : \langle s_0, s_2 \rangle$, and $\langle s_0, a, s_1 \rangle : \langle s_0, s_1, s_0 \rangle$.

5.4 Explanation of Representative Traces

By definition, a representative trace in $\mathcal{R}(\mathcal{L}(M_d \| M_n))$ contains only actions from Σ_e. While this trace describes how the environment deviates from its expected behavior as observed by the machine, it does not capture how the internal behavior of the environment could have caused this deviation. To provide such explanation for an environmental deviation, we propose a method for augmenting the representative traces with additional domain-specific information (called *faulty events*) about the underlying root cause behind the deviation. In this approach, the normative model is augmented with additional transitions on these faulty events (which are internal to the environment) and an automated method is used to extract a *minimal explanation* for a particular representative trace.

5.4.1 Explanations from a Deviation Model. In order to build explanations for representative traces, our tool takes a deviation model as input, which contains normative and deviated behaviors, and maps each representative trace to a trace in the deviation model.

Definition 5.3. A deviation model of environment is an LTS $M_d = \langle S_d, \Sigma_d, \Delta_d, s_0 \rangle$ where $\Sigma_d = \Sigma_e \cup \{f_1, f_2, \ldots, f_n\}$, f_i is a fault in the environment, $\Delta(s, a) = S_e$, and $\Delta(s, a) = \Sigma_e$, and $\Delta(s, a) = \Sigma_e \setminus \mathcal{L}(M_n)$.

Our tool makes no assumptions on how to generate such a deviation model. It can be built manually (e.g., Section 7.2 uses a manually defined deviation model); or it can be derived from existing fault models in other fields (e.g., Section 7.3 derives the deviation model from an existing human error behavior model). The model may not necessarily cover all the traces in $\mathcal{L}(M_d \| M_n)$; however, we say a deviation model is *complete* with respect to $\mathcal{L}(M_d \| M_n)$ if and only if $\mathcal{L}(M_d \| M_n) = \Delta(s, a) = \Sigma_e$.

Then, an explanation of a representative trace is a trace in the deviation model:

Definition 5.4. For any trace $t \in \mathcal{R}(\mathcal{L}(M_d \| M_n))$ and $t' \in \Delta(s, a)$, if $t' \downarrow \Sigma_e = t$, then we say t' is an explanation of t.

Consider a deviation model for our simple example in Figure 5, then: for the representative trace $\langle s_0, s_2 \rangle$, we can build explanations $\langle s_0, s_2, s_2 \rangle$ and $\langle s_0, s_3, s_4, s_2 \rangle$; and for the representative trace $\langle s_0, s_1, s_0 \rangle$, we can build an explanation $\langle s_0, s_1, s_2, s_0 \rangle$.

Figure 5: Deviation model for the simple example.

5.4.2 The Minimal Explanation. Of course, there could be infinite number of explanations for a representative trace. However, similar to software testing where we are often interested in the smallest test cases against certain errors, here we are also only interested in the explanation of t which contains the minimal number of faults.

Definition 5.5. The *minimal explanation* for $t = \langle s_0, a_1, s_1, \ldots, a_n, s_n \rangle$ in $\mathcal{R}(\mathcal{L}(M_d \| M_n))$ under deviation model is the shortest trace $t' \in \Delta(s, a)$ where $t' \downarrow \Sigma_e = t$ and faulty actions only exist between s_{i-1} and s_i.

A minimal explanation describes: 1) how the environment can reach the last normative state without any faults; 2) and what minimal sequence of faults have caused the environment to deviate from the normative behavior.

To compute the minimal explanation for $t \in \mathcal{R}(\mathcal{L}(M_d \| M_n))$, let $M_t = \langle S_t, \Sigma_d, \Delta_t, s_0 \rangle$ be the LTS where t and its prefixes are the only traces in it. Besides, we make the last action of t lead to s_e to denote the end state, i.e., $\Delta(s_{n-1}, a_n) = s_e$. Then, we use BFS to search the minimal explanation in $\mathcal{L}(M_t)$, as shown in Algorithm 1.

