Robustification of Behavioral Designs against Environmental Deviations

Changjian Zhang
Carnegie Mellon University
Pittsburgh, PA USA
changjiz@andrew.cmu.edu

Tarang Saluja
Swarthmore College
Swarthmore, PA USA
tsaluja1@swarthmore.edu

Rômulo Meira-Góes
The Pennsylvania State University
State College, PA USA
romulo@psu.edu

Matthew Bolton
University of Virginia
Charlottesville, VA USA
mlb4b@virginia.edu

David Garlan
Carnegie Mellon University
Pittsburgh, PA USA
dg4d@andrew.cmu.edu

Eunsuk Kang
Carnegie Mellon University
Pittsburgh, PA USA
eunsukk@andrew.cmu.edu

***Abstract*—Modern software systems are deployed in a highly dynamic, uncertain environment. Ideally, a system that is *robust* should be capable of establishing its most critical requirements even in the presence of possible *deviations* in the environment. We propose a technique called *behavioral robustification*, which involves systematically and rigorously improving the robustness of a design against potential deviations. Given behavioral models of a system and its environment, along with a set of user-specified deviations, our robustification method produces a redesign that is capable of satisfying a desired property even when the environment exhibits those deviations. In particular, we describe how the robustification problem can be formulated as a *multi-objective optimization* problem, where the goal is to restrict the deviating environment from causing a violation of a desired property, while maximizing the amount of existing functionality and minimizing the cost of changes to the original design. We demonstrate the effectiveness of our approach on case studies involving the robustness of an electronic voting machine and safety-critical interfaces.**

I. Introduction

A fundamental activity in software engineering is establishing the following relationship between requirements, domain assumptions, and specifications [1], [2]:

$$M, E \vdash P$$

That is, if the machine (e.g., software being developed) satisfies its specification (M) and certain environmental assumptions (E) hold, then the desired requirement (P) must follow.

In practice, the environment rarely remains fixed, and often *deviates* from its expected behavior over time, due to a fault or a change in its operating conditions. For example, to ensure safe treatment, a radiation therapy system may rely on assumptions about the order in which a therapist carries out various actions (e.g., enter patient ID, set radiation settings) through its clinical software interface. However, the therapist might inadvertently make a mistake from time to time (e.g., omit or repeat an action), and unless the system is explicitly designed to tolerate these errors, safety failures might occur [3]–[5]. Similarly, a web protocol that is proven to be secure may become vulnerable to attacks when some of its participants deviate from the intended protocol steps [6], [7].

Ideally, a system that is *robust* would ensure that a critical requirement is satisfied even under possible deviations in the environment (e.g., protect patient from radiation overdose even if the therapist commits an error). In our prior work [8], we proposed a formal definition of *robustness* to enable a rigorous design-level analysis. In this definition, a system (M) is said to be *robust* with respect to some property (P) and some set of environmental deviations (δ) if M is capable of satisfying P under the environment (E') that exhibits these deviations; i.e., $M, E' \vdash P$ (roughly, $E' = E \oplus \delta$, where E is the original, expected environment). Furthermore, given models of M and E as labelled transition systems, we proposed a technique for automatically computing the set of deviations against which the system is (not) robust. The output of this analysis provides information about how to redesign the system to be more robust, although in [8], this step was left as a manual task for the developer to perform.

In this paper, we propose a technique called *behavioral robustification* as an approach to systematically improving the robustness of a software system at the design stage. In particular, given models of a system (M) and its environment (E) specified as transition systems, along with a set of possible deviations (δ), our approach *robustifies* M into a new design, M', such that M' is capable of satisfying desired property P even under those deviations. For example, given models of a user interface for a radiation therapy system (M), the expected therapist behavior (E), and a set of possible human errors (δ), our robustification method constructs a redesign of the interface, M', that prevents a safety failure (e.g., patient overdose) even when the therapist commits one of those errors.

There are a number of technical challenges to overcome in developing an effective robustification method. First, the space of possible candidate redesigns (M') can be enormous, and so an effective method must be able to efficiently search this space. Second, not all of these redesigns may be desirable. If P is a safety property (i.e., "something bad should not happen"), then a redesign that simply disables all of the environmental events is a trivial solution, but also not a useful one, as it would disable existing system functionality. In addition, a redesign that incurs a small cost of change is arguably more desirable than one that drastically modifies the existing design.

To capture the desirability of a candidate redesign, we introduce two types of quality metrics: (1) the amount of *common behavior* with respect to the original design M, and (2) the *cost of change*. Then, robustification becomes a multi-objective optimization problem [9], where the goal is to find a redesign M that preserves as much of the existing behavior as possible while minimizing the cost of changes incurred. In this paper, we describe a novel robustification method that leverages techniques from supervisory control theory [10] to automatically generate a set of optimal candidate redesigns. As far as we are aware, **our approach is the first to enable automated robustification of designs that take into account multiple quality metrics**.

We have built a prototype implementation of our robustification method and demonstrated its feasibility on three case studies: (1) an electronic voting protocol, where the original design was vulnerable to voter fraud [11], (2) a radiation therapy interface, where the therapist error could result in a safety failure, and (3) an infusion pump system, where possible environmental deviations include not only therapist errors but power failures. We show how our approach can be used to automatically robustify these systems into ones that are robust against the respective deviations.

The contributions of this paper are as follows:

- A formal definition of *robustification* and a formulation as a multi-objective optimization problem over two quality metrics for robustified designs, (Section IV);
- A novel approach to robustification that leverages supervisory control theory (Section V);
- A set of heuristics for efficiently generating optimal redesigns (Section VI), and;
- Three case studies that illustrate the feasibility of our approach (Section VII).

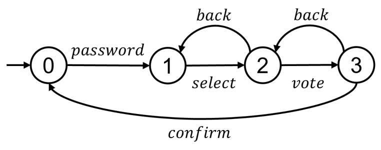
II. Motivating Example

As a running example, consider a simplified design of an electronic voting system, described in [12] and based on a real system that was used in a 2010 election in Kentucky, USA. During the election, the system was exploited by malicious actors, where they were able to commit voter fraud by modifying the vote selection made by other voters [11].

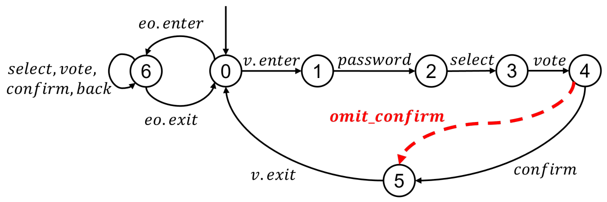
The system consists of a touch-screen voting machine named iVotronic[1]. Figure 1a shows a simplified model of the machine (M) as a *labelled transition system* (LTS), where labels on the edges represent events. In a typical scenario, a voter is expected to interact with the machine by carrying out the following actions:

- Enter password to verify their identity (*password*);
- Select the candidate of their choice (*select*);

[1]https://verifiedvoting.org/election-system/ess-ivotronic



(a) The voting machine M.



