Safe Environmental Envelopes of Discrete Systems

Rômulo Meira-Góes[1(✉)], Ian Dardik[2], Eunsuk Kang[2], Stéphane Lafortune[3], and Stavros Tripakis[4]

[1] School of EECS, Pennsylvania State University, State College, USA
romulo@psu.edu

[2] School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
{idardik,eunsukk}@andrew.cmu.edu

[3] EECS Department, University of Michigan, Ann Arbor, USA
stephane@umich.edu

[4] Khoury College of Computer Science, Northeastern University, Boston, USA
stavros@northeastern.edu

Abstract. A safety verification task involves verifying a system against a desired safety property under certain assumptions about the environment. However, these environmental assumptions may occasionally be violated due to modeling errors or faults. Ideally, the system guarantees its critical properties even under some of these violations, i.e., the system is *robust* against environmental deviations. This paper proposes a notion of *robustness* as an explicit, first-class property of a transition system that captures how robust it is against possible *deviations* in the environment. We modeled deviations as a set of *transitions* that may be added to the original environment. Our robustness notion then describes the safety envelope of this system, i.e., it captures all sets of extra environment transitions for which the system still guarantees a desired property. We show that being able to explicitly reason about robustness enables new types of system analysis and design tasks beyond the common verification problem stated above. We demonstrate the application of our framework on case studies involving a radiation therapy interface, an electronic voting machine, a fare collection protocol, and a medical pump device.

Keywords: Robustness · Discrete Transition Systems · Model Uncertainty

1 Introduction

A common type of verification task involves verifying a system (C) against a desired property (P) under certain assumptions about the environment (E); i.e., $C||E \models P$. Such assumptions may capture, for example, the expected behavior of a human operator in a safety-critical system, the reliability of the communication

© The Author(s) 2023

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 326–350, 2023.
https://doi.org/10.1007/978-3-031-37706-8_17

channel in a distributed system, or the capabilities of an attacker. However, the actual environment (E') may occasionally deviate from the original model (E), due to changes or faults in the environment entities (e.g., errors committed by the operator or message loss in the channel). For certain types of deviations, a system that is *robust* would ideally be able to guarantee the property even under the deviated environment; i.e., $C||E' \models P$.

This paper proposes the notion of *robustness* as an explicit, first-class property of a transition system that captures how robust it is against possible *deviations* in the environment. A deviation is modeled as a set of *extra transitions* that may be added to the original environment, resulting in a new, deviated environment E' that has a larger set of behaviors than E does. Then, system C is said to be *robust* to this deviated environment with respect to P if and only if it can still guarantee P even in presence of the deviation. Finally, the overall *robustness* of C with respect to E and P, denoted Δ, is the largest set of deviations that the system is robust against.

Conceptually, Δ defines the safe operating envelopes of the system: As long as the deployment environment remains within these envelopes, the system can guarantee a desired property. Being able to explicitly reason about Δ enables new types of system analysis and design tasks beyond the common verification problem stated above. Given a pair of alternative system designs, C_1 and C_2, one could rigorously compare them with respect to their robustness levels; they both may satisfy property P under the normal operating environment E, but one may be more robust to deviations than the other. Given two properties, P_1 and P_2 (the latter possibly more critical than the former), one could check whether the system would continue to guarantee P_2 under a deviated environment even if it fails to do so for P_1. Finally, given E, P, and a desired level of robustness, Δ, one could *synthesize* machine C to be robust to Δ.

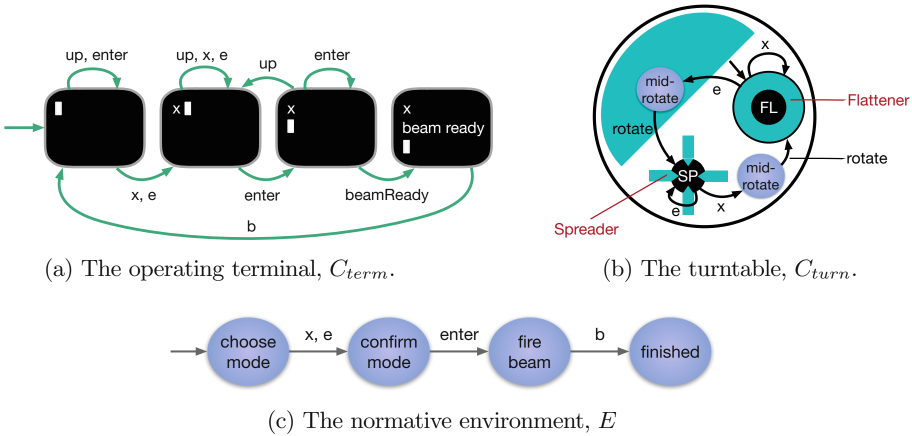
In this paper, we formalize (1) the proposed notion of robustness and (2) the problem of computing Δ for given C, E, and P. One approach to automatically compute Δ is a brute-force method that enumerates all possible sets of deviations; however, as we will show, this approach is impractical, as the number of deviations is exponential in the size of the environment. To mitigate this, we present an approach for computing Δ by reduction to a controller synthesis problem [35,37].

We have built a prototype of the proposed approach for computing robustness and applied it to several case studies, including models of (1) a radiation therapy interface, (2) an electronic voting machine, (3) a public transportation fare collection protocol, and (4) a medical pump device. Our results show that our approach is capable of computing Δ to provide information about deviations under which these systems are able to guarantee their critical safety properties.

The contributions of this paper are as follows: (i) A novel, formal definition of robustness against environmental deviations (Sect. 4); (ii) A simple, brute-force method for computing robustness and a more efficient approach based on controller synthesis (Sect. 5); and (iii) A prototype tool for computing Δ and an experimental evaluation on several case studies (Sect. 6).

2 Motivating Example

As a motivating example, we consider the Therac-25 radiation therapy machine. This machine is infamous for a design flaw that caused radiation overdoses, several of which led to the deaths of patients who received treatment [18]. In this section, we introduce a model for the Therac-25 based on the descriptions in [18] and discuss several methods for analyzing its safety. We show that robustness provides a generally richer analysis than classic verification.



(a) The operating terminal, C_{term}.

(b) The turntable, C_{turn}.

(c) The normative environment, E

Fig. 1. The Therac-25 is modeled as $C_{T25} = C_{term}||C_{beam}||C_{turn}$. C_{beam} is in Fig. 7b.

System. We model the Therac-25 as the composition of the following three finite-state machines: (1) C_{term}, a computer terminal that nurses use to operate the Therac-25, (2) C_{beam}, a beam-emitter that fires a radiation treatment beam in either *X-ray* or *electron* mode, and (3) C_{turn}, a turntable that rotates between two hardware components called the *flattener* and the *spreader*. Formally, we define the Therac-25 as the composition all three machines: $C_{T25} = C_{term}||C_{beam}||C_{turn}$. We show the terminal and turntable in Figs. 1a and 1b respectively. We show the beam in Sect. 6.2 (Fig. 7b), where we present a case study on the Therac-25.

Environment. Nurses operate the Therac-25 by typing at a keyboard connected to a terminal. A nurse begins by choosing a beam mode by typing either an "x" for X-ray or an "e" for electron mode. The nurse then hits the "enter" key and waits for the terminal to display "beam ready" before finally pressing the "b" key to fire the beam. This workflow defines the operating environment which we call E, shown in Fig. 1c.

Safety property. Since the X-ray beams contain a high concentration of radiation, it is imperative that the flattener is in place when the machine fires an X-ray. We capture this key safety property in the following LTL [36] formula:

$$\mathbf{G}(\text{XFIRED} \rightarrow \text{FLATMODE})$$

In this formula, XFIRED is a predicate that is true if an X-ray beam was just fired, while FLATMODE is a predicate that is true when the turn table is in flattener mode. We refer to this safety property as P_{xflat} in this example.

Safety Analyses. Robustness opens our safety analysis beyond classic verification. We discuss several analysis options below.

(1) Standard Verification: We can check that the Therac-25 is safe within the operating environment, that is, $E||C_{T25} \models P_{xflat}$. Standard model checking techniques [2] show that the Therac-25 is indeed safe with respect to E.