Line 1-3 define an empty queue to store the remaining search states and an empty set to store the visited states, and add the initial state to the queue. The algorithm loops until the queue is empty (Line 4). If the current visiting state is s_e, then it returns the current trace as the explanation (Line7-8); otherwise, it adds the next states to the queue. Specifically, if the current trace does not match the prefix of t, i.e., $\langle s_0, \ldots, a_{i-1}, s_{i-1} \rangle$, then it only adds states with a non-faulty transition (Line 12-13). Since BFS returns on the first result, it is guaranteed to find the minimal explanation. For example, our algorithm returns $\langle s_0, s_2, s_2 \rangle$ as the minimal explanation for $\langle s_0, s_2 \rangle$ instead of $\langle s_0, s_3, s_4, s_2 \rangle$ in the deviation model (Figure 5(b)).

6 ROBUSTNESS COMPARISON

This section describes a method to compare robustness between a pair of machines (Problem 4.2), or a machine against a pair of properties (Problem 4.3). According to Equation (1), to solve Problem 4.2, we have

$$\begin{aligned} \Gamma &= \mathcal{R}(\mathcal{L}(M_{d2} \| M_{n2})) \setminus \mathcal{R}(\mathcal{L}(M_{d1} \| M_{n1})) \\ &= \{t \mid t \in \mathcal{L}(M_{t2}), \Sigma_{e2}) \wedge t \notin \mathcal{L}(M_{t1}), \Sigma_{e1})\} \end{aligned}$$

By assuming $\mathcal{L}(M_{t1}) \cup \Sigma_{e1}) \subseteq \mathcal{L}(M_{t2}), \Sigma_{e2})$, we can simplify the equation to:

$$\Gamma = \mathcal{R}(\mathcal{L}(M_{d2} \| M_{n2})) \setminus \mathcal{R}(\mathcal{L}(M_{d1} \| M_{n1})) = \{t \mid t \in \mathcal{L}(M_{t2}), \Sigma_{e2}) \wedge t \notin \mathcal{L}(M_{t1}), \Sigma_{e1})\}$$

Then, we can use the same method described in Section 5.3 to generate its representation. By computing $\mathcal{L}(M_{t2}) \| \Sigma_{e1})$ we have Σ_{e1}, Σ_{e2}, representing all the prefixes of Γ. Similarly, we divide

Algorithm 1: Minimal explanation search

```
Data: A trace f 2 A4?1"• •%º and the LTS of jj) f
Result: The minimal explanation f 0 2 14 ¹ º
@:= empty queue ;          // remaining search states
E:= empty set of states ;          // visited states
enqueue(@, ¹B0•hiº) ;
while : isEmpty(q) do
    B•C= dequeue(q) ; // B the current state, C the
      current trace
    if B8 E then
        if B= c then
            return t;
        else
            E:= E[ f Bg;
            for ¹B•0•B0 2 ' do
                if C U " = BD1) A02f 0• = 1º then
                  enqueue(@, ¹B0•C§ 0º) ;
                /* C does not match h00•""""•0 1i . */
                else if 0 is not a faultthen
                  enqueue(@, ¹B0•C§ 0º) ;
            end
        end
    end
end
```

it into equivalence classes, i.e.,$_{B•0}$¹, " 2•, " 1º where ¹B•0 leads to the error state. Then, we have

$$- = {}^{1}\text{"}\ {}_{2}\bullet\ \bullet\%\theta \quad {}^{1}\text{"}\ {}_{1}\bullet\ \bullet\%\theta$$
$$= \mathrm{ff}\ 2\,14\ {}^{1}, \text{"}\ {}_{2}{}^{o}\,\mathrm{j}\,9\ {}_{B\bullet 0}{}^{1}, \text{"}\ {}_{2}\bullet, \text{"}\ {}_{1}{}^{o}\bullet \tag{4}$$
$$9\mathrm{f}\ {}^{0}2\ {}_{B\bullet 0}{}^{1}, \text{"}\ {}_{2}\bullet, \text{"}\ {}_{1}{}^{o} : ?\mathrm{A}458\mathrm{f}\mathrm{G}\bullet\mathrm{f}{}^{o}\mathrm{g}$$

