A Behavioral Notion of Robustness for Software Systems

Changjian Zhang
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
changjiz@andrew.cmu.edu

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
garlan@cs.cmu.edu

Eunsuk Kang
School of Computer Science
Carnegie Mellon University
Pittsburgh, USA
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.

CCS CONCEPTS

• **Software and its engineering** → **Designing software**; *Formal software verification*; *Software system models*; **Software reliability**; *Software safety*.

KEYWORDS

software robustness, formal methods, labelled transition systems, compositional reasoning

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 '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409753

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.

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 [35]. 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., [18, 30]), 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 [43]. 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.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11.
https://doi.org/10.1145/3368089.3409753

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 [7, 38] 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.

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 [32]. State machines in Figure 1 describe three components in the system, including the *treatment interface* (M_I), 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_B), 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_S), 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, $M = (M_I \| M_B \| M_S)$.

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) [36]:

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

where BeamDelivered is a proposition that holds when M_I 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 [14] 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) [38]. 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.

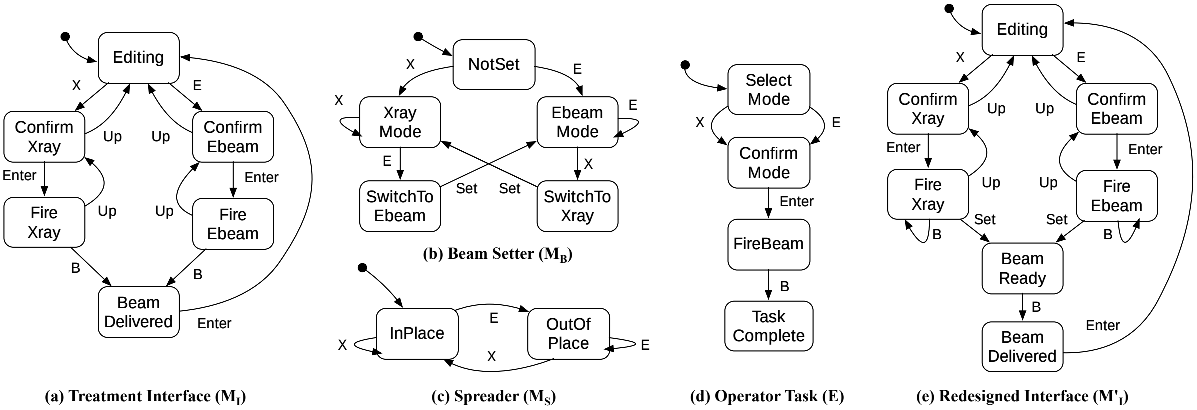
To discover these potential environmental deviations, the designer decides to perform the following analysis task:

> 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 [33]) 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, Up, E, Enter, B⟩. 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 confirming the mode and firing the beam). This sequence of operator actions, however, may lead to a violation of the safety property P in the following way: When the operator presses B, 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 [32].

(3) Robustness analysis. Having discovered how the operator's mistake could lead to a safety violation, the designer modifies the treatment interface to improve its robustness against the possible error. In this redesign, shown in Figure 1(e), the operator can press B to fire 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).



(a) Treatment Interface (M_I) **(b) Beam Setter (M_B)** **(c) Spreader (M_S)** **(d) Operator Task (E)** **(e) Redesigned Interface (M'_I)**

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

The designer could check $M' \models P$ where M' is the redesign, if no errors returned, it means that M' is robust against the mistake and also M' 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 satisfies the property but the old design does not?

Given the original system model M, modified system model M', normative environment E, and property P, our analysis returns a set of traces (expressed over environmental actions), each trace describes a scenario where system M' satisfies the property but M does not. For example, one of the traces is the sequence of operator actions discussed above: ⟨X, Up, E, Enter, B⟩, confirming 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 define 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* T is a tuple $\langle S, \alpha T, R, s_0\rangle$ where S is a set of states, αT is a set of actions called the *alphabet* of T, $R \subseteq S \times \alpha T \cup \{\tau\} \times S$ defines the state transitions (where τ is a designated action that is unobservable to the system's environment), and $s_0 \in S$ is the initial state. An LTS is *non-deterministic* if $\exists (s, a, s'), (s, a, s'') \in R : s' \neq s''$; otherwise, it is *deterministic*.

A trace $\sigma \in \alpha T^*$ of an LTS T is a sequence of observable actions from the initial state. Then, the behavior of T is the set of all the traces generated by T, denoted $beh(T)$.

3.1.2 Operators. For an LTS $T = \langle S, \alpha T, R, s_0\rangle$, the *projection* operator $\upharpoonright$ is used to expose only some subset of the alphabet of T. Given $T \upharpoonright A = \langle S, \alpha T \cap A, R', s_0\rangle$ where for any $(s, a, s') \in R$, if $a \notin A$, then $(s, \tau, s') \in R'$, i.e., a will be hidden by τ; otherwise, $(s, a, s') \in R'$.

The $\upharpoonright$ operator can also be applied to traces. We use $\sigma \upharpoonright A$ to denote the trace that results from removing all the occurrences of actions $a \notin A$ from σ.