(b) The normative environment E (without the voter omission error in red); and the deviated environment E' (with the omission error). Event prefixes *eo* and *v* correspond to the official and the voter, respectively.

Fig. 1: Models of the voting machine and the environment

- Proceed to the next step by pressing vote (*vote*) or return to the previous step by pressing back (*back*);
- Complete the vote by confirming the selection (*confirm*) or return to the previous step by pressing back (*back*).

The voting machine is placed inside a voting booth, and a nearby election official is responsible for guiding voters to and away from the booth. We assume that the voting booth can be occupied by at most one person at a time. In addition, we assume that some of the election officials may be politically motivated or malicious, in that they have an intent to tamper with or manipulate votes that are cast by other voters.

Figure 1b without the red labelled edge represents the *normative* behavior of the environment E, which consists of a (potentially malicious) election official and a voter. This model shows the voter carrying out the voting process in the expected order of events, first by entering the booth, entering a password and desired candidate selection, and confirming the vote before exiting. Note that the election official can also interact with the machine, by entering the booth when it is empty.

Next, we describe a typical workflow that the system developer would follow in our proposed robustification approach.

(1) Analysis under the normative environment: Suppose that the developer is concerned about potential voter fraud and wishes to check the design of the machine against the following integrity requirement:

For each voter, the voting system must record the vote that was selected by that voter.

This type of system requirement can be specified as a safety property [13], and existing analysis tools (such as model checkers [14]) could be leveraged to check that the system satisfies the property under the normative environment (i.e., $M||E \models P$, where P is a specification of the property). Given the specific M and E shown in Figure 1, an analysis tool confirms that P indeed holds: Since only the voter is able to enter the password, the official is unable to modify other votes.

(2) Analysis under the deviated environment: The developer now wishes to check whether the same requirement holds even when the voter begins to *deviate* from the expected behavior.

In particular, the developer is aware that certain populations of voters may not be familiar with e-voting interfaces and inadvertently commit errors from time to time, such as omitting to confirm the vote selection before exiting the voting booth (also called a *post-completion error* [15]). This deviation (δ) is specified by augmenting the original model E with an additional transition (*omit_confirm*), resulting in a new model of the environment (i.e., $E' = E \oplus \delta$), as shown in Figure 1b.

A re-analysis of the same integrity requirement (P) under this new environment reveals that the property no longer holds (i.e., $M||E' \not\models P$). A counterexample depicts the following scenario in the system: After pressing *vote*, the voter exits the booth without confirming; the election official then enters the booth, presses *back* twice to go back to the selection screen, selects the candidate of their own interest, and then completes the rest of the voting process—resulting in a possibly incorrect vote being recorded for the voter (i.e., a violation of P). This scenario depicts the actual voter fraud that was committed by election officials during the aforementioned Kentucky election (which ultimately led to arrests of co-conspirators [11]).

(3) Robustification: Based on the result of the analysis in Step (2), the developer wishes to *robustify* the existing design against possible deviations; i.e., it becomes capable of ensuring P even under the deviated environment. However, given a large number of ways in which M can be modified, it would be too costly and time-consuming for the developer to construct and check candidate redesigns manually. Instead, the developer can use our tool to perform the following *robustification* task:

> Given system design M, environment E, deviations δ, and property P such that $M||E' \not\models P$ for deviated environment $E' = E \oplus \delta$, construct a redesign M' such that $M'||E' \models P$.

Not every solution to this problem, however, may be desirable to the developer. For example, one possible way to robustify M is to remove all of the *back* transitions; this way, the election official would be prevented from changing the vote that was already cast by the voter, thus ensuring that the new design satisfies the integrity requirement. However, this design is also undesirable, in that it also removes the ability for the voter to modify their vote by pressing *back*.

To enable generation of more "desirable" solutions, we consider two quality metrics for candidate redesigns: (1) the solution should retain the behavior of the original design as much as possible, and (2) the solution should incur minimal cost of change. Then, the above task can be rephrased as the following *optimization* problem:

> Given design M, environment E, deviation δ, and property P, for $E' = E \oplus \delta$, construct M' such that $M'||E' \models P$, and M' *maximizes* common behavior with M and *minimizes* cost of change.

In the following sections, we formally define the robustification problem and a notion of optimal redesigns in terms of the above quality metrics. We then present a method that leverages supervisory control theory [10] to generate optimal redesigns.

III. Preliminaries

A. Labelled Transition System

A *labelled transition system* (LTS) T is a tuple $\langle S, \alpha T, R, s_0 \rangle$ where S is a set of states, αT is a set of events called the *alphabet* of T, $R \subseteq S \times \alpha T \cup \{\tau\} \times S$ defines the state transitions (where τ is a designated event that is unobservable to the system's environment), and $s_0 \in S$ is the initial state. An LTS is *non-deterministic* if $\exists (s, a, s'), (s, a, s'') \in R : s' \neq s''$ or $\exists (s, \tau, s') \in R$; otherwise, it is *deterministic*. An event $a \in \alpha T$ is *enabled* at state $s \in S$ if $\exists (s, a, s') \in R$; otherwise, a is *disabled* at s.

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

1) Operators: For LTS $T = \langle S, \alpha T, R, s_0 \rangle$, the *projection* operator $\restriction$ exposes a subset of the alphabet of T. Given $T \restriction A = \langle S, \alpha T \cap A, R', s_0 \rangle$, for any $(s, a, s') \in R$, if $a \notin A$, then $(s, \tau, s') \in R'$; otherwise, $(s, a, s') \in R'$. The $\restriction$ operator also applies to traces; $\sigma \restriction A$ denotes the trace that results from removing the occurrences of every event $a \notin A$ from σ. The *parallel composition* $||$ is a commutative and associative operator that combines two LTSs by synchronizing on their common events and interleaving the others [16].

2) Properties: In this work, we consider a class of properties called *safety properties* [13], which define the acceptable behaviors of a system. A safety property P can be represented as a deterministic LTS, and we say that an LTS T satisfies P if and only if $beh(T \restriction \alpha P) \subseteq beh(P)$.

We also consider another class of properties called *progress properties*, which are a restricted subset of liveness properties [13]. A progress property $L \subseteq \alpha T$ states that the system must eventually be able to execute $a \in L$ along all paths [17].

B. Supervisory Control

Our proposed robustification approach leverages techniques from an area of control theory called *supervisory control* [10]. Supervisory control assumes an "uncontrolled" system (also called *plant*) for which a desired property needs to be enforced. The premise is that the plant may not satisfy the property on its own, and it needs to be "controlled" by *restricting* its behavior to a subset of its original behavior. This modification is done by a component named *supervisory controller*.