Finally, therepresentation of - = ¹" 2• •%θ ¹" 1• •%θ is a nite set of shortest traces of$_{B•0}$¹, " 2•, " 1º.

We apply the same process to = ¹"• •%$_{2}$º ¹"• •%$_{1}$º. By assuming that14 ¹ U " º 14 ¹, %$_{1}$º and computing ¹, %$_{2}$• , %$_{1}$º and its equivalence classes, we have

$$- = {}^{1}\text{"}\bullet\ \bullet\%{}_{2}{}^{o}\quad {}^{1}\text{"}\bullet\ \bullet\%{}_{1}{}^{o}$$
$$= \mathrm{ff}\ 2\,14\ {}^{1}, \%{}_{2}{}^{o}\,\mathrm{j}\,9\ {}_{B\bullet 0}{}^{1}, \%{}_{2}\bullet, \%{}_{1}{}^{o}\bullet \tag{5}$$
$$9\mathrm{f}\ {}^{0}2\ {}_{B\bullet 0}{}^{1}, \%{}_{2}\bullet, \%{}_{1}{}^{o} : ?\mathrm{A}458\mathrm{f}\mathrm{G}\bullet\mathrm{f}{}^{o}\mathrm{g}$$

Then, therepresentation of - = ¹"• •%$_{2}$º ¹"• •%$_{1}$º is a nite set of shortest traces of$_{B•0}$¹, %$_{2}$•, %$_{1}$º.

7 CASE STUDIES

This section reports on our experience applying our proposed method to evaluate the robustness of software designs. In particular, our goal was to answer the following research questions: (1) Does our proposed notion of robustness capture the types of environmental deviations that occur in practice? (2) Is our notion of robustness applicable across multiple domains? To answer (2), we demonstrate the application of our method to two di erent types of systems: namely, network protocols and safety-critical interfaces. For (1), we show that the robustness computed by our method indeed corresponds to environmental deviations that have been studied in the respective domains.

Data availability All of the implementation code and models used in our case studies will be made available open source and archived on a public repository upon acceptance.

7.1 Implementation

We created our robustness analyzer on top of LTSA [30, 31], a modeling tool that supports automated reachability-based analysis of labelled transition systems. In our tool, the LTS's corresponding to the input system, environment, and property are speci ed using FSP, the input modeling language of LTSA. We implement the functions including weakest assumption generation, representation generation, and explanation generation in a Kotlin program (a JVM-based language). In particular, we take advantage of the built-in tool support of LTSA for composition, projection, and property checking over LTS. Our evaluation was done on a Windows machine with 3.6GHz CPU and 32GB memory.

7.2 Network Protocol Design

This section describes a case study on rigorously evaluating the robustness of network protocol designs. In particular, we focus on two protocols: A naive protocol that assumes a perfectly reliable communication channel, and the Alternate Bit Protocol (ABP) [9], which is speci cally designed to guarantee integrity of messages over a potentially unreliable communication channel. By computing and comparing the robustness of the two, we formally show that the ABP is indeed more robust than the naive protocol against possible failures in the channel. As far as we know, our method is the rst automated technique for formally evaluating the robustness of network protocols.