(2) Robustness Calculation: Given that the Therac-25 is safe with respect to E, we can calculate its robustness Δ. This calculation identifies the set of safe environmental envelopes of the Therac-25. Importantly, these envelopes reveal the environmental deviations that the Therac-25 can safely handle. For example, in Sect. 6.2, we show that the Therac-25 is robust against the environmental deviations in Fig. 8 in which a nurse repeatedly hits "enter" or the "up" arrow key after choosing a beam mode.

(3) Controller Comparison: Holding the environment E and the property P_{xflat} constant, we can compare the robustness of the Therac-25 against other models. In Sect. 6.2, we introduce the Therac-20 (C_{T20}) and compare the robustness between C_{T25} and C_{T20}. Although both machines are safe with respect to the normative environment, we will find that C_{T25} is strictly less robust than C_{T20}. We will show how contrasting the robustness between the two machines exposes a critical software bug in the Therac-25. Furthermore, we will show that fixing the bug in the Therac-25 causes its robustness to be equivalent to the Therac-20.

(4) Property Comparison: Holding the environment E and the machine C_{T25} constant, we can compare the machine's robustness with respect to P_{xflat} and a second safety property. For example, we could consider a new safety property P' that strengthens P_{xflat} by additionally enforcing the spreader to be in place when a beam is fired in electron mode. The property P' might be of interest to avoid an *underdose*, a situation that might result from the flattener being in place when an electron beam is fired. Because P' is stronger than P_{xflat}, a designer may be interested to compare the robustness between the properties to understand which environmental deviations maintain P_{xflat}, but violate P'.

3 Modeling Formalism

This section describes the underlying formalism used to model the environment, controlled systems, and the properties enforced by them.

Labeled Transition Systems. Given a finite set A, the usual notations $|A|$ and A^* denote the cardinality of A and the set of all finite sequences over A respectively. In this work, we use finite labeled transition systems to model the behavior of the environment, the controller, and the property.

Definition 1. *A labeled transition system (LTS) E is a tuple $\langle Q_E, Act_E, R_E, q_{0,E}\rangle$, where Q_E is a finite set of states, Act_E is a finite set of actions, $R_E \subseteq Q_E \times Act_E \times Q_E$ is the transition relation of E, and $q_{0,E} \in Q_E$ is the initial state.*

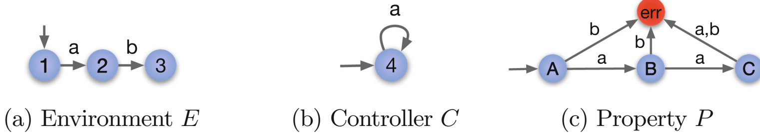
LTS E is said to be deterministic if for any $(q, a, q'), (q, a, q'') \in R_E$, then $q' = q''$; otherwise it is nondeterministic. We extend the transition relation R_E to finite sequences of actions as ${R_E}^* \subseteq Q_E \times {Act_E}^* \times Q_E$ in the usual manner. A *trace* of E is a finite sequence of actions $a_0 \dots a_n$ of E complying with the transition in ${R_E}^*$, i.e., $(q_{0,E}, a_0 \dots a_n, q) \in {R_E}^*$ for some $q \in Q_E$. The set of all traces in E is denoted by $beh(E)$.

Given LTSs E_1 and E_2, the parallel composition $||$ defines standard synchronization of E_1 and E_2 [2,7]. The composed LTS $E_1||E_2 = \langle Q_{E_1} \times Q_{E_2}, Act_{E_1} \cup Act_{E_2}, R_{E_1||E_2}, (q_{0,E_1}, q_{0,E_2})\rangle$ synchronizes over the common actions between E_1 and E_2 and interleaves the remaining actions. Lastly, given LTSs E_1 and E_2, we say that E_1 is a subset of E_2, denoted $E_1 \subseteq E_2$, if $Q_{E_1} \subseteq Q_{E_2}$, $Act_{E_1} = Act_{E_2}$, $R_{E_1} \subseteq R_{E_2}$, and $q_{0,E_1} = q_{0,E_2}$.

Control Strategy. Let an LTS E represent the environmental model to be controlled. A control strategy, or simply *controller*, for E is a function that maps a finite sequence of actions to a set of actions, i.e., $C : {Act_E}^* \to 2^{Act_E}$. A *controlled trace* of E is a trace of E, $a_0 \dots a_n \in beh(E)$, such that $a_i \in C(a_0 \dots a_{i-1})$ for any $i \leq n$. The set of all controlled runs, denoted by $beh(E/C)$, defines the closed-loop system of C controlling E. For convenience, this closed-loop system is denoted by E/C. In this work, we assume that controller C has finite memory and it can be represented by a deterministic LTS. With an abuse of notation, the LTS controller representation is also denoted by C. For convenience, we define controller $C = \langle Q_C, Act_C, R_C, q_{0,C}\rangle$ to have the same actions as in E, i.e., $Act_C = Act_E$. In this manner, the closed-loop system E/C can be represented by the composition of environment E and controller C: $E/C = E||C$.

Remark 1. We assume that all elements of the set of actions Act_E are "controllable" actions, that can be acted upon by a controller. However, the nondeterministic transition relation of E can be used to model uncontrollable actions of the environment. After an action a is selected by the controller at state q, the environment decides which state the system will be in, similarly to two-player games [15].

Safety Property. In this work, we consider a class of regular linear-time properties called safety properties over an environment E [2]. A safety property P is represented by a deterministic LTS P that defines the set of accepted behaviors. Usually, the LTS P encodes both the traces that satisfy P and those that violate it by including a sink error state. Formally, any trace that reaches the error state $err \in Q_P$ violates the safety property. An LTS E satisfies property P, denoted by $E \models P$, whenever the traces in $beh(E)$ do not reach the error state in P. In this manner, we can test if $E \models P$ by composing $E||P$ and investigating if the err is reached.



(a) Environment E (b) Controller C (c) Property P

Fig. 2. LTSs for the running example

Example 1. We describe a simple example that we use as a running example throughout the paper. Figure 2 depicts the environment E, controller C, and property P considered in this example. The environment E defines that action a is immediately followed by action b. Although controller C in Fig. 2b only shows action a, we assume that $Act_C = \{a, b\}$. In this manner, C only allows action a to occur. Lastly, property P defines that action a should happen at most two times while action b should never happen. It follows that $E/C \models P$ since the controller disables action b and the environment only executes one instance of action a.

4 Robustness Against Environmental Deviations

4.1 Deviations

A deviation is a set of transitions $d \subseteq (Q_E \times Act_E \times Q_E)$ A *deviated system* is defined by augmenting the transitions of environment E with a deviation set:

Definition 2. *Given an LTS* $E = \langle Q_E, Act_E, R_E, q_{0,E}\rangle$ *and a deviation* $d \subseteq Q_E \times Act_E \times Q_E$. *We define the* deviated system E_d *as* $E_d := \langle Q_E, Act_E, R_E \cup d, q_{0,E}\rangle$.

A controller C that guarantees property P for environment E, i.e., $E/C \models P$, might violate this property for the deviated environment E_d, i.e., $E_d/C \not\models P$.

Definition 3. *Controller* C *is a* robust controller *with respect to environment* E, *deviation* d, *and property* P *if* $E_d/C \models P$. *Deviation* d *is a* robust deviation *with respect to* E, C, *and* P *if* C *is a robust controller with respect to* E, d, *and* P.

Remark 2. In this paper, we are only interested in ensuring safety properties over the controlled system. For this reason, it is sufficient to only consider adding new transitions to the environment. If a controlled system is safe, then deleting transitions from the environment does not violate the safety property.

4.2 Comparing Deviations

Each deviation set affects the environment in different ways. To reason about the effects of each deviation set, we compare them using a partial order relation over $Q_E \times Act_E \times Q_E$. For deviations d_1 and d_2 such that $d_1 \subseteq d_2$, d_2 deviates

LTS E more than d_1 since $beh(E_{d_1}) \subseteq beh(E_{d_2})$. For this reason, we select the relation $\subseteq$ over $Q_E \times Act_E \times Q_E$ to be the partial order to compare different deviation sets.