Given a deterministic LTS G as the model of a plant that needs to be controlled, a *controller* S for G is a function that maps any trace in $beh(G)$ to a subset of events in αG, i.e., $S : beh(G) \rightarrow 2^{\alpha G}$. Then, given a trace $\sigma \in beh(G)$, $S(\sigma)$ defines the set of events that G is *allowed* to perform after σ.

A typical controller S has limited actuation and sensing capabilities. These limited capabilities are described by the pair of partitions of αG: (1) αG_c and αG_{uc}, which represent the sets of *controllable* and *uncontrollable* events; and (2) αG_o and αG_{uo}, which represent the sets of *observable* and *unobservable* events. Intuitively, a controller only perceives events in αG_o and can only disable events in αG_c. Then, we formally define a controller as follows:

Definition 3.1: A *supervisory controller* is a function $S : beh(G \upharpoonright \alpha G_o) \rightarrow 2^{\alpha G}$ s.t. $\forall \sigma \in beh(G \upharpoonright \alpha G_o) : \alpha G_{uc} \subseteq S(\sigma)$. The control enforced by a controller can change only after some observable event occurs. Also, in this paper, we assume that *every* controllable event is observable, i.e., $\alpha G_c \subseteq \alpha G_o$.

A controller S can also be represented as a deterministic LTS, where given trace $\sigma \in beh(G)$, only events in $S(\sigma)$ are enabled at the state reached after executing σ. In the following sections, unless explicitly specified, S refers to the LTS representation of a controller. Then, the behavior defined by applying a controller S to G (i.e., plant under control) can be represented by $beh(S||G)$.

Finally, the goal of *supervisory controller synthesis* is to find a controller S over plant G to achieve property P:

Definition 3.2: Given plant G with controllable events αG_c and observable events αG_o, $\alpha G_c \subseteq \alpha G_o$, and property P, a *controller synthesis* problem $C(G, P, \alpha G_c, \alpha G_o)$ searches for a *minimally restrictive* controller S such that $S||G \models P$.

Supervisory control theory provides algorithmic techniques for computing a controller; more details can be found in [10].

IV. Robustification problems

A. Basic Robustification Problem

Let us first introduce the concepts of a *deviation model* and the *augmentation* operator $\oplus$. A deviation model describes how the environment may deviate from its original behavior, in terms of additional transitions, states, or events:

Definition 4.1: Given an LTS $T = \langle S, \alpha T, R, s_0 \rangle$ and a deviation model $\delta = \langle S_\delta, \alpha\delta, R_\delta \rangle$, where $S \subseteq S_\delta$, $\alpha T \subseteq \alpha\delta$, and $R_\delta \subseteq S_\delta \times \alpha\delta \times S_\delta$, the *augmentation* operator $\oplus$ augments T by adding states and transitions to it, i.e., $T \oplus \delta = \langle S_\delta, \alpha\delta, R \cup R_\delta, s_0 \rangle$, and $beh(T) \subseteq beh(T \oplus \delta)$.

For example, in Figure 1b, to model the deviation from the expected voter behavior, the original environment model is augmented with an additional transition over a new event, *omit_confirm* $\in \alpha\delta \setminus \alpha T$, from state s_4 to s_5.

One might also consider deviations that involve *removing* behaviors from the environment (i.e., remove transitions or states). In this paper, we focus on adding behaviors only, as we believe that it already captures a large and interesting class of deviations where the environment exhibits behaviors beyond those captured in its original model (e.g., security attacks, human errors, etc., [8]). A deviation model that integrates both adding and removing behaviors is part of our future work.

Then, the task of *robustifying* a design is defined as follows:

Definition 4.2 (Robustification): Given system M, environment E, a deviation model δ, and property P such that $M||E \models P$, the goal of robustification, $Rb(M, E, \delta, P)$, is to find an LTS M' such that for $E' = E \oplus \delta$, $M'||E' \models P$.

Property P can be a combination of safety and progress properties. A safety property defines the unsafe behavior that should be avoided. However, it is possible to have an overly restrictive M' that satisfies the safety property, but does nothing "meaningful". Recall the voting example in Section II; we could disable all *confirm* events, but this solution would also prevent voters from being able to confirm their votes. A progress property can be specified to avoid such "useless" solutions by requiring that *confirm* can eventually occur.

B. Quality Metrics for Robustified Designs

In general, there may be a large number of possible solutions (i.e., M') to the above problem, but some of them may be considered more desirable than others. We consider two desirable qualities of a robustified design: (1) the redesign should retain as much of the important functionality from the original design as possible, and (2) the cost of modifying M to M' should be small. Let us further elaborate on these two:

1) Common Behavior: To define the first quality, we introduce the notion of *preferred behavior*. A preferred behavior D is an execution trace and represents an operational scenario that the developer wishes for machine T to contain[2], i.e., $D \in beh(T \upharpoonright \alpha D)$. Then, maximizing the common behavior between the original design M and the new design M' can be formulated as maximizing the number of D's such that $D \models M||E$ and $D \models M'||E'$. Formally:

Definition 4.3 (Preferred Behaviors): Given a set of preferred behaviors $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, we state $\mathcal{D} \models T$ for some LTS T if and only if $\bigwedge_{D_i \in \mathcal{D}} D_i \models T$.

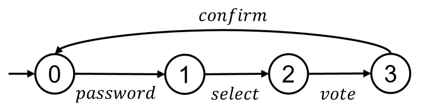
Moreover, the developer may associate each scenario D_i with a different importance value. Then, we can quantitatively measure the amount of common behavior achieved by M' in terms of the total importance value of the subset of preferred behaviors $\mathcal{D}' \subseteq \mathcal{D}$ that is retained by $M'||E'$.

2) Cost of Changes: The second type of quality that we introduce is the cost of change between the original and new design. One way to measure the cost would be in terms of syntactic differences between M and M', e.g., the number of changes to states and transitions. However, these syntactic-based changes in LTS do not necessarily reflect the actual cost of redesign effort.

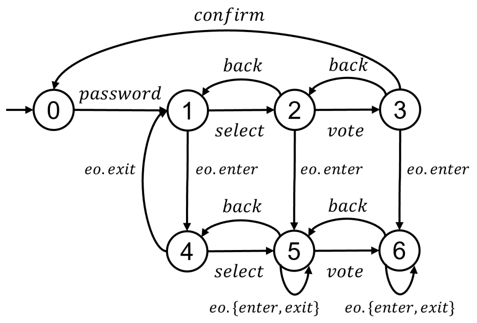
Instead of syntactic changes to an LTS, our intuition is that the cost of redesign can be better approximated by the set of *environment and system events that are observed or controlled* by the system for the purpose of robustification. Intuitively, to make the system more robust, one may need to place an additional sensor to observe a part of the environment (e.g., add an ID scanner to observe $\{$*v*, *eo*$\}$.*enter*, $\{$*v*, *eo*$\}$.*exit*) or modify an existing actuator to disable a particular event under certain situations (e.g., make the *confirm* button toggleable).