7.2.1 Models. Figure 6 shows the LTS's for the environment and machines (i.e., network protocols). Here, the environment corresponds to a communication channel over which messages are transmitted (with U = fB4=30""1¼A42:0""1¼02:»0""1¼64C02:0""1¼g). Under normal circumstances, we expect that the channel reliably delivers messages to the intended receiver (i.e., it does not lose, duplicate, or corrupt messages); this model of the normative environment is captured as the perfect channel in Figure 6(a).

A machine in this case study corresponds to a network protocol whose goal is to reliably deliver each message from the sender to its intended receiver. In particular, we compare two protocols: A naive protocol " # , which simply sends and receives messages assuming the channel is reliable, and the Alternate Bit Protocol (ABP) % , which is designed to ensure reliable delivery even in presence of potential faults in the underlying channel. Figure 6(b) and 6(c) show their speci cations respectively.

7.2.2 Computing Robustness and Explanations. We de ned property % as the input and output should alternate ; in FSP:

```
property P = (input -> output -> P).
```

This property ensures that the sender sends a new message only after it receives the receiver's acknowledgement that the previously sent message was successfully delivered.

[1] B4=30""1¼refers to a set of actions fB4=30¼B4=31¼g

Figure 6: (a) The perfect channel: the transmission channel transmits messages with parameter 0 or 1 from the sender to the receiver; and the acknowledge channel transmits acknowledgements from the receiver back to the sender. (b) The naive protocol: The sender sends user input data with either 0 or 1, and waits on the acknowledgement; the receiver waits on messages, output the data, and acknowledges with either 0 or 1. (c) The ABP protocol [16]: The sender rst sends a message with 0, and it continues sending the message until it receives an acknowledgement with 0. Then, it alternates the bit to send a message with 1. The receiver rst waits on a message with 0, and it continues sending acknowledgements with 0 until it receives a new message with 1. Then, it acknowledges with 1 and waits for a new message with 0.

We used our tool to compute the robustness of the two protocols, i.e., ¹" # • •%ℓ and ¹" % • •%ℓ. Speci cally, contains 9 states and 24 transitions; " # contains 20 states and 67 transitions, and our tool spent 130ms to generate A4? ¹" # • •%ℓ and build their explanations. A4? ¹" # • •%ℓ contains 4 traces corresponding to 4 equivalence classes. " % contains 30 states and 104 transitions, and our tool spent 1s317ms to generate A4? ¹" % • •%ℓ and their explanations. A4? ¹" % • •%ℓ contains 107 traces corresponding to 107 equivalence classes.

7.2.3 Analysis. We built a deviation model which contains message loss, duplication, and corruption of bits (only the bit parameter 0 and 1, but not the message content) to provide explanations for these representative traces. Figure 7 shows its speci cation.

Figure 7: Deviation model that describes the faulty transmission channel. The faulty acknowledge channel is similarly structured and omitted here.

All the 4 traces in A4? ¹" # • •%ℓ correspond to the bit corruption error. For example, the explanation for hB4=3¼0A42¼1i is h8=?DC•B4=3¼2>AAD?C•2¼A42i. We were surprised to nd that the naive protocol is robust against such errors (our expectation was that the naive protocol would be not robust against any kind of environmental deviations at all). This is because property % is somewhat under-speci ed: It requires only that the input and output actions alternate, and does not say anything about the bit parameters in the sent and corresponding received messages.

For the 107 traces in A4? ¹" % • •%ℓ, our tool nds the minimal explanations for 99 of them. For example, the explanation for hB4=3¼0B4=3¼0i is h8=?DC•B4=3¼;>B4•B4=3¼0i corresponding to message loss during transmission; the explanation for hB4=3¼0A42¼0A42¼0i is h8=?DC•B4=3¼A42¼>DC?DC•3D?;820C4•A42¼0i corresponding to message duplication during transmission; and the explanation for hB4=3¼0A42¼0 2:•0¼64C02¼1i is h8=?DC•B4=3¼0A42¼0>DC?DC•0¼2>AAD?C•64C02¼0i corresponding to the bit corruption error during acknowledgement.