Definition 4. *Given E and deviations d_1, d_2, d_1 is* at least as powerful *as d_2 if $d_2 \subseteq d_1$.*

4.3 Robustness

Intuitively, robustness is defined as the set of all possible robust deviations d with respect to the environment E, controller C, and safety property P_{saf}. Additionally, we introduce an environmental constraint, P_{env}, to capture domain knowledge about the system under analysis. P_{env} will filter environment deviations that might not be physically feasible or of interest to analyze. This constraint is captured as a safety property over E, i.e., $E \models P_{env}$ states that the environment satisfies the constraint. Formally, our robustness notions is defined as follows:

Definition 5. *Let environment E, controller C, property P_{saf} such that $E/C \models P_{saf}$, and environment constraint P_{env} such that $E \models P_{env}$ be given. The robustness of controller C with respect to E, P_{saf}, and P_{env}, denoted by $\Delta(E, C, P_{saf}, P_{env})$, is a set of robust deviations $\Delta \subseteq 2^{Q_E \times Act_E \times Q_E}$. Δ is defined to be the (unique) set of robust deviations satisfying the following conditions:*

1. $\forall d \in \Delta.\ E_d/C \models P_{saf}$ [d is robust];
2. $\forall d \subseteq Q_E \times Act_E \times Q_E . E_d/C \models P_{saf} \wedge E_d \models P_{env} \Rightarrow \exists d' \in \Delta . d \subseteq d'$ [d is represented];
3. $\forall d, d' \in \Delta.\ d \neq d' \Rightarrow d \not\subseteq d'$ [unique representation].
4. $\forall d \in \Delta.\ E_d \models P_{env}$ [d is feasible].

When E, C, P_{saf}, and P_{env} are clear from context, we simply write Δ. The set Δ is also denoted as the safety envelope of C with respect to E, P_{saf}, and P_{env}.

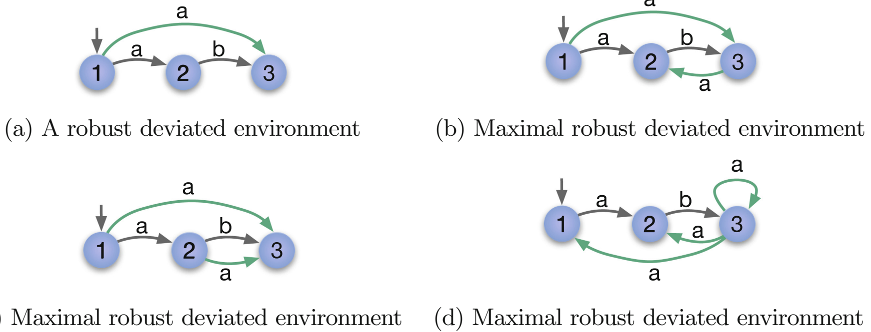
Intuitively, the set Δ defines an upper bound on the possible deviations from E that controller C is robust against. In other words, Δ captures the envelopes for which controller C remains safe.

If a designer does not have domain knowledge about the system, then P_{env} can be set to not constrain the environment, i.e., $P_{env} = Act_E^*$. After computing Δ without environmental constraints, a designer can obtain important information about the system and the environment. In the next analysis iteration, this knowledge can be transformed into environmental constraints to enhance the robustness analysis, i.e., $P_{env} \subseteq Act_E^*$.

By definition, Δ is always non-empty since $d = \emptyset$ is always robust. Moreover, due to conditions 2 and 3, only maximal robust deviations are included in Δ. We show that there is a unique set of deviations that satisfies the conditions of Def. 5. The proof of this lemma is available at [27], pg. 23.

Lemma 1. *Given LTS E, controller C, safety property P_{saf}, and environment property P_{env}, there is a unique Δ that satisfies the conditions in Def. 5.*

Example 2. Back to our running example, we investigate robust deviations and Δ. For simplicity, we do not impose any environment constraint, i.e., $P_{env} = Act_E^*$. Figure 3 shows four robust deviations for our running example, where transitions in green are deviations added to the environment. All robust deviations allow at most two transitions with action a, which is the maximum number allowed by the property. In this example, Δ has three robust deviations that are represented in Figs. 3b–3d. Since the robust deviation shown in Fig. 3a is a subset of both deviations in Fig. 3b and Fig. 3c, it is not included in Δ.



(a) A robust deviated environment

(b) Maximal robust deviated environment

(c) Maximal robust deviated environment

(d) Maximal robust deviated environment

Fig. 3. Robust deviated environments. Robust transitions $Q_E \times \{b\} \times Q_E$ are omitted.

4.4 Problem Statement

Although Def. 5 has formally introduced our notion of robustness, it does not show how to compute robustness. Therefore, we investigate the problem of computing the set Δ.

Problem 1. Given E, C, P_{saf}, and P_{env} as in Def. 5, compute Δ.

4.5 Comparing Robustness

Our robustness definition also allows us to compare the robustness between different controllers as well as different safety properties.

Comparing Controllers. Holding the environment and safety property constant, we can compare the robustness of the controllers.

Definition 6. *Given an environment E, controllers C_1 and C_2, safety property P_{saf}, and environment constraint P_{env}, controller C_1 is at least as robust as C_2 if and only if for all $d_2 \in \Delta(E, C_2, P_{saf}, P_{env})$ there exists $d_1 \in \Delta(E, C_1, P_{saf}, P_{env})$ such that $d_2 \subseteq d_1$. Equality and strictly less/more robust are defined in the usual manner using $\subseteq$.*

Comparing Safety Properties. Holding the environment and controller constant, we can compare the robustness between safety properties.

Definition 7. *Given an environment E, controllers C, safety properties $P_{saf,1}$ and $P_{saf,2}$, and environment constraint P_{env}, controller C is at least as robust with respect to $P_{saf,1}$ than with respect to $P_{saf,2}$ if and only if for all $d_2 \in \Delta(E, C, P_{saf,2}, P_{env})$, there exists $d_1 \in \Delta(E, C, P_{saf,1}, P_{env})$ such that $d_2 \subseteq d_1$.*

5 Computing Robustness

This section presents two manners of solving Problem 1. One is a brute-force algorithm whereas the second uses control techniques to obtain the solution. Usually when dealing with regular safety properties, one transforms the safety property into an invariance property. This transformation is simply obtained by composing the environment with the safety property; then, an invariance property equivalent to the safety is defined over this composed system [2]. In this composed system, an invariance property is simply defined by a set of safe states. Unfortunately, computing robustness for safety properties does not directly reduce to computing robustness for invariance properties.

When transforming a safety property P_{saf} to an invariance property, we compose the environment and the safety property. Let us assume that there are no environmental constraints. In our scenario, the invariance property P_{inv} is defined based on the composed system $E||C||P_{saf}$, i.e., $P_{inv} \subseteq Q_{E||C||P_{saf}}$. The composed system P_{inv} introduces memory to the environment to differentiate when the safety property is violated or not. This memory addition prevents a simple reduction between invariance and safety properties since robustness is defined with respect to the environment. Robustness defines new transitions in E whereas computing robustness with respect to P_{inv} defines new transitions in $E||C||P_{saf}$. For this reason, we cannot simply reduce the problem of computing Δ with respect to safety properties to the problem of computing Δ with respect to an invariance property.

5.1 Brute-Force Algorithm

One way of solving Problem 1 is via a brute-force algorithm. Intuitively, this algorithm is broken into two parts: (i) finding the set of robust deviations that satisfy the environmental constraint, and (ii) identifying the maximal ones within this set. In part (i), we verify $E_d||C \models P_{saf}$ and $E_d \models P_{env}$ for all deviations $d \subseteq (Q_E \times Act_E \times Q_E) \setminus R_E$, which can be solved using standard model checking techniques [2]. Since this algorithm checks if every deviation set is robust or not, it is clear that it computes Δ.

5.2 Controlling the Deviations Without Environmental Constraints

Due to the lack of scalability of the brute-force algorithm, we search for more efficient ways to compute Δ. For readability purposes, we start by describing our

algorithm in detail assuming no environmental constraints, i.e., unconstrained environment $P_{env} = Act_E^*$. In the next section, we show how to use this algorithm to completely solve Problem 1, i.e., for a possibly constrained environment $P_{env} \subseteq Act_E^*$.

Overview of the Control Algorithm. At a high level, we transform the problem of computing Δ to a problem of controlling environmental transitions to avoid safety violations. Intuitively, we control deviations to force them to be robust, i.e., we take the viewpoint that we can control transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$. Different ways of controlling transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$ provide different robust deviations.