More precisely, the developer can designate a pair of event sets, $\mathcal{A} = (\mathcal{A}_c, \mathcal{A}_o)$, where $\mathcal{A}_c, \mathcal{A}_o \subseteq \alpha E \cup \alpha M$, that are controllable and observable, respectively, for the purpose of robustification. Furthermore, each event in $\mathcal{A}$ can be associated with a cost measure to reflect the effort of implementing an actuator or a sensor to control or observe (respectively) that event in the real world. This, in turn, allows us to measure the total cost of changes as the sum of the individual costs of the events in $\mathcal{A}$ that are used to robustify the system.

[2]We denote this as $D \models T$, based on the interpretation of $\models$ as trace inclusion, where αD refers to the events in trace D.



(a) Redesign by disabling *back*.



(b) Redesign by observing additional events *eo.*{*enter, exit*} and controlling *confirm* as needed.

Fig. 2: Alternative ways to robustify the voting machine.

3) Example: Consider two alternative ways to redesign the voting machine, as shown in Figure 2. In solution (a), we remove all the *back* events, to prevent the vote from being changed. Alternatively, in (b), we add *eo.enter* and *eo.exit* events to the system model so that the machine can keep track of who is in the voting booth and disable *confirm* depending on the context (e.g., if the election official has entered the booth).

In terms of preserving behavior of the original design, the developer may decide that being able to *select* and then go *back* is an essential operation that should be retained, since it is crucial for the voter to be able to change their vote. Then, the developer could specify trace ⟨*select, back*⟩ as a preferred behavior, to assert their preference for its inclusion in the redesign M'. It's easy to see that solution (a) does not satisfy this preferred behavior but (b) does.

In terms of the cost of changes, both solutions require adding or removing states and transitions to LTS M, but it may be inaccurate to conclude that one is less "costly" than another simply because it requires fewer syntactic changes. Solution (a) may involve removing all the *back* buttons from the user interface (costly if the development of the interface was outsourced to another company, for example). On the other hand, the second solution may require extending the voting machine with an ID scanner to determine the identity.

C. Optimal Robustification Problem

Given a robustification problem $Rb(M, E, \delta, P)$, preferred behavior $\mathcal{D}$, and modifiable events $\mathcal{A}$, let $\vec{R} = \langle M', \mathcal{D}', \mathcal{A}' \rangle$ be a solution such that it satisfies a subset of preferred behaviors $\mathcal{D}' \subseteq \mathcal{D}$ using a subset of events $\mathcal{A}' = (\mathcal{A}'_c, \mathcal{A}'_o)$ where $\mathcal{A}'_c \subseteq \mathcal{A}_c$ and $\mathcal{A}'_o \subseteq \mathcal{A}_o$. We define the following objective function:

$$\vec{U}(\vec{R}) = \langle U_D(\vec{R}), U_A(\vec{R}) \rangle$$

- $U_D(\vec{R}) = \sum_{D_i \in \mathcal{D}'} u_d(D_i)$ is the amount of utility gained from fulfilling the preferred behavior, and
- $U_A(\vec{R}) = \sum_{a_c \in \mathcal{A}'_c} u_c(a_c) + \sum_{a_o \in \mathcal{A}'_o} u_o(a_o)$ is the total cost of events used to redesign M.

The *utility function*, $u = (u_d, u_c, u_o)$, assigns different degrees of importance to preferred behaviors and implementation costs to events. Note that $u_d(D_i)$ returns a positive integer whereas $u_c(a_c)$ and $u_o(a_o)$ are non-positive, to reflect the positive and negative impact of preferred behavior and cost, respectively.

Intuitively, using a larger set of events to modify M allows a more fine-grained control over the behavior of the machine, which can help maximize the preferred behavior (i.e., larger $U_D(\vec{R})$). However, modifying more events also leads to a higher cost (i.e., larger negative value of $U_A(\vec{R})$). Thus, the problem becomes a *multi-objective optimization* problem that attempts to generate a solution that maximizes these two conflicting objectives [9]. Formally, this optimization problem, denoted $Opt(Rb, \mathcal{D}, \mathcal{A})$, is defined as follows:

Definition 4.4 (Optimal Robustification): Given a robustification problem Rb, a set of preferred behaviors, $\mathcal{D}$ (where $\mathcal{D} \models M||E$), and a set of available events for modification, $\mathcal{A}$, the goal of $Opt(Rb, \mathcal{D}, \mathcal{A})$ is to find one or more solutions $\vec{R} = \langle M', \mathcal{D}', \mathcal{A}' \rangle$ such that M' is a solution to problem Rb, $\mathcal{D}' \models M'||E'$, and $\vec{R}$ maximizes the objective function $\vec{U}$.

Fig. 2 illustrates the trade-off between the amount of preferred behavior retained and the cost of change. Solution (b) retains more behavior than solution (a) does but also incurs potentially higher implementation costs (identity check versus UI upgrade). In general, the developer may wish to examine and consider multiple such solutions before selecting the final redesign. Next, we describe an algorithm that leverages supervisory control synthesis to generate a set of alternative *Pareto-optimal* redesigns [18].

V. Optimal Robustification Method

We present a method for solving the optimal robustification problem (Defn. 4.4). We first describe an approach for solving the basic robustification problem (Defn. 4.2) using supervisory control, and then present an algorithm that builds on this basic method to generate all Pareto-optimal solutions to the optimization problem, $Opt(Rb, \mathcal{D}, \mathcal{A})$.

A. Basic Robustification as Supervisory Control

The task of robustifying a system can be reduced to the supervisory controller synthesis problem as follows:

Theorem 5.1: Given machine M, environment E, deviation δ, property P, and a set of controllable αG_c and observable events αG_o, where $\alpha G_c \subseteq \alpha G_o \subseteq \alpha G$, let S be a solution to the controller synthesis problem $C(G, P, \alpha G_c, \alpha G_o)$, where $G = M||E'$, $E' = E \oplus \delta$. Then, $M' = S||M$ is a solution to the robustification problem $Rb(M, E, \delta, P)$, where $\alpha M' = \alpha M \cup \alpha G_c \cup \alpha G_o$.

The intuition behind the reduction is as follows: Since M is not capable of ensuring P under the deviated environment E', their overall composition, $G = M||E'$, itself can be treated as a plant that can behave undesirably (i.e., violates P) and thus needs to be controlled. The resulting controller, S, describes

TABLE I: The priority categories for preferred behavior and modifiable events. Priority 0 is used for events with no cost and does not apply to preferred behavior.