The *parallel composition* $||$ is a commutative and associative operator which combines two LTSs by synchronizing their common actions and interleaving the remaining actions. Let $T_1 = \langle S^1, \alpha T^1, R^1, s_0^1\rangle$ and $T_2 = \langle S^2, \alpha T^2, R^2, s_0^2\rangle$, $T_1||T_2$ is an LTS $T = \langle S, \alpha T, R, s_0\rangle$ where $S = S^1 \times S^2$, $\alpha T = \alpha T^1 \cup \alpha T^2$, $s_0 = (s_0^1, s_0^2)$, and R is defined as: For any $(s^1, a, s^{1'}) \in R^1$ and $a \notin \alpha T^2$, we have $((s^1, s^2), a, (s^{1'}, s^2)) \in R$; for any $(s^2, a, s^{2'}) \in R^2$ and $a \notin \alpha T^1$, we have $((s^1, s^2), a, (s^1, s^{2'})) \in R$; and for $(s^1, a, s^{1'}) \in R^1$ and $(s^2, a, s^{2'}) \in R^2$, we have $((s^1, s^2), a, (s^{1'}, s^{2'})) \in R$.

3.1.3 Properties. In this work, we consider a class of properties called *safety properties* [29]. In particular, a safety property P can be represented as a deterministic LTS that contains no τ transitions. It defines the acceptable behaviors of a system T over αP, and we say that an LTS T satisfies P (denoted $T \models P$) if and only if $beh(T \upharpoonright \alpha P) \subseteq beh(P)$.

We check whether an LTS T satisfies a safety property $P = \langle S, \alpha P, R, s_0\rangle$ by automatically deriving an *error* LTS $P_{err} = \langle S \cup \{\pi\}, \alpha P, R_{err}, s_0\rangle$ where π denotes the error state, and $R_{err} = R \cup \{(s, a, \pi) | a \in \alpha P \land \nexists s' \in S : (s, a, s') \in R\}$. With this P_{err} LTS, we test whether the error state π is reachable in $T||P_{err}$. If π is not reachable, then we can conclude that $T \models P$.

3.2 Robustness Definition

Let M be the LTS of a machine, E the LTS of the environment, and $\alpha E_M = \alpha M \cap \alpha E$ the common actions between the machine and the environment. Then, we say $M \upharpoonright \alpha E_M$ represents the set of all environmental behaviors that are *permitted* by machine M.

Machine M is said to be *robust* against a set of traces $\delta \subseteq beh(M \upharpoonright E_M)$ if and only if the system satisfies a desired property under a new environment (E') that is capable of additional behaviors in δ compared to the original environment (E):

Definition 3.1. Machine M is *robust* against a set of traces δ with respect to environment E and property P if and only if $\delta \subseteq beh(M \upharpoonright \alpha E_M)$, $\delta \cap beh(E \upharpoonright \alpha E_M) = \emptyset$, and for every E' such that $beh(E' \upharpoonright \alpha E_M) = beh(E \upharpoonright \alpha E_M) \cup \delta$, $M||E' \models P$.

The set of traces in δ are also called *deviations* of E' from E over αE_M. Then, the *robustness* of a machine is defined as the largest set of environmental deviations under which the system continues to satisfy a desired property:

Definition 3.2. The *robustness* of machine M with respect to environment E and property P, denoted $\Delta(M, E, P)$, is the set of traces δ such that M is robust against δ with respect to E and P, and there exists no δ' such that $\delta \subset \delta'$ and M is also robust against δ'.

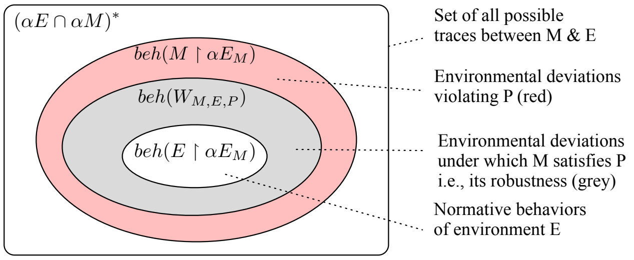

$(\alpha E \cap \alpha M)^*$
$beh(M \upharpoonright \alpha E_M)$
$beh(W_{M,E,P})$
$beh(E \upharpoonright \alpha E_M)$
Set of all possible traces between M & E
Environmental deviations violating P (red)
Environmental deviations under which M satisfies P i.e., its robustness (grey)
Normative behaviors of environment E

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 $\alpha E \cap \alpha M$. 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 classified into two sets: those under which the machine still maintains a desired property P (i.e., its robustness), and the others that lead to its violation (the area shaded red in Figure 2).

4 ANALYSIS PROBLEMS

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

Problem 4.1 (Robustness analysis)**.** Given machine M, environment E, and property P, compute $\Delta(M, E, P)$.

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 M_1 and M_2, environment E, and property P such that $\alpha M_1 \cap \alpha E = \alpha M_2 \cap \alpha E$, compute set $X = \Delta(M_2, E, P) - \Delta(M_1, E, P)$.

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. M_2, for example, may be an evolution of M_1; the result of the analysis would describe precisely the environmental deviations under which M_2 is more robust than M_1. Note that M_1 and M_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 M, environment E, and properties P_1 and P_2, compute set $X = \Delta(M, E, P_2) - \Delta(M, E, P_1)$.

For instance, suppose that P_1 says that "the radiation therapy system should always deliver the correct amount of dose to each patient", while P_2 states that "the system never overdoses patients by delivering X-ray while the spreader is out of place" (similar to property P from Section 2). The result of this analysis could tell us, for example, that the system is capable of guaranteeing P_2 (weaker and arguably more critical of the two) even under certain operator errors, while P_1 may be violated under similar deviations.