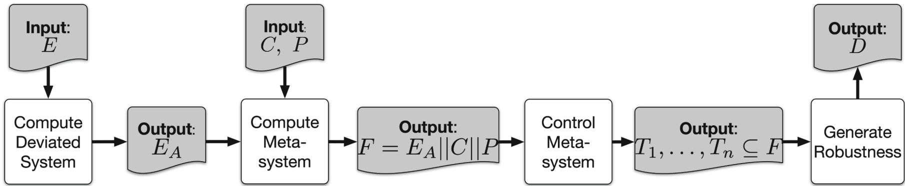


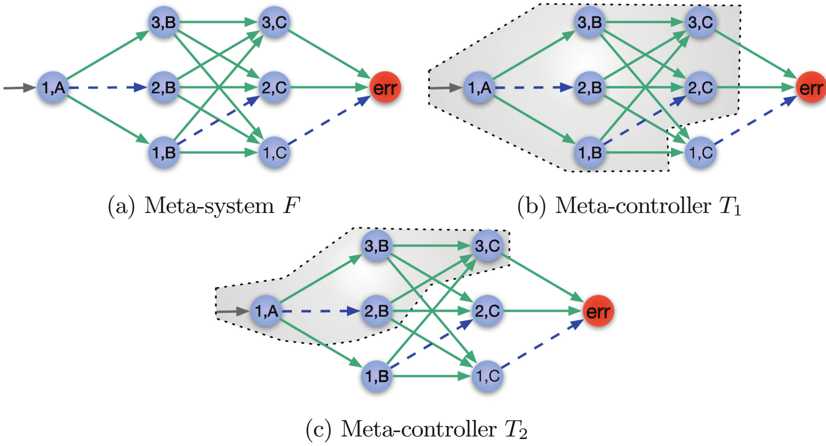
Fig. 4. Overview of our approach to compute robustness for the unconstrained environment. The inputs are the LTSs of environment E, controller C, and property P_{saf}. The set A is the set of all environment transitions, $A = Q_E \times Act_E \times Q_E$. The LTSs $T_1, \ldots, T_n \subseteq F$ represent controlled meta-systems.

Figure 4 provides an overview of our approach. First, we define LTS E_A to be the deviated system with all possible transitions, i.e., $A = Q_E \times Act_E \times Q_E$. The deviated system E_A is the maximally deviated environment since it encompasses every possible deviated system E_d for $d \subseteq Q_E \times Act_E \times Q_E$.

Next, we compose the deviated environment E_A with controller C and property P_{saf}, to create a "meta-system" F. This meta-system provides information about how the deviated environment E_A under the control of C can violate P_{saf}. Following this composition, we pose a control problem over the meta-system to prevent any violation of P_{saf}. There are multiple ways of controlling this composed system; in our approach, we obtain a finite number of controllers encoded as $T_i \subseteq F$. These different ways of controlling the meta-system provide different robust deviations from which we can extract Δ. To make our approach concrete, we describe each step in detail using our running example, shown in Fig. 2.

Constructing the Meta-system. The deviated environment $E_A = E_{Q_E \times Act_E \times Q_E}$ contains the behavior of any other deviated environment. Therefore, we define the meta-system to be the composition of deviated environment E_A, controller C, and property P_{saf}, i.e., $F = E_A||C||P_{saf}$. Figure 5a shows the meta-system F for our running example. Since C only has one state, we omit its state from the state names in Fig. 5a, i.e., states in Fig. 5a are defined as $(q_e, q_p) \in Q_E \times Q_{P_{saf}}$ instead of $(q_e, q_c, q_p) \in Q_E \times Q_C \times Q_{P_{saf}}$. All transitions in F are labeled a, omitted in Fig. 5a, since controller C only enables action a. We

also identify in F which transitions are derived from the environment (dashed blue) and which are derived from deviations (green). For simplicity, we define a single error state in F to capture every $(q_e, q_c, err) \in Q_E \times Q_C \times Q_{P_{saf}}$.



(a) Meta-system F (b) Meta-controller T_1

(c) Meta-controller T_2

Fig. 5. Meta-systems. All transitions have action a since C only enables action a (see Fig. 2b). Dashed blue transitions represent transitions that are feasible in R_E while solid green transitions represent the deviated transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$. The shaded area in Fig. 5b contains all safe states in the meta-system.

Controlling the Meta-system. Once the meta-system is constructed, we pose a meta-control problem over F to ensure that the meta-system avoids the error states, i.e., states $(q_e, q_c, err) \in Q_E \times Q_C \times Q_{P_{saf}}$. These error states represent safety violations in the closed-loop system. For instance, in Fig. 5a, if transition $(2, C) \rightarrow err$ occurs, then the closed-loop system violates P_{saf} since more than two actions a were executed. In this meta-control problem, a meta-controller can disable transitions in F that originated from deviations in E, i.e., transitions in $(Q_E \times Act_E \times Q_E) \setminus R_E$.

Problem 2. Given meta-system F, synthesize a meta-controller $T \subseteq F$ such that (1) for any $(q_e, q_c, q_p) \in Q_T$ then state $q_p \neq err$; and (2) for any $\big((q_e, q_c, q_p), a, (q_e', q_c', q_p')\big) \in R_F \setminus R_T$ such that $(q_e, q_c, q_p) \in Q_T$, it follows that $(q_e, a, q_e') \notin R_E$.

Problem 2 states that the meta-controller is a subset of the meta-system F. We want to maintain the same structure as in F since we need to enforce that the meta-controller does not disable any transition associated with R_E. Condition (1) in Problem 2 ensures that property P_{saf} is not violated. On the other hand, condition (2) guarantees that only transitions assigned to deviations are disabled.

Back to our example, the LTS T described by the shaded area in Fig. 5b demonstrates a possible meta-controller that satisfies Problem 2. Condition (1) is satisfied since the error state is not included in the shaded area. With respect to condition (2), only solid green transitions are disabled. Figure 5c shows another meta-controller.

To solve Problem 2, one can solve a safety game over F using fixed-point computation [15,25]. Due to space limitations, we point the reader to [27], pg. .23 for the solution to this safety game.

Extracting Robust Deviations. Each meta-controller that solves Problem 2 relates to a robust deviation. Intuitively, a meta-controller disables deviations that would violate P_{saf}. For instance, the meta-controller T_1 shown in Fig. 5b disables transition $(3, B) \rightarrow (1, C)$, which relates to disabling transition $3 \xrightarrow{a} 1$ in the environment. Figure 3a depicts the deviated environment related to meta-controller T_1. Similarly, Fig. 3b shows the deviated environment associated with meta-controller T_2.

To extract a robust deviation from a meta-controller, we have to (1) identify the transitions that the meta-controller has disabled; and (2) project the disabled transitions to transitions $Q_E \times Act_E \times Q_E$. Since a meta-controller is a subset of the meta-system, the disabled transitions are obtained by comparing F and T. Intuitively, the disabled transitions are those that escape the shaded area in Fig. 5.

$$Disabled := \{(q,\ a,\ q') \in R_F \mid q \in Q_T\ \wedge\ (q,\ a,\ q') \notin R_T\} \tag{1}$$

For instance, in the case of meta-controller T_1, the transition $((1, B), a, (1, C))$ belongs to the *Disabled* set. Next, based on the disabled transitions, we project them to transitions in $Q_E \times Act_E \times Q_E$, i.e., transitions in the environment.

$$del := \{(q_e,\ a,\ q'_e) \in Q_E \times Act_E \times Q_E \mid ((q_e, q_c, q_p), a, (q'_e, q'_c, q'_p)) \in Disabled\} \tag{2}$$

Transitions in *del* are the transitions to be deleted from $Q_E \times Act_E \times Q_E$ such that $(Q_E \times Act_E \times Q_E) \setminus del$ is a robust deviation set. If transitions in *del* are included in a deviation set, they can cause a violation of property P_{saf}. In the case of T_1, the transition $(1, a, 1)$ is included in *del*. If we maintain, for instance, transition $1 \xrightarrow{a} 1$ as part of a deviation set d, then the closed-loop E_d/C violates the property P_{saf} since the path $(1, A) \rightarrow (1, B) \rightarrow (1, C) \rightarrow err$ would be feasible in the meta-controller.

Computing Robustness $\boldsymbol{\Delta}$. Problem 2 searches for meta-controllers that guarantee the satisfaction of property P_{saf}. To compute Δ, we need to obtain a finite number of meta-controllers. Algorithm 1 formalizes our description in Fig. 4. It takes as input the environment E, the controller C, a deviation set d, and a safety property P. From the algorithm overview description in Fig. 2, we have that for the unconstrained environment $d = A = Q_E \times Act_E \times Q_E$ and $P = P_{saf}$.