Fault types	#Traces	Fault types	#Traces
trans.lose	23	ack.duplicate	14
trans.duplicate	18	trans.{duplicate,corrupt}	4
trans.corrupt	8	ack.{duplicate,corrupt}	2
ack.lose	22	unexplained	8
ack.corrupt	8	Total	107

Table 1: Summary of A4? for ABP. trans refers to errors during transmission, and ack refers to errors during acknowledgements.

We further grouped the representative traces by the type of fault in their explanations, as shown in Table 1. For example, trans.{ duplicate, corrupt} represents a set of deviations in which the transmitted message is duplicated and then corrupted (e.g., hA42¼0A42¼1i). There may be multiple representative traces of the same fault type, since the fault may occur at di erent points during an expected sequence of environmental actions.

Our analysis shows that the ABP protocol is more robust than the naive protocol in being able to handle message loss and duplication, as intended by the protocol design [39]. In addition, the 8 unexplained traces also gave us an insight into a type of error that ABP was previously unknown to be robust against; namely, that the sender may receive acknowledgments even when the receiver does not send them. This type of deviation may occur, for example, when a malicious channel generates a dubious acknowledgement to deceive the sender into believing that a message has been delivered.

7.3 Radiation Therapy System

The second case study focuses on the radiation therapy system introduced in Section 2. Speci cally, we compare the robustness of the two designs (i.e., the original design and the redesign involving an additional check to ensure the completion of the mode switch before beam delivery) and show that the redesign is indeed more robust against potential human errors. In particular, to model normative and erroneous human behavior, we adopt the Enhanced Operator Function Model (EOFM) [37], a formal notation for modeling tasks performed over human-machine interfaces. Human behavior modeling has been studied by researchers in human factors and cognitive science [21, 35], and we reuse their results in this case study to demonstrate that our approach can be combined with existing behavior models in elds other than network protocols.

7.3.1 EOFM. The Enhanced Operator Function Model (EOFM) [37] is a formal description language for human task analysis, a well-established sub eld of human factors that focuses on the design of human operator tasks and related factors (e.g., training, working conditions, and error prevention) [21]. An EOFM describes the task to be performed by an operator over a machine interface as a hierarchical set of activities. Each activity includes a set of conditions that describe (1) when the activity can be undertaken (pre-conditions) and (2) when it is considered completed (completion conditions). Each activity is decomposed into lower-level sub-activities and, nally, into atomic interface actions. Decomposition operators are used to specify the temporal relationships between the sub-activities or actions. The EOFM language is based XML, and it also supports a tree-like visual notation.

Figure 8: The EOFM model of the Beam Selection Task. A rounded box de nes an activity, a rectangular box de nes an atomic action, and a rounded box in gray includes all the sub-activities/actions of a parent activity. The labels on the directed arrows are decomposition operators. The triangle in yellow de nes the pre-conditions of an activity, and the triangle in red de nes the completion conditions.

Figure 8 shows a fragment of the EOFM model of the operator's tasks for the radiation therapy system (from [39]). It de nes the Beam Selection Task, which can be performed only if the interface is in the Editing state; the operator can select either X-ray or electron beam by pressing X or E, respectively; and the activity is completed only if the interface leaves the editing state.

7.3.2 Models. The LTS's used for this case study (shown in Figure 1) were adopted from a prior work on formal safety analysis of radiation therapy system under potential human errors [39], where the system is modeled as a nite state machine and the human operator task is speci ed using an EOFM. Adopting their system model into our LTS was straightforward. To translate the EOFM to a corresponding LTS, we implemented an automatic EOFM-to-LTS translator using a technique proposed in [10]; due to limited space, we omit the details about our translation process.