In general, since improving robustness might introduce additional complexity into the system, it may be a cost-effective strategy to design the system to be robust for most critical of the system requirements [27]; 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 M with respect to environment E and property P. The input of our tool is the LTS of M, E, and P. We first generate the *weakest assumption* of M (Section 5.2) to compute $\Delta(M, E, P)$. Since Δ may be infinite, we then generate a succinct representation of it. We compute the *representative* model of Δ (Section 5.3.1), group the traces into equivalence classes, and generate a finite 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 final output is a set of pairs of a representative trace and its explanation.

5.2 Weakest Assumption

In assume-guarantee style of reasoning [28], 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:

Definition 5.1. The *weakest assumption* $W_{M,E,P}$ of a machine M with respect to environment E and property P is an LTS which defines the largest subset of the permitted environment behaviors

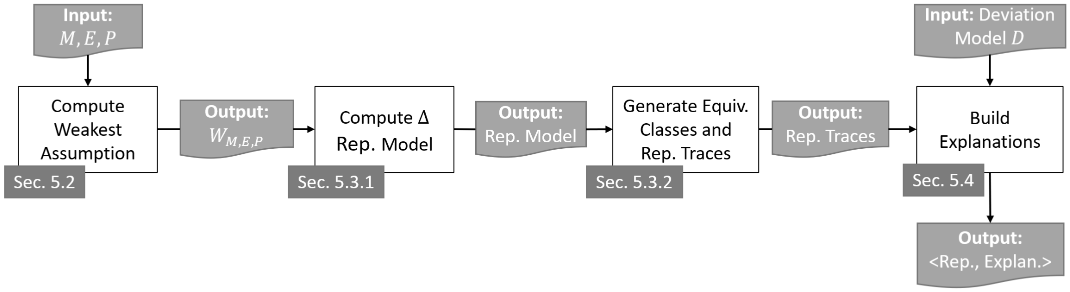


Figure 3: Overview of the process for robustness computation. The input is the LTS of machine M, environment E, and property P. Section 5.2 describes weakest assumption generation for computing Δ. Section 5.3 describes generating robustness representation (i.e., a set of representative traces). Finally,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

of M which satisfy property P, i.e.,

$$beh(W_{M,E,P}) \subseteq beh(M \restriction \alpha E_M) \ \wedge \ M||W_{M,E,P} \models P \ \wedge$$
$$\forall E' : M||E' \models P \leftrightarrow E' \models W_{M,E,P}$$

If stated otherwise, we will simply write W to mean $W_{M,E,P}$ 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 M with respect to environment E and property P by constructing the following set:

$$\Delta(M, E, P) = \{\sigma \in beh(W) \mid \sigma \notin beh(E \restriction \alpha E_M)\} \qquad (1)$$

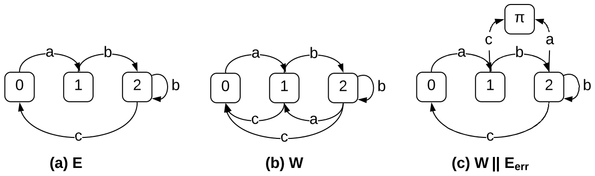
We use the approach by Giannakopoulou et al. [20] to generate the weakest assumption of a system to satisfy a certain safety property. We briefly describe the approach: Given the system LTS M, the environment LTS E, and the safety property P,

(1) Compose system M with the *error* LTS of property P (as defined in Section 3.1.3) and project its alphabet to the common actions between M and E, i.e., let $\alpha E_M = \alpha M \cap \alpha E$, we compute $(M||P_{err}) \restriction \alpha E_M$.
(2) Perform backward propagation of the error state over τ transitions in the LTS obtained from Step 1. We prune all states that are backward reachable from the error state via one or more τ 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 τ elimination and subset construction [25].
(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 infinite. Since simply enumerating this set may not be an effective way to present this information to the system designer, we propose a succinct, *finite* representation of the robustness. The key idea behind our approach is that many of the traces in $\Delta(M, E, P)$ 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 Δ into a finite number of such equivalence classes (and thus, a finite set of representative traces).

5.3.1 Representative Model of Robustness. Recall from Equation (1) that Δ contains traces that are in the weakest assumption W but not in the original normative environment E. To construct an LTS that represents Δ, 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 E projected over αE_M as a safety property, and compute traces in W that lead to a violation of this property; any such trace represents a prefix of the traces in $\Delta(M, E, P)$.



(a) E

(b) W

(c) W || E_{err}

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

To illustrate our approach, consider a simple example in Figure 4, where W is the weakest assumption generated from some machine M and E is the original environment. To compute the representation of $\Delta(M, E, P)$, we first test whether $W \models (E \restriction \alpha E_M)$, which is equivalent to testing whether the error state is reachable in $W||(E \restriction \alpha E_M)_{err}$, as shown in Figure 4(c). We say $W||(E \restriction \alpha E_M)_{err}$ is the *representative model* of $\Delta(M, E, P)$, and let $\Pi(W, E)$ be the set of all the error traces in it. Then,

$$\Delta(M, E, P) = \{\sigma \in beh(W) \mid \exists \sigma' \in \Pi(W, E) : prefix(\sigma', \sigma)\} \quad (2)$$

where $prefix(\sigma_1, \sigma_2)$ means σ_1 is the prefix of σ_2. Thus, a trace in $\Pi(W, E)$ can represent a set of traces in $\Delta(M, E, P)$ that share this prefix. For our example, trace $\langle a, c \rangle$ in $\Pi(W, E)$ can represent, e.g., $\langle a, c, a, b, \ldots \rangle$ and $\langle a, c, a, c, \ldots \rangle$ in $\Delta(M, E, P)$.