In Algorithm 1, line 4 computes the largest possible set of invariant states that avoid the error state, i.e., $Inv(Q_F \setminus Err)$ solves the safety game as shown

Algorithm 1. COMPUTE-ROBUSTNESS

Input: LTSs E, C, P and deviation d
Output: Set of deviations D
1: $D \leftarrow \emptyset$
2: $F \leftarrow E_d||C||P$
3: $Err \leftarrow \{(q_e, q_c, q_p) \in Q_F \mid q_p = err\}$
4: $W \leftarrow Inv(Q_F \setminus Err)$
5: **for all** $S \in 2^W \setminus \{\emptyset\}$ **do**
6: $T \leftarrow$ META-CONTROLLER(S, F)
7: $del \leftarrow \{(q_e, a, q_e') \in d \mid \exists((q_e, q_c, q_p),\ a,\ (q_e', q_c', q_p')) \in R_F \setminus R_T$ s.t. $(q_e, q_c, q_p) \in Q_T\}$
8: $D \leftarrow D \cup \{d \setminus del\}$
9: **while** $\exists d_1, d_2 \in D$ s.t. $d_1 \subseteq d_2$ **do**
10: $D \leftarrow D \setminus \{d_1\}$
return D
11: **procedure** META-CONTROLLER(S, F)
12: $S \leftarrow Inv(S)$
13: **if** $q_{0,F} \notin S$ **then**
14: $T \leftarrow \emptyset$
15: **else**
16: $Q_T \leftarrow S$, $Act_T \leftarrow Act_F$, $q_{0,T} \leftarrow q_{0,F}$
17: $R_T \leftarrow \{(q, a, q') \in S \times Act_T \times S \mid (q, a, q') \in R_F\}$
return T

in [27], pg. 23. Based on this invariant set, each iteration in the loop (lines 5–8) computes a meta-controller (line 6) and stores its respective robust deviation (line 8). The meta-controller T is also computed by using the function Inv. The meta-controller solution ensures that $Q_T \subseteq S$. Line 7 computes environmental transitions that must be deleted in order to obtain a robust deviation. The computed robust deviations are stored in Δ. Lastly, the loop in lines 9–10 ensures that only maximal robust deviations are included in Δ.

In more detail, to solve Problem 2, we must guarantee that the meta-system F does not reach any states in $Err := \{(q_e, q_c, q_p) \in Q_F \mid q_p = err\}$. Formally, we compute the set $Inv(Q_F \setminus Err)$, which contains every state in F that does not reach a state in Err via a transition associated with R_E. Based on this invariant set, we can extract any meta-controller that remains within this set. Informally, the META-CONTROLLER(S, F) in line 11 of Algorithm 1 computes a meta-controller that remains within states in S. First, this procedure computes the invariant set of S, i.e., $Inv(S)$ with respect to meta-system F (line 12). In this manner, a meta-controller is defined by projecting the meta-system F to states and transitions in the set of state $Inv(S)$ (lines 16–17).

The following theorem shows that Δ computed via Algorithm 1 is equal to Δ as in Definition 5 when $P_{env} = Act_E^*$, i.e., Algorithm 1 *partially* solves Problem 1.

Theorem 1. *Given LTS E, controller C, and property P_{saf}, Algorithm 1 outputs Δ as in Definition 5 when $P_{env} = Act_E^*$.*

Proof. Sketch. In order to show that Theorem 1 holds, we provide two intermediate lemmas whose proofs are available at [27], pg. 24 (Lemma 2 and Lemma 3). The first lemma states that every meta-controller T produces a robust deviation. In this manner, we show that for every $d \in \Delta$, the deviation d is robust. The second lemma shows that for every maximal robust deviation $d \in \Delta$, there exists a meta-controller T associated with deviation d. Consequently, Algorithm 1 computes every possible maximal robust deviation.

Using Algorithm 1 to compute Δ for our running example, we obtain Δ that contains the three maximal robust deviations shown in Fig. 3. Lastly, we provide the computational complexity of Algorithm 1.

Theorem 2. *Algorithm 1 outputs* Δ *in* $O(2^{|Q_E||Q_C|(|Q_P|-1)})$.

Proof. It follows from the size of 2^W.

Although Algorithm 1 has exponential complexity, we empirically show in Sect. 6 that it scales better than the brute-force algorithm.

Heuristics to Exploit the Structure of *F*. In Algorithm 1, we compute robust deviations for every possible subset of the largest invariant state set, c.f., line 5. To improve the efficiency of Algorithm 1, we provide a sound and complete heuristic that identifies and skips redundant subsets of $2^W \setminus \emptyset$. The heuristic is based on the observation that sets of states that are not directly connected in F correspond to redundant deletion sets from $Q_E \times Act_E \times Q_E$. As such, the heuristic exploits the structure of F by performing a depth-first search over its state space, hence skipping disconnected groups of states. For instance, the heuristic will skip the subset $\{(1, A), (3, C)\}$ because $(1, A)$ and $(3, C)$ are not connected in F. This subset is redundant because its deletion set $del = \{((1, A), (1, B)), ((1, A), (2, B)), ((1, A), (3, B))\}$ is identical to the deletion set for the subset $\{(1, A)\}$ which is connected. In the worst-case scenario, our heuristic computes the power set of W, i.e., exactly as in line 5.

5.3 Controlling the Deviations with Environmental Constraints

When introducing environmental constraints, we must eliminate the robust deviations that violate these constraints as described in Definition 5. One might think that P_{env} and P_{saf} could be combined as a single safety property for which we then compute Δ. However, this approach does not work since P_{env} must be enforced only by the environment whereas P_{saf} is a property of the closed-loop system. Another approach is to verify if P_{env} is satisfied for each deviation obtained in the for-loop (lines 5–8) in Algorithm 1. Although this approach is feasible, in practice, we want to reduce the number of deviations, using P_{env}, before we compute the robust deviations. For this reason, we describe a sequential algorithm shown in Fig. 6. In this algorithm, Algorithm 1 is used multiple times in this constrained scenario instead of a single time as in the unconstrained scenario (Sect. 5.2).

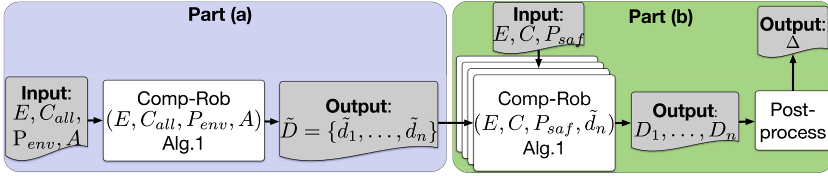


Fig. 6. Overview of our approach to compute robustness for constrained environments.

The algorithm to compute robustness for constrained environments can be broken into two parts: (a) computing all maximal environments $\tilde{d}_i$ that satisfy P_{env}; and (b) computing robust deviations for each deviated environment $E_{\tilde{d}_i}$ found in part (a). Computing the maximal environments that satisfy P_{env} reduces to computing maximal deviations of E with respect to a controller that allows every environment action, C_{all}. Formally, the behavior of C_{all} does not restrain E, $beh(C_{all}) = Act_E^*$; and it can be described by a one-state LTS. Therefore, the output of part (a) is the set of maximal deviations $\tilde{d}_i$ with respect to E, C_{all}, and P_{env}, denoted as maximal environment deviations. Each maximal deviated environment $E_{\tilde{d}_i}$ satisfy the P_{env}.

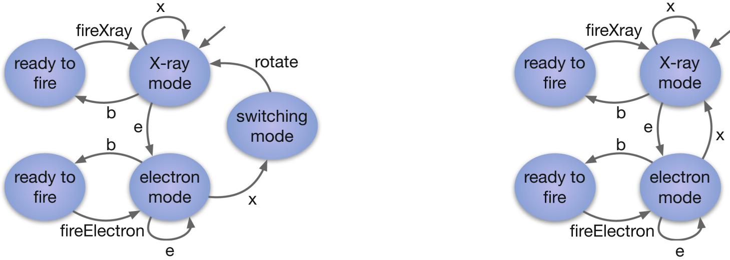
Once we have obtained all maximal environment deviations that satisfy P_{env}, we focus on finding the maximal robust deviations with respect to C and P_{saf}. In other words, we run Algorithm 1 for each maximal deviated environment $E_{\tilde{d}_i}$ together with C and P_{saf}. Since d is a subset of $\tilde{d}_i$, we have that the perturbed system E_d satisfies P_{env}.