7.3.3 Deviation Model. To generate explanations for that involve human errors, we adopted a method for automatically augmenting a model of a normative operator task (speci ed in EOFM) with additional behaviors that correspond to human errors [38]. In particular, this approach leverages a catalog of human errors called genotypes [35]. For example, one type of genotype errors named commission describes errors where the operator accidentally performs an activity under a wrong condition. Other genotype errors include omission (skipping an activity) and repetition.

Figure 9: A partial deviation model of the operator task.

Figure 9 shows a simpli ed version of the deviation model that was automatically generated from the EOFM model of the therapist task. This model captures the operator making a potential commission error; i.e., deviating from the expected task by pressing Up. For simplicity, we only show one faulty transition here; the complete deviation model is considerably more complex, since commission, omission, or repetition errors can occur at any state in the normative operator model.

7.3.4 Comparing Robustness of $\mathcal{A}$ and $\mathcal{A}'$. We compared the robustness of the two designs by computing $\mathcal{A} = \mathcal{A}' \setminus (\mathcal{A}' \setminus \mathcal{A})$ (using Equation (4)) and generated representative traces that illustrate di erences in their robustness. Speci cally, $\mathcal{A}$ contains 19 states and 40 transitions; $\mathcal{A}'$ contains 19 states and 42 transitions. Our tool spent 958ms to compute the representation of $\mathcal{A}$, which contains 3 representative traces (i.e., implying that $\mathcal{A}'$ is more robust than $\mathcal{A}$ against three types of operator deviations). One of the traces represents the error that was discussed in Section 2: $\langle$X, Up, E, Enter, Bi$\rangle$. This shows that the redesign is indeed robust against the operator error involving the switch from X-ray to EBeam. Moreover, we used the deviation model to generate the following minimal explanation for this trace: $\langle$X, commission, Up, E, Enter, Bi$\rangle$, corresponding to the operator making a commission error by unexpectedly pressing Up during the task.

In addition, computing $\mathcal{A}' \setminus (\mathcal{A}' \setminus \mathcal{A})$ yielded an empty set, demonstrating that the redesign of the system is strictly more robust than the original design.

7.3.5 Comparing Robustness Under Two Properties. Recall that property φ states that the system should not fire X-ray when the spreader is out of place. It may also be desirable to ensure that the system does not fire electron beam when the spreader is in place (for example, resulting in under-dose, which, while not as life-threatening as overdose, is still considered a critical error.) Let φ' be a property stating that the system must prevent both overdose as well as under-dose by ensuring the right mode of beam depending on the configuration of the spreader. Intuitively, φ' is a stronger property than φ.

To compare the robustness of the system against these two properties, we computed $d = \mathcal{R}(\mathcal{M}, \varphi) - \mathcal{R}(\mathcal{M}, \varphi')$ by using Equation (5). Our tool spent 2s98ms and returned one representative trace, i.e., $\langle$E, Up, X, Enter, Bi$\rangle$. Since this behavior is allowed in $\mathcal{M}$, but not in $\mathcal{R}(\mathcal{M}, \varphi')$, we can conclude from the the analysis that the the system (as expected) is less robust in establishing the stronger property φ' under potential operator errors.

8 RELATED WORK

Most of the prior works on robustness within the software engineering community have focused on testing [36]. Techniques such as fuzz testing (e.g., [18]), model-based testing (particularly those that use a fault model [4, 14]) and chaos testing [3] are designed to evaluate the robustness of systems against unexpected inputs or environmental failures. However, the primary goal of these techniques is to identify undesirable system behaviors (e.g., crashes or security violations) rather than to compute robustness as an intrinsic characteristic of the software. In addition, we believe that our robustness metric can potentially be used to complement and further systematize robustness testing; for instance, traces in $\mathcal{C}$ could be used to guide the generation of test cases that are designed to evaluate the system against specific types of environmental deviations.