5.3.2 Representative Traces of Robustness. Nevertheless, $\Pi(W, E)$ may also be infinite due to possible cycles. For example, in Figure 4(c), $\langle a, b, b, \ldots, a \rangle$ would result in infinite number of error traces. Therefore, we further divide the traces into equivalence classes:

Let $T_{W,E} = \langle S_{W,E}, \alpha E_M, R_{W,E}, s_0 \rangle$ be the composition $W \| (E \upharpoonright \alpha E_M)_{err}$. Then,

$$\Pi(W,E) = \bigcup_{\substack{s \in S_{W,E} \\ a \in \alpha E_M}} \Pi_{s,a}(W,E) \text{ where } (s,a,\pi) \in R_{W,E}$$

i.e., $\Pi_{s,a}(W,E)$ denotes a subset of traces in $\Pi(W,E)$ that all end with transition (s,a,π). Then, we have

$$\Delta(M,E,P) = \{\sigma \in beh(W) \mid \exists \Pi_{s,a}(W,E), \exists \sigma' \in \Pi_{s,a}(W,E) : prefix(\sigma', \sigma)\} \quad (3)$$

We say that $\Pi_{s,a}(W,E)$ is an *equivalence* class of $\Pi(W,E)$. In our example, we have two equivalence classes: $\Pi_{1,c}(W, E)$ and $\Pi_{2,a}(W,E)$. Traces like $\langle a,c \rangle$ and $\langle a,b,c,a,c \rangle$ all belong to class $\Pi_{1,c}(W,E)$; and traces like $\langle a,b,a \rangle$ and $\langle a,b,b,b,a \rangle$ all belong to class $\Pi_{2,a}(W,E)$.

The rationale is that: s is the last state by following the normative behaviors of the original environment, and a is the first deviated action. Thus, $\Pi_{s,a}(W,E)$ describes a class of traces that deviate from the original environment from the same normative state s and by the same action a.

Since $S_{W,E}$ and αE_M 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 $\Delta(M,E,P)$. 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 $\Delta(M,E,P)$, denoted by $\Delta_{rep}(M,E,P)$, is a finite set of traces such that each trace in it is the shortest trace of one of the equivalence classes of $\Pi(W_{M,E,P}, E)$.

Therefore, for our conceptual example, $\Delta(M,E,P)$ can be represented by: $\Pi_{1,c}(W,E) : \langle a,c \rangle$, and $\Pi_{2,a}(W,E) : \langle a,b,a \rangle$.

5.4 Explanation of Representative Traces

By definition, a representative trace in $\Delta_{rep}(M,E,P)$ contains only actions from αE_M. 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 an *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.

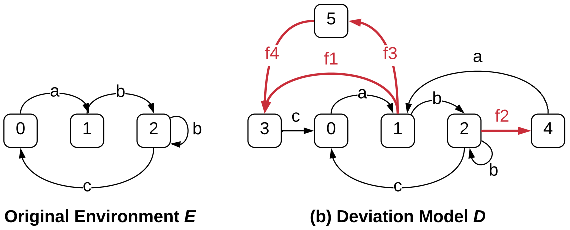
5.4.1 Explanations from a Deviation Model. In order to build explanations for representative traces, our tool takes a deviation model as input, which contains normative and deviated behaviors, and maps each representative trace to a trace in the deviation model.

Definition 5.3. A *deviation* model D of environment E is an LTS $T = \langle S, \alpha D, R, s_0 \rangle$ where $\alpha D = \alpha E \cup \{f_1, f_2, \ldots, f_n\}$, f_i is a fault in the environment, $beh(E) \subseteq beh(D \upharpoonright \alpha E)$, and $beh(D \upharpoonright \alpha E_M) \cap \Delta(M,E,P) \neq \emptyset$.

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 $\Delta(M,E,P)$; however, we say a deviation model is *complete* with respect to $\Delta(M,E,P)$ if and only if $\Delta(M,E,P) \subseteq beh(D \upharpoonright \alpha E_M)$.

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

Definition 5.4. For any trace $\sigma \in \Delta_{rep}(M,E,P)$ and $\sigma' \in beh(D)$, if $\sigma' \upharpoonright \alpha E_M = \sigma$, then we say σ' is an *explanation* of σ.



(a) Original Environment *E*

(b) Deviation Model *D*

Figure 5: Deviation model for the simple example.

Consider a deviation model for our simple example in Figure 5, then: for the representative trace $\langle a,c \rangle$, we can build explanations $\langle a, f_1, c \rangle$ and $\langle a, f_3, f_4, c \rangle$; and for the representative trace $\langle a,b,a \rangle$, we can build an explanation $\langle a, b, f_2, a \rangle$.

5.4.2 The Minimal Explanation. In general, 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 σ which contains the minimal number of faults.

Definition 5.5. The *minimal explanation* for $\sigma = \langle a_0, \ldots, a_{n-1}, a_n \rangle$ in $\Delta_{rep}(M,E,P)$ under deviation model D is the shortest trace $\sigma' \in beh(D)$ where $\sigma' \upharpoonright \alpha E_M = \sigma$ and faulty actions only exist between a_{n-1} and a_n.