Each maximal deviated environment $E_{\tilde{d}_i}$ generates a set of maximal robust deviations D_i with respect to C and P_{saf}. The final step is combining these maximal robust deviations with respect to each $\tilde{d}_i$. Since they are maximal with respect to $\tilde{d}_i$, there could be deviations that are not maximal as defined by Definition 5. The post-processing step combines the deviations and eliminates any non-maximal deviations; and it outputs Δ as in Definition 5. The correctness of this algorithm follows from Theorem 1.

6 Case Studies

6.1 Implementation

We have implemented a prototype tool for computing robustness [28]. The tool accepts a model of an environment, a controller, and a safety property–as well as an optional list of environmental constraints–and outputs Δ. The tool has support for comparing the robustness of two controllers as well as the robustness of a controller with respect to two separate safety properties. Currently, the environment, controller, safety property, and environmental constraints must be encoded in Finite State Process (FSP) notation [23] but this is not a fundamental limitation.



(a) The beam C'_{beam} *with* hardware interlocks used in the Therac-20.

(b) The beam C_{beam} *without* hardware interlocks used in the Therac-25.

Fig. 7. The beam components of the two Therac machines. The hardware interlocks cause C'_{beam} to have a fifth state "switching mode" that will only switch to X-ray mode after the flattener rotates into place.

We wrote the tool in the Kotlin programming language. Our tool includes an implementation of the brute-force algorithm from Sect. 5.1, as well as an implementation of Algorithm 1 and Algorithm 1 with heuristics. In the following case studies, we leverage the tool to calculate and compare the robustness of several systems. We summarize our performance results for each case study in Sect. 6.6.

6.2 Therac-25

Background. In Sect. 2, we introduced the Therac-25 radiation therapy machine. In this section, we present a case study in which we compare the robustness of the Therac-25 to that of its predecessor, the Therac-20. We begin by showing that the Therac-20 is strictly more robust than the Therac-25. We then use this information to identify and fix a critical safety bug in the Therac-25 model.

Therac-20. The Therac-20 is a radiation therapy machine that was designed before the Therac-25. Unlike the Therac-25, the Therac-20 was not known for causing accidents that led to injuries and death. A key difference between the two machines is that the Therac-20 includes hardware *interlocks* in its beam component (Fig. 7a), while the Therac-25 does not (Fig. 7b). The purpose of the hardware interlocks is to provide a layer of security at the hardware level for upholding P_{xflat}. In our model, the interlocks work by ensuring that the flattener is completely rotated into place before allowing an operator to fire an X-ray beam. Unfortunately, hardware interlocks were considered expensive so they were omitted from the design of the later Therac-25 model. In the following section, we compare the robustness between the two Therac machines with respect to the normative environment E and the key safety property P_{xflat}.

Comparing Controllers. Using standard model checking techniques [2], we can confirm that both the Therac-20 and the Therac-25 are safe with respect

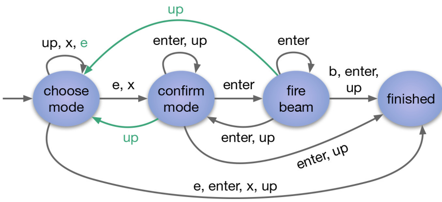


Fig. 8. Visual robustness comparison between the two Therac machines. Both machines are robust against gray transitions, but only the Therac-20 is robust against green transitions. (Color figure online)

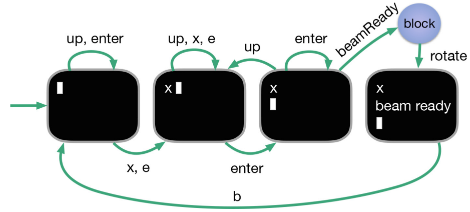


Fig. 9. Software fix that eliminates the race condition in the Therac-25.

to E and P_{xflat}. Historically, however, the Therac-20 is known to be safer than the Therac-25. Therefore, we improve our safety analysis by also comparing the robustness between the two machines with respect to E, P_{xlfat}, and an environmental constraint P_{env}. P_{env}, shown in [27], pg. 26, Fig. 11, restricts the environment to firing the beam at most once.

Our tool reports that the Therac-20 is strictly more robust than the Therac-25. To understand this result, we can examine the difference between the robustness for each machine. We show this difference visually by presenting one maximal robust deviation from each machine in Fig. 8. This figure shows that the Therac-20 is robust against the scenario in which the operator 1) types "e" to select electron beam mode, 2) optionally types "enter", 3) presses the "up" arrow key, and finally 4) types "x" to switch the beam into X-ray mode. The Therac-25, however, is not robust against this scenario. We see this in Fig. 8 because the series of actions must pass through at least one green arrow, where a green arrow indicates a transition that the Therac-25 is not robust against. In fact, the Therac-25 does not have *any* maximal robust deviations that allow this scenario.

The Therac-25's lack of robustness to the scenario above represents a race condition that occurs after the operator switches into X-ray mode from electron mode. In this scenario, if the operator types "enter" and fires the X-ray beam before the flattener rotates into place, the beam will fire an unflattened X-ray at the patient. This critical bug was responsible for real-world radiation overdoses, several of which resulted in death [18].

Fixing the Software Bug. In the previous section, we identified a critical software bug in the Therac-25. Our goal in the current section is to fix this bug entirely in the terminal software, thus avoiding an expensive hardware solution.

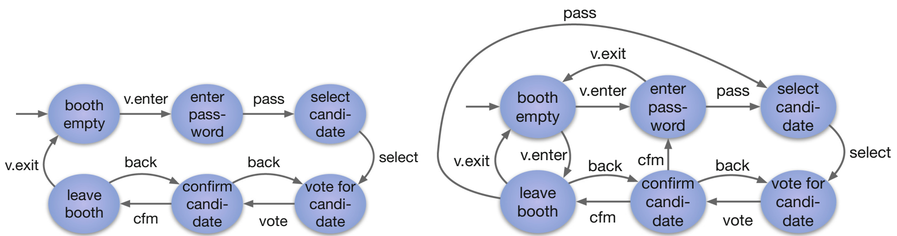
In Fig. 7a, we see that the hardware interlocks prevent a race condition by blocking the operator from typing a "b" until the flattener is rotated into place. Thus we can fix the race condition in software by altering the terminal to block the operator from typing a "b" until the flattener is rotated into place. We implement this fix by redesigning the terminal to block all key strokes from

the instant it issues a "beam ready" message until the turntable rotates into place, as shown in Fig. 9. Finally, we use our tool to evaluate the robustness of the fix. The tool reports that the fixed Therac-25 design is strictly more robust than the original, and equally robust to the Therac-20.

6.3 Voting

Background. In this section, we consider a case study of an electronic voting machine, introduced in [46]. In this case study, we model the voting machine, a voter, and a corrupt election official who attempts to "flip" the voter's choice. We define the voting machine as the composition of a voting booth and a user interface, shown at [27], pg. 26 in Fig. 12a and Fig. 12b respectively.

In the normative environment–shown in Fig. 10a–the voter enters the booth, enters their password, selects a candidate, clicks the vote button, and finally confirms the choice. Unfortunately, some voters may inadvertently skip the confirmation step and leave the booth early. This deviation from the normative behavior presents an opportunity for the election official to "flip" the intended vote: after the voter leaves the booth, the corrupt official can enter the booth, press "back" and change the vote to their liking. This scenario represents an actual election fraud that took place in the US [38].



(a) Normative environment for the voting machine.

(b) The voting machine's robustness is identical with respect to P_{all} and P_{cfm}.

Fig. 10. Models for the voting machine example. In the figures above, the prefix "v" represents actions by the voter.

Comparing Properties. In this case study, we will consider two safety properties, P_{all} and P_{cfm}, both of which imply the absence of vote flipping. P_{all} requires that the election official cannot at any point select, vote, or confirm a candidate. P_{cfm} is weaker, only requiring that the election official cannot at any point confirm a candidate selection.