	0	1	2	3
Behavior	-	Minor	Important	Essential
Event	No Cost	Cheap	Moderate	Costly

how the interactions between M and E' should be further restricted to ensure P. Thus, composing M and S amounts to augmenting M with the additional control logic in S to ensure P even under the deviation δ.

Our approach also relies on the fact that supervisory control synthesis, by default, generates the minimally restrictive controller (Defintion 3.2), which facilitates the goal of retaining as much behavior from the original design as possible.

Theorem 5.2: Given an optimal robustification problem $Opt(Rb, \mathcal{D}, \mathcal{A})$ and a corresponding controller synthesis problem $C(G, P, \alpha G_c, \alpha G_o)$, supervisory controller synthesis generates a controller S s.t. $M' = S||M$ satisfies the maximal possible $\mathcal{D}' \subseteq \mathcal{D}$ for αG_c and αG_o.

B. Priority-Based Utility Function

The utility function u, introduced in Section IV-C, can be defined in several possible ways. We present one definition that assigns utility values based on *priorities* among preferred behaviors and modifiable events: Given optimization problem $Opt(Rb, \mathcal{D}, \mathcal{A} = (\mathcal{A}_c, \mathcal{A}_o))$, the developer assigns priorities to the elements of $\mathcal{D}$, $\mathcal{A}_c$, and $\mathcal{A}_o$. We provide a default set of priority categories as shown in Table I; in general, the priorities can be configured with other user-defined categories.

In Table I, (1) a preferred behavior with a higher priority indicates that it is more critical to a system (i.e., greater utility), and (2) a controllable or observable event with a higher priority means that it is more costly (greater cost). Formally, let H^D be the function that returns the priority of a given preferred behavior; similarly, H^c and H^o specify priorities for the controllable and observable events, respectively. Then, the overall utility function $u = (u_d, u_c, u_o)$ is defined as follows: $u_d(x) = W(H^D(x))$, $u_c(x) = -W(H^c(x))$, $u_o(x) = -W(H^o(x))$, and $W(i) = 1 + \sum_{k=0}^{i-1} W(k) \cdot |H_k|$, where $|H_k|$ is the number of preferred behaviors and events with priority k, and $W(0) = 0$.

This approach to defining utility is called the *lexicographic method* [9]. With these rules, the cost of making some event controllable or observable is assigned the negative utility value of fulfilling some preferred behavior in the same priority bracket. Also, these rules prioritize saving a cost or fulfilling a preferred behavior in a particular priority bracket over incurring any costs or gaining any utilities with a lower priority. As discussed later in Section VI-A, this enables our algorithm to search in the order of higher-to-lower priorities.

For example, in the voting system, we can define operation sequence "*select* and then *back*" as an *Essential* preferred behavior because it is crucial for the voter to be able to change their selection. Moreover, one might specify that observing *enter* and *exit* events has *Moderate* cost; however, it could be *Costly* to make them controllable, as doing so requires additional mechanisms to control who enters the voting booth.

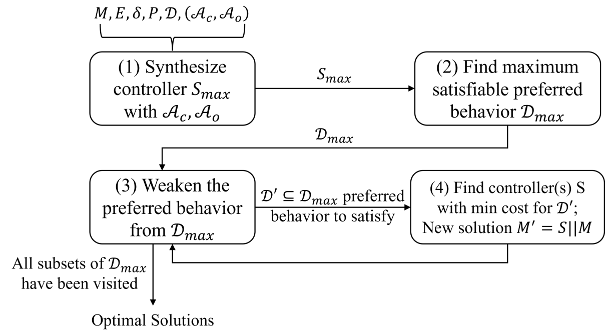

$M, E, \delta, P, \mathcal{D}, (\mathcal{A}_c, \mathcal{A}_o)$
(1) Synthesize controller S_{max} with $\mathcal{A}_c, \mathcal{A}_o$
S_{max}
(2) Find maximum satisfiable preferred behavior $\mathcal{D}_{max}$
$\mathcal{D}_{max}$
(3) Weaken the preferred behavior from $\mathcal{D}_{max}$
$\mathcal{D}' \subseteq \mathcal{D}_{max}$ preferred behavior to satisfy
(4) Find controller(s) S with min cost for $\mathcal{D}'$; New solution $M' = S||M$
All subsets of $\mathcal{D}_{max}$ have been visited
Optimal Solutions

Fig. 3: Overview of NAIVEPARETO for multi-objective search.

C. Algorithm for Multi-Objective Optimization

Figure 3 outlines the process for finding Pareto-optimal solutions, called NAIVEPARETO. At high-level, it employs a top-down, enumerative search approach, where it (1) searches for a solution that fulfills a subset of preferred behaviors $\mathcal{D}' \subseteq \mathcal{D}$ at the lowest cost possible for $\mathcal{D}'$ and (2) iteratively reduces $\mathcal{D}'$ to find other Pareto-optimal solutions.

In Step (1), NAIVEPARETO starts out by synthesizing a controller (S_{max}) that has access to all of the user-specified controllable and observable events ($\mathcal{A}_c$ and $\mathcal{A}_o$). Then, in Step (2), we check whether $S_{max}||M$ satisfies each preferred behavior $D \in \mathcal{D}$. Since this is the most "powerful" controller, based on Theorem 5.2, it fulfills the maximal subset of the user-specified preferred behaviors (D_{max}), while also being the most costly solution.

In Steps (3)-(4), NAIVEPARETO incrementally removes elements from $\mathcal{D}_{max}$ in the order of utility values to find solutions with a lower cost. For example, consider $\mathcal{D}_{max} = \{D_1, D_2, D_3\}$ where $u_d(D_1) = u_d(D_2) < u_d(D_3)$; at iteration $i = 0$, it removes $\emptyset$ from $\mathcal{D}_{max}$; then, at iteration $i = 1$, it tries to find a solution from two $\mathcal{D}'$s by removing $\{D_1\}$ and $\{D_2\}$, respectively; then, at $i = 2$, it removes $\{D_1, D_2\}$, etc., terminating after exploring all the subsets of $\mathcal{D}_{max}$.

In Step (4), at iteration i with $\mathcal{D}'_i \subseteq \mathcal{D}_{max}$, it enumerates all combinations of controllable and observable events except those where $\alpha G_c \nsubseteq \alpha G_o$ (which violates our assumption in Theorem 5.1) and attempts to synthesize a controller for each combination. The goal is to find a controller (if one exists) that fulfills $\mathcal{D}'_i$ at the lowest possible cost; if such a solution exists and is not dominated by any existing solutions, it is stored as one of the Pareto-optimal solutions, to be returned as the final output. It then goes back to Step (3) for the next iteration.