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 $\sigma \in \Delta_{rep}(M,E,P)$, let $T_\sigma = \langle S, \alpha E_M, R, s_0 \rangle$ be the LTS where σ and its prefixes are the only traces in it. Besides, we make the last action in σ lead to π to denote the end state, i.e., $(s, a_n, \pi) \in R$. Then, we use BFS to search the minimal explanation in $D \| T_\sigma$, 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 π, 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 σ, i.e., $\langle a_0, \ldots, a_{n-1} \rangle$, 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

Algorithm 1: Minimal explanation search

```
Data: A trace $\sigma \in \Delta_{rep}(M, E, P)$ and the LTS of $D||T_\sigma$
Result: The minimal explanation $\sigma' \in beh(D)$
$q$ := empty queue ;            // remaining search states
$v$ := empty set of states ;              // visited states
enqueue($q$, $(s_0, \langle\rangle)$);
while $\neg$isEmpty($q$) do
    $s, t$ := dequeue($q$); // $s$ the current state, $t$ the
     current trace
    if $s \notin v$ then
        if $s = \pi$ then
            return $t$;
        else
            $v := v \cup \{s\}$;
            for $(s, a, s') \in R$ do
                if $t \upharpoonright \alpha E_M = subTrace(\sigma, 0, n-1)$ then
                  enqueue($q$, $(s', t \frown a)$) ;
                /* $t$ does not match $\langle a_0, \ldots, a_{n-1}\rangle$.   */
                else if $a$ is not a fault then
                  enqueue($q$, $(s', t \frown a)$) ;
            end
        end
    end
end
```

example, our algorithm returns $\langle a, f_1, c\rangle$ as the minimal explanation for $\langle a, c\rangle$ instead of $\langle a, f_3, f_4, c\rangle$ 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

$$\begin{aligned} X &= \Delta(M_2, E, P) - \Delta(M_1, E, P) \\ &= \{\sigma \in beh(W_{M_2}) \mid \sigma \notin beh(E \upharpoonright \alpha E_M) \wedge \sigma \notin beh(W_{M_1})\} \end{aligned}$$

By assuming $beh(E \upharpoonright \alpha E_M) \subseteq beh(W_{M_1})$, we can simplify the equation to:

$$X = \Delta(M_2, E, P) - \Delta(M_1, E, P) = \{\sigma \in beh(W_{M_2}) \mid \sigma \notin beh(W_{M_1})\}$$

Then, we can use the same method described in Section 5.3 to generate its representation. By computing $W_{M_2}||(W_{M_1})_{err}$, we have $\Pi(W_{M_2}, W_{M_1})$ representing all the prefixes of X. Similarly, we divide it into equivalence classes, i.e., $\Pi_{s,a}(W_{M_2}, W_{M_1})$ where (s, a) leads to the error state. Then, we have

$$\begin{aligned} X &= \Delta(M_2, E, P) - \Delta(M_1, E, P) \\ &= \{\sigma \in beh(W_{M_2}) \mid \exists \Pi_{s,a}(W_{M_2}, W_{M_1}), \\ &\qquad \exists \sigma' \in \Pi_{s,a}(W_{M_2}, W_{M_1}) : prefix(\sigma', \sigma)\} \end{aligned} \tag{4}$$

Finally, the *representation* of $X = \Delta(M_2, E, P) - \Delta(M_1, E, P)$ is a finite set of shortest traces of $\Pi_{s,a}(W_{M_2}, W_{M_1})$.

We apply the same process to $X = \Delta(M, E, P_2) - \Delta(M, E, P_1)$. By assuming that $beh(E \upharpoonright \alpha E_M) \subseteq beh(W_{P_1})$ and computing $\Pi(W_{P_2}, W_{P_1})$ and its equivalence classes, we have

$$\begin{aligned} X &= \Delta(M, E, P_2) - \Delta(M, E, P_1) \\ &= \{\sigma \in beh(W_{P_2}) \mid \exists \Pi_{s,a}(W_{P_2}, W_{P_1}), \\ &\qquad \exists \sigma' \in \Pi_{s,a}(W_{P_2}, W_{P_1}) : prefix(\sigma', \sigma)\} \end{aligned} \tag{5}$$

Then, the *representation* of $X = \Delta(M, E, P_2) - \Delta(M, E, P_1)$ is a finite set of shortest traces of $\Pi_{s,a}(W_{P_2}, W_{P_1})$.

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 different 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.

7.1 Implementation

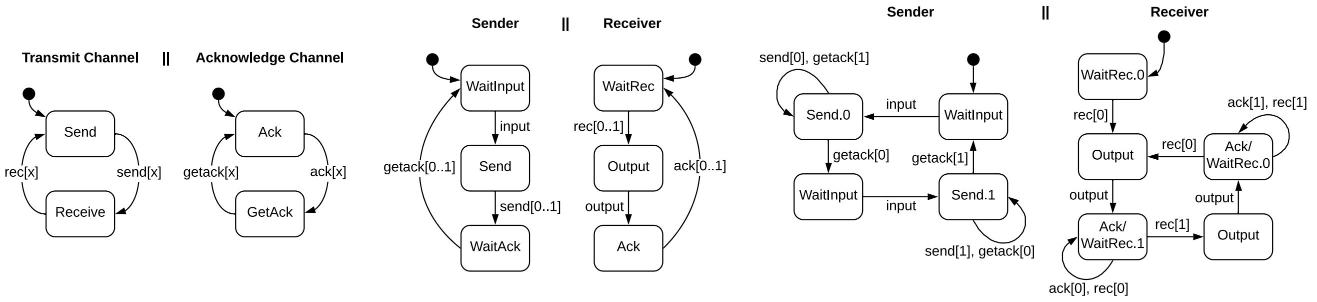
We created our robustness analyzer on top of LTSA [33, 34], a modeling tool that supports automated reachability-based analysis of labelled transition systems. In our tool, the LTS's corresponding to the input system, environment, and property are specified using FSP, the input modeling language of LTSA. 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. The source code of the implementation can be found on GitHub, https://github.com/SteveZhangBit/LTSA-Robust.