Using our tool for comparison, we see that the voting machine is equally robust with respect to each property. However, this result is surprising because P_{cfm} is weaker than P_{all}. To understand this result, we examine Fig. 10b where we present the sole maximal robust deviation for each property. In this figure, it is clear that the voting machine is not robust against any deviation in which the voter enters their password and then exits the booth without confirming their vote. The key insight is that, when an election official has the ability to confirm, it *implies* that the official can also select and vote. Therefore, we desire a voting machine without this implication because it will reduce the number of points of failure. For example, we could redesign the voting machine to require a password as part of the confirmation step. In lieu of this insight, a designer could choose to specify a margin of safety into the machine's specification by requiring that it is strictly more robust against P_{cfm} than P_{all}.

6.4 Oyster

Background. The Oyster example was introduced in [41], in which the authors modeled the Oyster card that is used the public transportation system in the United Kingdom. In our model, the controller consists of an *entry gate* and an *exit gate*, where the card holder taps the Oyster card at the start and end of their journey respectively. The environment models the actions of a card holder; in the normative environment, a card holder chooses to tap with either their Oyster card or a credit card, and taps in and out with the chosen card. The key safety property is avoiding an *incomplete journey*, in which a card holder taps in with one card and taps out with a different card.

Calculating Robustness. An incomplete journey is avoided under the normative environment. We calculate the robustness of the system under the two environmental constraints 1) Oyster cards and credit cards give the correct information to the gates and 2) the gates operate correctly and calculate the correct fare when a card is tapped in and out. Unfortunately, the system is not robust to *any* deviations.

6.5 PCA Pump

Background. In this section, we model a patient-controlled analgesia (PCA) pump, originally introduced in [5]. A PCA pump is a medical device that dispenses pain medicine to a patient, offering them partial control over the dose rate. A nurse uses the device interface to program the volume per dosage, as well as a minimum and maximum dose rate to protect the patient from an overdose. The pump includes batteries to power the device in case it is unplugged (e.g., by mistake by the nurse or patient), yet the power may fail if the device runs out of battery. In this case, the device cannot monitor the dosage amount or frequency, which may cause an overdose. Therefore, we define the key safety property P_{pfail}

which requires the PCA pump to abstain from administering medicine after a power failure.

In the normative environment, the nurse operates the pump using the following three step workflow: 1) plug in the pump and turn it on, 2) program the desired dosage parameters into the pump and administer the treatment, and 3) turn off the device and unplug it. The nurse begins with step (1) and ends with step (3), but may omit or repeat step (2) as many times as needed. A diagram of the normative environment is available at [27], pg. 26, Fig. 13. Crucially, the pump is safe with respect to this environment and P_{pfail} because the workflow assumes that the pump is never unplugged in step (2).

Calculating Robustness. We use our tool to calculate the robustness of the pump with respect to the normative environment, P_{pfail}, and an environmental constraint P_{env}. In this case study, P_{env} restricts the environment to actions that are allowed by the pump's interface. A diagram of the sole maximal robust deviation is available at [27], pg. 27, Fig. 14. The tool reports that the pump is robust against four actions, three of which allow the operator to change settings before administering the treatment, and the fourth allows the operator to turn off the device prematurely after programming the dosage parameters. Unfortunately, the pump is not robust against any deviations in which it is unexpectedly unplugged. This poses a key weakness in the pump that the designers may wish to improve upon.

6.6 Results and Discussion

We have run our tool on the examples and case studies above, and we present our results in Table 1. All tests were run on a Mac Book Pro with an M1 Pro chip and 32GB of RAM. In the table, $|Act|$ is the union of Act_E, Act_C, $Act_{P_{saf}}$ and $Act_{P_{env}}$, $|d_{max}|$ is the size of the largest deviation in Δ, and $|W_{P_{env}}|$ is the size of the winning set for each maximal deviation $\tilde{d_i}$ (separated by a comma); NA indicates the absence of an environmental constraint. Furthermore, "Wall Heur" denotes the wall time for running Algorithm 1 with the heuristic, while "Wall Plain" denotes the wall time for running Algorithm 1, and "TO" indicates a time-out after five minutes.

Our results demonstrate that calculating robustness is tractable across several different case studies. In particular, our tool's performance on the larger PCA pump case study shows promising results in terms of scalability. Furthermore, we have shown that Δ is useful as a means for both analysis and comparison of controllers. For example, in the Therac-25 case study, robustness provided a richer analysis than classic verification that helped us discover–and ultimately fix–a critical race condition. Finally, we have also demonstrated in the voting machine case study that robustness provides a means for comparing two properties with respect to a controller and an environment.

Table 1. Summary of results from running our tool.

Example	$\lvert Act \rvert$	$\lvert Q_E \rvert$	$\lvert Q_C \rvert$	$\lvert Q_P \rvert$	$\lvert W \rvert$	$\lvert W_{P_{env}} \rvert$	$\lvert \Delta \rvert$	$\lvert d_{max} \rvert$	Wall Heur	Wall Plain
Running Example	2	4	2	4	6	NA	3	13	0.433 s	0.431 sec
Therac-25 w/bug	9	5	21	5	62	28,30,31,37	4	21	4.921 sec	TO
Therac-25 w/fix	9	5	19	5	72	18,20,23,25	4	26	0.852 sec	TO
Therac-20	9	5	11	5	40	17,19,21,23	4	26	0.626 sec	TO
Voting wrt. P_{cfm}	9	7	13	3	66	7	1	12	0.469 sec	TO
Voting wrt. P_{all}	9	7	13	3	66	7	1	12	0.426 sec	TO
Oyster	8	4	17	2	15	8	1	4	0.472 sec	TO
PCA Pump	21	11	105	4	1396	34	1	15	1.922 sec	TO

7 Related Work

Quantitative robustness notions for discrete transition systems have been investigated in several works [3,4,8,16,24,32,40,42]. We capture robustness qualitatively, which avoids the need for external cost functions over the discrete transition systems. The problem of synthesizing robust controllers against deviated environments given by a designer is investigated in [45]. Since [45] focuses on synthesizing robust controllers, their framework does not address the analysis of robustness. Moreover, robust controllers are measured via a rank function (quantitatively). Robust linear temporal logic (rLTL) extends the binary view of LTL to a 5-valued semantics to capture different levels of property satisfaction [43]. This work is tangent to ours as it focuses on specifying robustness.

In [17,49], the authors define robustness as a set of environmental behaviors for which a software system can guarantee safety. Defining robustness in the semantic domain–i.e. in terms of behaviors–implicitly describes safe environmental deviations. Our notion of robustness captures safe environmental deviations explicitly in terms of transitions, which offer both syntactic (transitions) and semantic (implied behaviors) information. Transition-based robustness also allows us to capture the safe environmental envelopes of a system; it is not clear how one might efficiently capture this information with only behaviors.

In [29], the authors define robustness also based on additional transitions to the environment. Their definition of robustness compares the perturbed controlled behavior, i.e., $beh(E_d|f)$, instead of directly comparing the additional transitions. In this manner, the partial order used to define robustness in [29] is different from our notion of robustness. Moreover, only an efficient algorithm for invariance properties is presented. Extending the work in [29], the authors explore the relationship between controller robustness and permissiveness for invariance properties [30].

Robust control in discrete event systems is also an active area of research [1,10,19–21,26,31,33,39,44,47,48]. However, they usually deal with specific types of faults such as communication delays, loss of information, or deception

attacks [1,20,21,26,31,39,47]. We capture model uncertainty with our robustness definition, which can be attributed to these faults. Robustness against model uncertainty is tackled in the works of [10,19,44,48]. In these works, deviations are modeled by the behavior generated by the environment. On the other hand, we modeled deviations by the inclusion of extra transitions. In [11], a controller realizability problem is studied for environments modeled as modal transition systems, where a controller satisfies a property in all, some, or none of the LTS family. Our notion of robustness explicitly computes which systems in the LTS family satisfy the property.

Lastly, robustness also relates to fault-tolerance. Fault-tolerance has been studied in the context of distributed systems [13,22,34]. In [6,9,12,14], synthesis of fault-tolerant programs by retrofitting initial fault-intolerant programs. These works focus on specific types of fault models, whereas our robustness model computes the safety envelope the controller is robust against.