Complexity. The complexity of NAIVEPARETO comes from two tasks: 1) searching all possible combinations of preferred behaviors and events, and 2) controller synthesis. For (1), the complexity is $O(2^{|\mathcal{D}|+|\mathcal{A}^{>0}|})$ where $\mathcal{A}^{>0}$ is the subset of $\mathcal{A}$ with a non-zero cost. For each combination, the algorithm solves a controller synthesis problem, which is in general

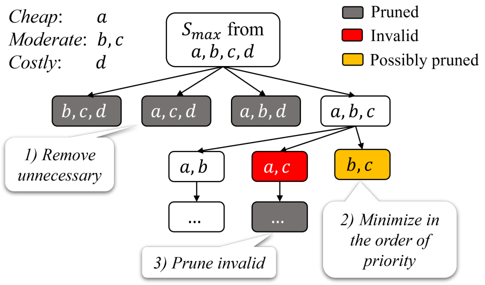


Fig. 4: Illustration of the heuristics in SMARTPARETO.

a hard problem (NP-hard) [19], [20]. Thus, the worst-case complexity can be approximated as $O(2^{|\mathcal{D}|+|\mathcal{A}^{>0}|+N})$ where N is the number of states of the plant, $M||E'$.

VI. HEURISTICS FOR MULTI-OBJECTIVE SEARCH

Given the inherent complexity and the brute-force nature of the algorithm, NAIVEPARETO is unlikely to scale to larger models. We present two improved versions of the algorithm: SMARTPARETO and LOCALSEARCH.

A. SMARTPARETO: *Searching with Pruning Strategies*

SMARTPARETO is a variant of NAIVEPARETO that leverages the following set of strategies for pruning the search space, based on insights about supervisory control synthesis:

1) Removing unnecessary events: By analyzing S_{max}, we can extract αG_u and αG_{un} that indicates the set of controlled/observed events and unused events, respectively. For any event $a \in \alpha G_{un}$, we can remove it from further searches if its cost is greater than the total cost of events in αG_u. Since we know that αG_u forms a valid solution, then any solution with event a would always have a higher cost and thus, there is no need to search for them. Similar reduction of a controller can be found in [21] but without the consideration of cost.

2) Minimizing cost in the order of event priority: Because of the strict ordering property of the lexicographic method, we can always remove high priority events before low priority ones. For a combination of events, if removing a high priority event generates a valid solution, then removing a lower priority event from it cannot generate a solution with a lower cost.

3) Pruning invalid combinations: When a combination of events produces no controller (i.e., cannot satisfy the property) or violates some $\mathcal{D}'$, SMARTPARETO can stop minimizing from this combination. This is because removing events from such a combination would further limit the behavior of the controller, which would certainly result in an invalid solution.

For example, in Figure 4, SMARTPARETO first generates S_{max} with events $\{a, b, c, d\}$. By analyzing S_{max}, we know that $\{a, b, c\}$ forms a valid solution and $\{d\}$ is not used. Since d's cost is higher than the total cost of $\{a, b, c\}$, we don't need to search for any combinations with d (Heuristic 1). Then, to minimize $\{a, b, c\}$, it first removes event b and c respectively before a because if $\{a, b\}$ or $\{a, c\}$ generate a valid solution, then we don't need to search from $\{b, c\}$ as we cannot find a lower cost solution from it (Heuristic 2). Finally, if $\{a, c\}$ is an invalid solution, its followed set $\{a\}$ and $\{c\}$ are also invalid; thus, they do not need to be searched (Heuristic 3).

As demonstrated in Section VII, these heuristics improve the performance of search by significantly reducing the number of synthesis calls while still guaranteeing the Pareto-optimality.

B. LOCALSEARCH: *Finding Locally Optimal Solutions*

As a further improvement to SMARTPARETO, we present another algorithm called LOCALSEARCH that gives up the Pareto-optimality as a trade-off for improved performance.

LOCALSEARCH is similar to NAIVEPARETO but replaces the minimizing process in Step (4) of Figure 3. At iteration i with $\mathcal{D}'_i$, instead of enumerating every possible combination of events, it incrementally removes one event at a time (high priority events before low priority events) from the given event set if removing that event would still generate a controller and retain $\mathcal{D}'_i$. The result is a local-optimal solution w.r.t. $\mathcal{D}'_i$, i.e., removing any event from it would produce no controller or violate $\mathcal{D}'_i$. However, it does not guarantee the cost to be the minimal and thus is not necessarily Pareto-optimal.

For example, consider events $\{a, b, c, d\}$ where $u(a) = u(b) = u(c) < u(d)$. LOCALSEARCH first removes event d and checks whether a valid solution exists. Then, it arbitrarily selects one of a, b, or c to be removed since they have the same cost. Suppose it removes c and finds that removing either a or b after that would result in an invalid solution; then, LOCALSEARCH returns $\{a, b\}$ as a solution. This is locally optimal but not necessarily Pareto-optimal, since $\{c\}$ might also allow a valid solution, and has a lower cost than $\{a, b\}$.
Complexity. The complexity of LOCALSEARCH becomes $O(|\mathcal{A}^{>0}| \cdot 2^{|\mathcal{D}|+N})$. Compared to NAIVEPARETO and SMARTPARETO, it requires much fewer synthesis instances and thus is more efficient. Although it finds only local-optimal solutions, our evaluation in Section VII suggests that these solutions are often good enough compared to Pareto-optimal solutions.

VII. EVALUATION

We present an evaluation of our robustification approach on three case studies. We focus on two research questions:

RQ1 (Scalability): How well do our robustification algorithms scale? Do the heuristics in SMARTPARETO improve the performance of NAIVEPARETO? How does LOCALSEARCH compare against the two?

RQ2 (Quality of robustification): How does our robustification approach compare to other existing methods in terms of the quality of the generated solutions?

Our tool[3], called FORTIS, builds on two existing tools: LTSA, a modeling and analysis tool based on a process algebra called FSP [17], and Supremica [22], a state-of-the-art supervisory controller synthesis tool. FORTIS uses LTSA for specifying and verifying system and environment models, and Supremica to perform controller synthesis as part of the robustification algorithms. Our experiments were conducted on a Linux machine with a 3.6GHz CPU and 16GB memory.

[3]Available at: https://github.com/cmu-soda/Fortis

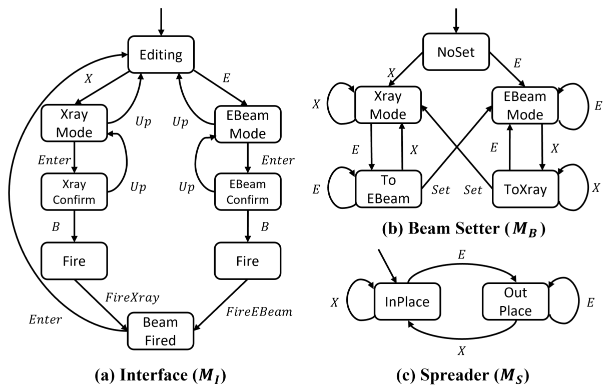


Fig. 5: A model of a radiation therapy system.