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) [42], which is specifically 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 first 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 E corresponds to a communication channel over which messages are transmitted (with $\alpha E = \{send[0..1], rec[0..1], ack[0..1], getack[0..1]\}$[1]). Under normal circumstances, we expect that the channel reliably delivers messages to the intended receiver (i.e., it does not lose,

[1] $send[0..1]$ refers to a set of actions $\{send[0], send[1]\}$.



(a) Perfect Channel (E) (b) Naive Protocol (M_N) (c) ABP Protocol (M_{ABP})

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 [19]: The sender first 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 first 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.

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 M_N, which simply sends and receives messages assuming the channel is reliable, and the Alternate Bit Protocol (ABP) M_{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 specifications respectively.

7.2.2 Computing Robustness and Explanations. We defined property P 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.

We used our tool to compute the robustness of the two protocols, i.e., $\Delta(M_N, E, P)$ and $\Delta(M_{ABP}, E, P)$. Specifically, E contains 9 states and 24 transitions, M_N contains 20 states and 67 transitions, and our tool spent 130ms to generate $\Delta_{rep}(M_N, E, P)$ and build their explanations. $\Delta_{rep}(M_N, E, P)$ contains 4 traces corresponding to 4 equivalence classes. M_{ABP} contains 30 states and 104 transitions, and our tool spent 1s317ms to generate $\Delta_{rep}(M_{ABP}, E, P)$ and their explanations. $\Delta_{rep}(M_{ABP}, E, P)$ contains 107 traces corresponding to 107 equivalence classes.

7.2.3 Analysis. We built a deviation model D 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 specification.

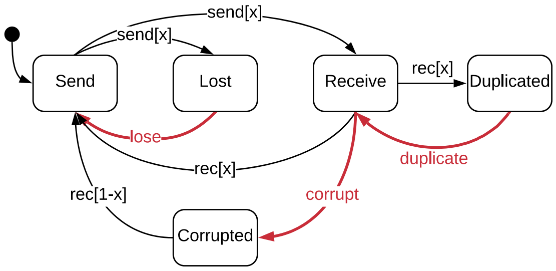


Figure 7: Deviation model that describes the faulty transmission channel. The faulty acknowledge channel is similarly structured and omitted here.

All the 4 traces in $\Delta_{rep}(M_N, E, P)$ correspond to the bit corruption error. For example, the explanation for $\langle send[0], rec[1]\rangle$ is $\langle input, send[0], corrupt, rec[1]\rangle$. We were surprised to find 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 P is somewhat under-specified: 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 $\Delta_{rep}(M_{ABP}, E, P)$, our tool finds the minimal explanations for 99 of them. For example, the explanation for $\langle send[0], send[0]\rangle$ is $\langle input, send[0], lose, send[0]\rangle$ corresponding to message loss during transmission; the explanation for $\langle send[0], rec[0], rec[0]\rangle$ is $\langle input, send[0], rec[0], output, duplicate, rec[0]\rangle$ corresponding to message duplication during transmission; and the explanation for $\langle send[0], rec[0], ack[0], getack[1]\rangle$ is $\langle input, send[0], rec[0], output, ack[0], corrupt, getack[1]\rangle$ corresponding to the bit corruption error during acknowledgement.

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., $\langle ..rec[0], rec[1]\rangle$). There may be multiple representative traces of the same fault type, since the fault may occur at different points during an expected sequence of environmental actions.

Table 1: Summary of Δ_{rep} for ABP. "trans" refers to errors during transmission, and "ack" refers to errors during acknowledgements.

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**

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 [42]. 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. Specifically, 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) [8], 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 [24, 38], and we reuse their results in this case study to demonstrate that our approach can be combined with existing behavior models in fields other than network protocols.

7.3.1 EOFM. The Enhanced Operator Function Model (EOFM) [8] is a formal description language for *human task analysis*, a well-established sub-field of human factors that focuses on the design of human operator tasks and related factors (e.g., training, working conditions, and error prevention) [2]. 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 complete (*completion conditions*). Each activity is decomposed into lower-level sub-activities and, finally, 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 shows a fragment of the EOFM model of the operator's tasks for the radiation therapy system (from [10]). It defines 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.

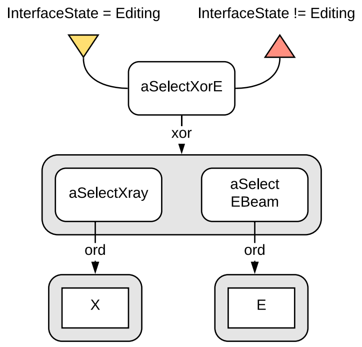


Figure 8: The EOFM model of the Beam Selection Task. A rounded box defines an activity, a rectangular box defines an atomic action, and a rounded box in gray includes all the sub-activities/actions of a parent activity. The labels on the directed arrows are decomposition operators. The triangle in yellow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

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 [10], where the system is modeled as a finite state machine and the human operator task is specified 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 [11]; due to limited space, we omit the details about our translation process.