Various formal definitions of robustness for discrete systems have been investigated [5, 19, 20, 37]. One common characteristics of these prior definitions is that they are all quantitative in nature. For instance, Bloem et al. propose a notion of robustness that relates the number of incorrect environment inputs and system outputs (e.g., the ratio of incorrect outputs over inputs should be small) [5]. Tabuada et al. propose a different notion of robustness that assigns costs to certain input and output traces (e.g., a high cost may be assigned to an input trace that deviates significantly from the expected behavior) and stipulates that an input with a small cost should only result in an output with a proportionally small cost [37]. Henzinger et al. adopt the notion of Lipschitz continuity from the control theory to discrete transition systems and use the distance between a pair of expected and actual input traces to quantify the amount of environmental deviations [19, 20].

In comparison, our notion of robustness is qualitative in that it captures the (possibly infinite) set of environmental deviations under which the system guarantees a desired property. These two types of metrics are complementary in nature and have their own potential uses. While a quantitative metric may directly enable ordering of design alternatives, our robustness contains additional information about the environmental behaviors (e.g., specific types of deviations) that can be used to improve the system robustness.

Tabuada and Neider propose an extension of linear temporal logic called robust linear temporal logic (rLTL), which allows specifications stipulating that a small violation of the environment assumption must cause only a small violation of the guarantee by the system [38]. In particular, they use a multi-valued semantics to capture different levels of property satisfaction by the environment (e.g., given an expected property of form $G\,i$, being able to satisfy only a weaker property $F^{1}G\,i$ would be considered a small violation) [37]. Although the focus of our paper is on computing robustness rather than specifying it, rLTL could potentially be used to characterize certain types of deviations that are temporal in nature.

Our notion of robustness can be regarded as one way of characterizing uncertainty about the environment under which the system is capable guaranteeing a certain property. Researchers have developed various notations and analysis techniques for specifying and reasoning about uncertainty [1, 11, 13, 23, 28]. For example, modal transition systems (MTS) allow one to express uncertainty about behavior by assigning a modality to transitions (e.g., a transition that can possibly but not necessarily occur is assigned modality may) [28]. More recently, partial models have been developed as a general modeling framework for specifying and reasoning about uncertainty on structural or behavioral aspects of a system [13]. Although the approach in this paper uses a purely trace-based encoding of robustness, these existing notations could potentially be used to provide a more high-level representation of robustness.

In safety engineering and risk management, operating envelope (or sometimes, safety envelope) has been used to refer to the boundary of environmental conditions under which the system is capable of maintaining safety [34]. This concept has been adopted in a number of domains such as aviation, robotics, and manufacturing, but as far as we know, has not been rigorously defined in the context of software. Our notion of robustness can be considered as one possible definition of the operating envelope for software systems.

9 LIMITATIONS AND DISCUSSIONS

One limitation of the proposed approach is that our current notion of robustness is specifically designed for safety properties. As a next step, to enable reasoning about liveness properties [1], we plan to investigate an extended notion of robustness where the environment deviates from its expectation not only by performing additional behaviors, but also by failing to perform expected behaviors (thus possibly resulting in a liveness violation).

Another limitation that we plan to address is that our current method of defining equivalence classes for $\mathcal{C}$ may sometimes result in a classification that is too fine-grained. For example, for the ABP protocol, our tool generated 107 different classes of environmental deviations (see Section 7.2). Intuitively, traces $\langle$Bi=3, Bi=4, Bi=3$\rangle$ and $\langle$Bi=3, Bi=4, Bi=2$\rangle$ refer to the same type of fault (i.e., message loss during sending) and could be grouped into the same class. In future work, we plan to explore different strategies for generating representative traces, leveraging abstraction-based methods to produce higher-level representations of deviations (e.g., $\langle$Bi=$\mathcal{G}$, Bi=$\mathcal{G}$$\rangle$ for some event parameter $\mathcal{G}$).

One potential future direction is to develop an approach for systematically redesigning a system to improve its robustness: Given machine $\mathcal{M}$ and some environmental deviations $\mathcal{X}$ under which the