A. Electronic Voting Machine

Recall the voting example described in Section II, with the following safety property: *the voting machine must record the vote that was selected by that voter*. We also impose a progress property that event *confirm* can eventually take place.

Preferred behaviors and events. In our experiment, we defined the following preferred behavior:

- D_1 (*Essential*): The voter should be able to change their vote by performing *select, back, select, vote, confirm*.

We assigned *NoCost* for observing all the internal events of the voting machine but *Cheap* cost for controlling them. We also specified that making $\{v, eo\}$*.enter* and $\{v, eo\}$*.exit* observable has *Moderate* cost and making them controllable is *Costly*. In practice, these costs might manifest as adding an ID scanner to determine who is entering or exiting (for observability) or a more costly security mechanism (e.g., an enclose booth with a machine-controlled door) to control entry into the booth (for controllability).

Results. With SMARTPARETO, FORTIS returns 16 Pareto-optimal solutions. As an example, one of them requires observing *eo.*$\{$*enter, exit*$\}$ and controlling *confirm*. It observes the official entering and exiting, and disables *confirm* when the official changes the vote. In addition, the LOCALSEARCH method returns one of the Pareto-optimal solutions.

B. Radiation Therapy System

Consider a radiation therapy machine described in [8], similar to the Therac-25 machine that exposed several patients to overdose [3]. The components of the system (Figure 5) are the *treatment interface* (M_I) that defines how the operator controls the machine, the beam setter (M_B) that determines the mode of radiation (electron beam vs stronger X-ray), and the spreader (M_S) that limits dose during the X-ray mode. The overall system model (M) is given as $M_B||M_I||M_S$.

Deviations. Consider the normative environment E for a therapist to treat a patient under the electron-beam mode: The user presses *E*, *Enter*, and then *B* to fire the beam. Then, we consider a deviation where the user erroneously selects the X-ray mode. The resulting deviated environment (E') allows behavior where the therapist presses *X*, *Up* to go back, *E* to select the electron beam, *Enter* to confirm, and finally *B* to fire. However, it could be that the beam setter is still in the process of switching (in state *ToEbeam* in M_B); if the beam is fired before it is fully switched, the patient could be given a much higher level of dose (X-ray) than intended.

We consider (1) a safety property stating that *the X-ray beam does not fire until the spreader is in-place* and (2) a progress property that *FireXray* and *FireEBeam* should eventually occur, to ensure that the machine will still be capable of carrying out treatments even after robustification.

Preferred behaviors and events. We defined the following preferred behaviors:

- D_1, D_2 (*Essential*): The user can select *X*/*E* and then *Up* to change the mode and fire the beam.
- D_3, D_4 (*Important*): The user can perform $\langle Up, Up \rangle$ after having pressed *Enter* to change the mode and fire.

D_1 and D_2 state that it is *Essential* to allow the user to switch the beam in case the wrong one was accidentally selected. Since it is less likely for the user to select the wrong beam and then press *Enter* without noticing the mistake, D_3 and D_4 are assigned a lower priority of *Important*.

We assigned *NoCost* for observing the events of the therapy machine, which are: *X, E, Enter, Up, B, FireXray, FireEBeam*, and *Set*. Then, we assigned *NoCost* to control *FireXray, FireEBeam*, and *Set* but *Cheap* to control *X, E, Enter, Up*, and *B* to reflect the cost of upgrading the user interface for controllability (e.g., by disabling those buttons contextually).

Results. Running FORTIS with SMARTPARETO generated two Pareto-optimal solutions. One solution involves (1) disabling *B* when the system is in X-ray mode and the spreader is out-place and (2) re-enabling *B* when the mode switching is completed by observing *Set*. This solution is similar to the manually devised one in [8]. In addition, LOCALSEARCH finds the other Pareto-optimal solution which disables *Enter* instead of *B*.

C. Infusion Pump

The goal of this case study is to apply our tool to a system that is considerably more complex than the other two. Consider an infusion pump machine (M) that is used to dispense a certain dose of medication through tube lines that are connected to a patient, based on the machine described in [23]. The machine is connected to a *power system* (E_P) with an alarm and a built-in battery which will be charged when the power cable is plugged in. When the cable is unplugged during operation, the power system automatically switches to the battery mode; and when the battery goes low, it rings the alarm to notify the nurse (E_N). We consider the infusion pump as the machine (M), and the overall behavior of the environment is given as $E = E_P||E_N$.

Deviations. We consider a deviation that may occur in the workflow of a nurse (E_N). Normally, the nurse plugs in the cable and starts the machine; then, the nurse sets up the medication rate, starts the dispensation, and waits for its completion. However, a deviation is that the user accidentally unplugs the cable while the pump machine is still dispensing the medication. In one possible scenario (allowed by E'), the battery goes low and the user fails to notice the alarm; then, the machine continues dispensing even when the power fails. This might cause serious medical accidents, such as overdose.

We consider (1) a safety property that *if the machine loses power during medicine dispensation, it should discontinue the dispensation* and (2) a progress property that *the dispensation must be able to eventually complete*.

Preferred behaviors and events. In our experiment, we specified the following two preferred behaviors:

- D_1 (*Essential*): The user should be able to turn on the machine, start and wait for the completion of a dispensation, and then turn off.
- D_2 (*Essential*): The user should be able to resume a dispensation from a power failure.

We define all the machine events to incur *NoCost* to observe. Environmental events like *plug_in* and *battery_charge* are *Costly* to observe, except for *power_failure*, which is made unobservable. All the machine events are free to control except events like *turn_on* and *turn_off*, which are assigned *Moderate* as they might require modifying the UI. Environmental events like *plug_in* are *Costly* to control, and physical events like *battery_spent* and *power_failure* are uncontrollable. The details of the cost assignment can be found in the online repository.

Results. Running SMARTPARETO generated one Pareto-optimal solution. This solution disables the dispensation when the machine is unplugged; it then re-enables it after the machine is plugged in and the battery is charged. The LOCALSEARCH method found the same Pareto-optimal solution.

D. Experimental Results

1) RQ1 (Scalability): Table II summarizes the performance of NAIVEPARETO, SMARTPARETO, and LOCALSEARCH over the case studies. For scalability evaluation, we also tested them over larger variants of *Voting* (by increasing the number of voters) and *Infusion Pump* (by adding a dispensation line).

It can be seen that NAIVEPARETO requires a large number of synthesis calls and times out on the *Voting-2,3,4* and *Pump-2* problems. In comparison, our heuristics for pruning the search space in SMARTPARETO are effective in reducing the number of synthesis calls, resulting in a significant performance improvement over NAIVEPARETO.