7.3.3 Deviation Model. To generate explanations for Δ that involve human errors, we adopted a method for automatically augmenting a model of a normative operator task (specified in EOFM) with additional behaviors that correspond to human errors [9]. In particular, this approach leverages a catalog of human errors called *genotypes* [38]. 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.

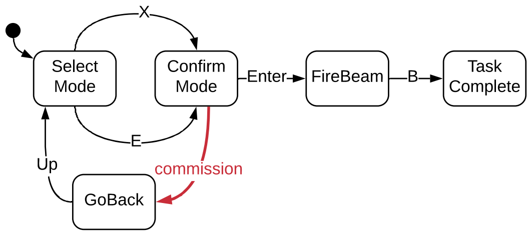


Figure 9: A partial deviation model of the operator task.

Figure 9 shows a simplified 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 M and M_R. We compared the robustness of the two designs by computing $X = \Delta(M_R, E, P) - \Delta(M, E, P)$ (using Equation (4)) and generated representative traces that illustrate differences in their robustness. Specifically, M contains 19 states and 40 transitions, M_R contains 19 states and 42 transitions. Our tool spent 958ms to compute the representation of X, which contains 3 representative traces (i.e., implying that M_R is more robust than M against three types of operator deviations). One of the traces represents the error that was discussed in Section 2: ⟨X, Up, E, Enter, B⟩. 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: ⟨X, commission, Up, E, Enter, B⟩, corresponding to the operator making a *commission* error by unexpectedly pressing Up during the task.

In addition, computing $\Delta(M, E, P) - \Delta(M_R, E, P)$ 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 P 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 P' 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, P' is a stronger property than P.

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

8 RELATED WORK

Most of the prior works on robustness within the software engineering community have focused on *testing* [39]. Techniques such as fuzz testing (e.g., [21]), model-based testing (particularly those that use a fault model [4, 17]) 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 Δ 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, 22, 23, 40]. 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 [40]. 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 [22, 23].

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 [41]. 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 $\mathbf{G}\varphi$, being able to satisfy only a weaker property $\mathbf{F}(\mathbf{G}\varphi)$ would be considered a "small" violation) [40]. 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 [12, 16, 26, 31]. 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*) [31]. 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 [16]. 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 [37]. 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 DISCUSSIONS

Role of robustness in software development. We envision a number of development scenarios in which our notion of robustness may prove useful. In a typical workflow, the developer may begin by constructing a candidate design that satisfies a desired property under a normative environment (i.e., one without any erroneous behaviors). Then, the robustness computed over this initial design may reveal that the system is not capable of tolerating certain deviations that the developer has in mind. Based on this information, the system may be re-designed with an error handling mechanism and analyzed again to compute its new robustness. This process may be repeated to incrementally achieve a desired level of robustness.

Our analysis may also be used to reveal that a design is *over-engineered*, in that it is robust against deviations that are unlikely to occur (this situation may arise, for example, when a system is deployed in a more constrained environment than originally anticipated). Over-engineering has its cost, often in form of additional complexity, and thus, the developer may wish to simplify the design to reduce its robustness to a desired level (e.g., by removing unnecessary failure handling mechanisms).

Our approach also supports development of *mixed-criticality* systems [13], where some of the system requirements are considered more critical than others (e.g., in certain distributed systems, preventing network message corruption may be more important than ensuring timely delivery). Such a system should be designed to satisfy its critical properties even under a faulty environment, while it might be considered acceptable for other, non-critical properties to be violated under the same situation. By applying the property comparison analysis as stated in Problem 4.3, the developer can rigorously check whether a given design achieves appropriate levels of robustness for properties with different levels of criticality.

Strengths of our robustness definition. Our robustness notion provides information about: (1) what additional environmental behaviors the system can handle compared to the ideal environment or an alternative design, and (2) what errors in the environment these additional behaviors represent. For (1), we compute the differences between the weakest assumption of a system and the normative environment to denote its robustness; and we compare the robustness of two designs by computing the differences in their weakest assumptions. Since robustness, in general, is an infinite trace set, our approach provides a technique for categorizing robustness in terms of a finite number of representative traces (Section 5.3).

For (2), a deviation model is used to generate an explanation that describes a deviation in terms of designated faulty events (Section 5.4). As we demonstrated on the radiation therapy system, these models can be constructed automatically from domain knowledge that captures a set of common deviations in an application domain (e.g., human errors). In general, a deviation model might not contain enough faulty events to produce an explanation for a particular representative trace, identified as an unexplained deviation. However, we believe that this can also be considered a strength of our approach, since these unexplained traces reveal the unexpected side-effects (may or may bot be good) of a design decision and can provide domain experts with insights about previously unknown types of deviations (e.g., ABP being robust against an injection of a dubious acknowledgement, as described at the end of Section 7.2).

10 LIMITATIONS AND FUTURE WORK

Properties. 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).

Scalability. The scalability of our tool highly depends on the weakest assumption generation algorithm; the generation of the representative traces and explanations involves only LTS composition and BFS search. As part of future work, we plan to improve our tool by leveraging techniques for efficient generation of weakest assumptions (e.g., those that use L^* learning [15] or a counterexample-guided method [6]).