8 Conclusion

In this paper, we introduced a new notion of robustness against environmental deviations for discrete-state transition systems. Our notion of robustness is syntactically defined by additional transitions and semantically defined by the controlled behavior generated by these additional transitions. We provided two methods to compute robustness: a brute-force algorithm, and an algorithm based on a controller synthesis problem. We implemented these methods in a prototype tool which we used to analyze several case studies. In these case studies, we demonstrated that our robustness analysis provides crucial information by identifying the environmental envelopes in which the system can guarantee its safety properties.

As part of future work, we plan to extend our work to investigate robustness in the context of partially observable systems as well as in stochastic systems such as Markov decision processes (MDPs). We also plan to investigate the benefit of considering additional environmental states–as well as additional transitions–in our robustness analysis. Finally, we plan to extend our work beyond safety properties, e.g. including liveness.

Acknowledgements. This project was supported by the US NSF Awards CCF-2144860, CNS-1801342, CNS-1801546, CCF-1918140, and ECCS-2144416.

References

1. Alves, M.V.S., da Cunha, A.E.C., Carvalho, L.K., Moreira, M.V., Basilio, J.C.: Robust supervisory control of discrete event systems against intermittent loss of observations. Int. J. Control 1–13 (2019)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Bloem, R., et al.: Synthesizing robust systems. Acta Inf. **51**(3–4), 193–220 (2014)
4. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B.: Synthesizing robust systems. In: 2009 Formal Methods in Computer-Aided Design, pp. 85–92 (2009)

5. Bolton, M.L., Bass, E.J.: Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In: 2011 IEEE International Conference on Systems, Man, and Cybernetics, pp. 1788–1794 (2011). https://doi.org/10.1109/ICSMC.2011.6083931
6. Bonakdarpour, B., Kulkarni, S.S.: SYCRAFT: a tool for synthesizing distributed fault-tolerant programs. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 167–171. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_16
7. Introduction to Discrete Event Systems. Lecture Notes in Computer Science, Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72274-6_9
8. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving programs robust. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011), pp. 102–112. Association for Computing Machinery (2011)
9. Cheng, C.-H., Rueß, H., Knoll, A., Buckl, C.: Synthesis of fault-tolerant embedded systems using games: from theory to practice. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 118–133. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_10
10. Cury, J., Krogh, B.: Robustness of supervisors for discrete-event systems. IEEE Trans. Automat. Control **44**(2), 376–379 (1999)
11. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: The modal transition system control problem. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 155–170. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_15
12. Ebnenasir, A., Kulkarni, S.S., Arora, A.: FTSyn: a framework for automatic synthesis of fault-tolerance. Int. J. Softw. Tools Technol. Transf. **10**(5), 455–471 (2008)
13. Gärtner, F.C.: Fundamentals of fault-tolerant distributed computing in asynchronous environments. ACM Comput. Surv. **31**(1), 1–26 (1999)
14. Girault, A., Rutten, E.: Automating the addition of fault tolerance with discrete controller synthesis. Formal Method. Syst. Des. **35**, 190–225 (2009)
15. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36387-4
16. Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz robustness of finite-state transducers. In: Raman, V., Suresh, S.P. (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 29, pp. 431–443. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2014)
17. Kang, E.: Robustness analysis for secure software design. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment (SEAD 2020), pp. 19–25. Association for Computing Machinery (2020)
18. Leveson, N., Turner, C.: An investigation of the therac-25 accidents. Computer **26**(7), 18–41 (1993). https://doi.org/10.1109/MC.1993.274940
19. Lin, F.: Robust and adaptive supervisory control of discrete event systems. IEEE Trans. Automat. Control **38**(12), 1848–1852 (1993)
20. Lin, F.: Control of networked discrete event systems: dealing with communication delays and losses. SIAM J. Control Optimiz. **52**(2), 1276–1298 (2014)
21. Lin, L., Zhu, Y., Su, R.: Towards bounded synthesis of resilient supervisors. In: 2019 IEEE 58th Conference on Decision and Control (CDC), pp. 7659–7664 (2019)

22. WDAG 1996. LNCS, vol. 1151. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61769-8_9
23. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. John Wiley and Sons Inc, USA (2000)
24. Majumdar, R., Render, E., Tabuada, P.: Robust discrete synthesis against unspecified disturbances. In: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control (HSCC 2011), pp. 211–220. Association for Computing Machinery (2011)
25. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2), 149–184 (1993)
26. Meira-Góes, R., Marchand, H., Lafortune, S.: Towards resilient supervisors against sensor deception attacks. In: 2019 IEEE 58th Annual Conference on Decision and Control (CDC) (2019)
27. Meira-Góes, R., Dardik, I., Kang, E., Lafortune, S., Tripakis, S.: Safe environmental envelopes of discrete systems. Zenodo (2023). https://doi.org/10.5281/zenodo.7999482
28. Meira-Goes, R., Dardik, I., Kang, E., Lafortune, S., Tripakis, S.: Transitional robustness github repository (2023). https://github.com/cmu-soda/transitional-robustness. Accessed 29 May 2023
29. Meira-Góes, R., Kang, E., Lafortune, S., Tripakis, S.: On tolerance of discrete systems with respect to transition perturbations. arXiv:2110.04200 [eess.SY] (2021)
30. Meira-Góes, R., Kang, E., Lafortune, S., Tripakis, S.: On synthesizing tolerable and permissive controllers for labeled transition systems. In: 16th IFAC Workshop on Discrete Event Systems WODES 2022, vol. 55, no. 28, pp. 158–164 (2022)
31. Meira-Goes, R., Lafortune, S., Marchand, H.: Synthesis of supervisors robust against sensor deception attacks. IEEE Trans. Automat. Control **66**(10), 4990–4997 (2021)
32. Neider, D., Weinert, A., Zimmermann, M.: Synthesizing optimally resilient controllers. Acta Inf. **57**(1), 195–221 (2020)
33. Paoli, A., Lafortune, S.: Safe diagnosability for fault-tolerant supervision of discrete-event systems. Automatica **41**(8), 1335–1347 (2005)
34. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980)
35. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989), pp. 179–190. Association for Computing Machinery (1989)
36. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), pp. 46–57 (1977)
37. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
38. U.S. Attorney's Office Eastern District of Kentucky. Clay county officials and residents convicted on racketeering and voter fraud charges (2010). https://archives.fbi.gov/archives/louisville/press-releases/2010/lo032510.htm
39. Rohloff, K.: Bounded sensor failure tolerant supervisory control. In: 11th IFAC Workshop on Discrete Event Systems, vol. 45, no. 29, pp. 272–277 (2012)
40. Samanta, R., Deshmukh, J.V., Chaudhuri, S.: Robustness analysis of string transducers. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 427–441. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_30

41. Sempreboni, D., Viganò, L.: X-men: a mutation-based approach for the formal analysis of security ceremonies. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 87–104 (2020). https://doi.org/10.1109/EuroSP48549.2020.00014
42. Tabuada, P., Balkan, A., Caliskan, S.Y., Shoukry, Y., Majumdar, R.: Input-output robustness for discrete systems. In: Proceedings of the Tenth ACM International Conference on Embedded Software (EMSOFT 2012), pp. 217–226. Association for Computing Machinery (2012)
43. Tabuada, P., Neider, D.: Robust Linear Temporal Logic. In: Talbot, J.M., Regnier, L. (eds.) 25th EACSL Annual Conference on Computer Science Logic (CSL 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 62, pp. 10:1–10:21. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2016)
44. Takai, S.: Maximizing robustness of supervisors for partially observed discrete event systems. Automatica **40**(3), 531–535 (2004)
45. Topcu, U., Ozay, N., Liu, J., Murray, R.M.: On synthesizing robust discrete controllers under modeling uncertainty. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control (HSCC 2012), pp. 85–94. Association for Computing Machinery (2012)
46. Tun, T.T., Bennaceur, A., Nuseibeh, B.: Oasis: weakening user obligations for security-critical systems. In: 2020 IEEE 28th International Requirements Engineering Conference (RE), pp. 113–124 (2020). https://doi.org/10.1109/RE48521.2020.00023
47. Wang, F., Shu, S., Lin, F.: Robust networked control of discrete event systems. IEEE Trans. Automat. Sci. Eng. **13**(4), 1528–1540 (2016)
48. Young, S., Garg, V.K.: Model uncertainty in discrete event systems. SIAM J. Control Optimiz. **33**(1), 208–226 (1995)
49. Zhang, C., Garlan, D., Kang, E.: A behavioral notion of robustness for software systems. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), pp. 1–12. Association for Computing Machinery (2020)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.