Classifying deviations. Another limitation is that our current method of defining equivalence classes for Δ 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 $\langle send[0], send[0] \rangle$ and $\langle \ldots, send[1], send[1] \rangle$ 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., $\langle \ldots, send[x], send[x] \rangle$ for some event parameter x).

Redesigning for robustness. One potential future direction is to develop an approach for systematically redesigning a system to improve its robustness: Given machine M and some environmental deviations δ under which the system fails to satisfy property P, how do we redesign the system to be robust against such deviations (i.e., $\delta \subseteq \Delta(M', E, P)$ for redesigned machine M')? In particular, we plan to formulate this problem as a type of model transformation (from M to M'), and explore algorithmic methods for (semi-)automatically synthesizing the robust redesign.

ACKNOWLEDGMENTS

We'd like to thank Daniel Jackson, Stéphane Lafortune, and Stavros Tripakis for their discussions on robustness, and anonymous reviewers for their suggestions that helped greatly improve this paper.

This work was supported in part by the National Science Foundation awards CCF-1918140 and CNS-1801546. The project was also co-financed by the ERDF - European Regional Development Fund through the Operacional Program for Competitiveness and Internationalisation - COMPETE 2020 and by the Portuguese Foundation for Science and Technology - FCT under CMU Portugal. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. (DM-0004369).

REFERENCES

[1] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[2] J. Annett and N. A. Stanton. *Task Analysis*. CRC Press, 2000.

[3] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, May 2016.

[4] F. Belli, A. Hollmann, and W. E. Wong. Towards scalable robustness testing. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 208–216. IEEE, 2010.

[5] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann. Specification-centered robustness. In *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on, SIES 2011. Vasteras, Sweden, June 15-17, 2011*, pages 176–185, 2011.

[6] M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *International Conference on Computer Aided Verification (CAV)*, volume 5123, pages 135–148, 2008.

[7] M. L. Bolton. A task-based taxonomy of erroneous human behavior. *Int. J. Hum. Comput. Stud.*, 108:105–121, 2017.

[8] M. L. Bolton and E. J. Bass. Enhanced operator function model: A generic human task behavior modeling language. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, pages 2904–2911. IEEE, 2009.

[9] M. L. Bolton and E. J. Bass. Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(6):1314–1327, 2013.

[10] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu. Generating phenotypical erroneous human behavior to evaluate human-automation interaction using model checking. *International Journal of Human Computer Studies*, 70(11):888–906, 2012.

[11] M. L. Bolton, R. I. Siminiceanu, and E. J. Bass. A systematic approach to model checking human-automation interaction using task analytic models. *IEEE Transactions on Systems, Man, and Cybernetics Part A:Systems and Humans*, 41(5):961–976, 2011.

[12] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pages 274–287, 1999.

[13] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, 2018.

[14] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[15] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning Assumptions for Compositional Verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

[16] M. Famelis, R. Salay, and M. Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *34th International Conference on Software Engineering, ICSE 2012, Zurich, Switzerland*, pages 573–583, 2012.

[17] J.-C. Fernandez, L. Mounier, and C. Pachon. A model-based approach for robustness testing. In *IFIP International Conference on Testing of Communicating Systems*, pages 333–348. Springer, 2005.

[18] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[19] D. Giannakopoulou, J. Kramer, and J. Magee. Practical behaviour analysis for distributed software architectures. In *UK Programmable Networks and Telecommunications Workshop*, 1998.

[20] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings - ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pages 3–12. IEEE, 2002.

[21] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[22] T. A. Henzinger, J. Otop, and R. Samanta. Lipschitz robustness of finite-state transducers. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pages 431–443, 2014.

[23] T. A. Henzinger, J. Otop, and R. Samanta. Lipschitz robustness of timed I/O systems. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 250–267, 2016.

[24] E. Hollnagel. *Cognitive Reliability and Error Analysis Method (CREAM)*. Elsevier Science, 1998.

[25] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[26] M. Huth, R. Jagadeesan, and D. A. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 155–169, 2001.

[27] D. Jackson and E. Kang. Separation of concerns for dependable software design. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 173–176, 2010.

[28] C. B. Jones. Specification and design of (parallel) programs. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 321–332, 1983.

[29] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[30] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[31] K. G. Larsen and B. Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 203–210, 1988.

[32] N. G. Leveson and C. S. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[33] J. Magee and J. Kramer. *State models and java programs*. wiley Hoboken, 1999.

[34] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Working Conference on Software Architecture*, pages 35–49. Springer, 1999.

[35] H. Petroski. *To engineer is human: The role of failure in successful design*. St Martins Press, 1985.

[36] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[37] J. Rasmussen. Risk management in a dynamic society: a modelling problem. *Safety Science*, 27(2):183 – 213, 1997.

[38] J. Reason. *Human Error*. Cambridge University Press, New York, 1990.

[39] A. Shahrokni and R. Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1–17, 2013.

[40] P. Tabuada, A. Balkan, S. Y. Caliskan, Y. Shoukry, and R. Majumdar. Input-output robustness for discrete systems. In *Proceedings of the 12th International Conference on Embedded Software, EMSOFT 2012, part of the Eighth Embedded Systems Week, ESWeek 2012, Tampere, Finland, October 7-12, 2012*, pages 217–226, 2012.

[41] P. Tabuada and D. Neider. Robust linear temporal logic. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, pages 10:1–10:21, 2016.

[42] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2 edition, 2000.

[43] K. Zhou and J. C. Doyle. *Essentials of Robust Control*. Prentice-Hall, 1998.