The LOCALSEARCH method further improves on the performance by giving up on the Pareto-optimality of the generated solutions. As shown in the case studies, LOCALSEARCH often finds a solution that is the same or close to Pareto-optimal solutions, and thus we believe that this is an acceptable compromise between performance and qualities of the redesigns.

The table also shows that controller synthesis is the key bottleneck. The time to solve one synthesis instance and the size of the solution space grow quickly with the increasing size of the plant. Moreover, for the same problem, the synthesis becomes harder to solve when fewer controllable and observable events are provided (when minimizing the cost). In the future, we plan to explore alternative synthesis techniques, such as GR(1) reactive synthesis [24]–[26],

TABLE II: Times for generating an optimal solution.

	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{A}^{>0}\rvert$	$\lvert M\rvert\lvert E'\rvert$	Space*	#Syn.†	$\mathcal{A}_u$ §	Time‡
Voting-1-N**					2,576	(1, 7)	25.24s
Voting-1-S	1	13	12	6x10^7	195	(1, 7)	3.68s
Voting-1-L					9	(1, 7)	0.54s
Voting-2-N					-	-	T/O
Voting-2-S	1	19	25	4x10^{13}	424	(1, 10)	9.23s
Voting-2-L					14	(1, 11)	1.14s
Voting-3-N					-	-	T/O
Voting-3-S	1	24	32	1x10^{17}	364	(1, 12)	31.04s
Voting-3-L					17	(1, 14)	4.69s
Voting-4-N					-	-	T/O
Voting-4-S	1	29	39	6x10^{20}	754	(1, 14)	6m04s
Voting-4-L					20	(1, 17)	28.31s
Therapy-N					32	(4, 8)	0.98s
Therapy-S	4	5	21	1x10^9	32	(4, 8)	1.02s
Therapy-L					6	(4, 8)	0.56s
Pump-1-N					2,304	(7, 14)	1m04s
Pump-1-S	2	12	104	3x10^{35}	99	(7, 14)	4.85s
Pump-1-L					13	(7, 14)	1.47s
Pump-2-N					-	-	T/O
Pump-2-S	4	16	760	6x10^{234}	-	-	T/O
Pump-2-L					17	(14, 25)	12.47s

* The approximate size of the search space $O(2^{|\mathcal{D}|+|\mathcal{A}^{>0}|+|M||E'|})$.
** -N, -S, and -L stand for NAIVEPARETO, SMARTPARETO, and LOCALSEARCH, respectively.
† The number of controller synthesis instances invoked.
‡ All runs have a 10 minutes timeout.
§ No. of controllable and observable events used for robustification.

2) RQ2 (Quality of robustification): We compared the quality of redesigns generated by our solutions to those by other approaches for robustifying behavioral models.

OASIS. As far as we know, our definitions of robustification problems and related qualities (in Section IV) are new, and there is no existing tool that is directly comparable. However, one existing work that is close to ours is OASIS by Tun et al. [12]. Although they do not explicitly mention robustness, their goal is similar, in that it aims to revise a machine (M) to fulfill a security requirement (P) in an environment (E) where some of the users might deviate from their expected behavior.

Like our approach, OASIS also leverages controller synthesis to generate designs that satisfy a property. However, OASIS and FORTIS differ in the way they generate and explore alternative designs: OASIS uses an *abstraction*-based technique that allows changing the sequencing of actions in the machine to generate alternative designs, while FORTIS allows additional events to be observed or controlled by the redesigned machine.

We note that OASIS is not designed to optimize for the two quality goals. Our comparison is not intended to show FORTIS is superior, but rather that if these quality goals are of importance to the developers, FORTIS may be the preferred method.

Vanilla Supervisory Control. We also compare FORTIS to a vanilla approach that utilizes supervisory controller synthesis to generate robustified designs without considering the two qualities (i.e., it solves the basic robustification problem).

Experiment. Since no tool for OASIS was publicly available,

TABLE III: Comparison results.

	Vanilla			OASIS			FORTIS-LOCAL		
	Sol.*	$\mathcal{A}_u$	Time	Sol.	$\mathcal{A}_u$	Time	Sol.	$\mathcal{A}_u$	Time
Voting-1	$\times$	(5, 5)	0.30s	$\oslash$	(5, 5)	0.45s	✓	(1, 7)	0.54s
Voting-2	$\times$	(7, 7)	0.36s	$\times$	(7, 7)	3.66s	✓	(1, 11)	1.14s
Voting-3	$\times$	(8, 8)	0.58s	$\times$	(8, 8)	23.00s	✓	(1, 14)	4.69s
Voting-4	$\times$	(9, 9)	1.30s	$\times$	(9, 9)	4m13s	✓	(1, 17)	28.31s
Therapy	✓	(8, 8)	0.35s	✓	(8, 8)	0.37s	✓	(4, 8)	0.56s
Pump-1	$\times$	(13, 13)	0.42s	$\oslash$	(13, 13)	1.04s	✓	(7, 14)	1.47s
Pump-2	$\times$	(24, 24)	0.93s	$\oslash$	(24, 24)	9.01s	✓	(14, 25)	12.47s

* ✓: it finds one or more solutions and satisfies all the user-defined preferred behavior; $\oslash$: it finds solutions but does not retain all the preferred behavior; $\times$: it fails to find a solution.

we implemented the algorithm in [12] with Supremica as the underlying controller synthesis engine. We then ran the three approaches over the case study models and compared them with respect to: (1) generation of a valid solution (i.e., satisfies P), (2) preferred behaviors satisfied, (3) the number of observable and controllable events used, and (4) computation time, as shown in Table III. For *Vanilla* and OASIS, the controller synthesis procedure was given access to all the machine events as controllable and observable.

Summary. It can be seen that FORTIS is able to generate solutions that satisfy the preferred behaviors. On the other hand, *Vanilla* solves only the *Therapy* problem; OASIS solves the *Therapy* problem, the *Voting-1* and *Pump* problems but without satisfying the preferred behaviors, and does not solve[4] *Voting-2,3,4*.

The *Vanilla* approach and OASIS assume that all machine events are available for generating new designs. By comparison, FORTIS is capable of finding solutions that make use of fewer events (and thus, a lower cost) in the *Therapy*. In addition, it finds solutions with fewer controllable events but more observable events in the *Voting* and *Pump* problems, while the other two approaches either find no solutions or fail to retain the preferred behaviors.

On the other hand, FORTIS sometimes takes longer to generate a solution (for *Therapy* and *Pump*), since for optimality, it typically solves a larger number of synthesis instances than OASIS and *Vanilla* do. We believe that this is an acceptable trade-off between performance and quality of the solutions.

E. Threats to Validity

One potential threat to validity is that the deviations and preferred behaviors selected for our experiments might have introduced bias that enabled FORTIS to perform more efficiently. To mitigate this bias, the deviations for our case studies were derived from the existing literature. In particular, the deviations for the radiation therapy system were obtained from [8], and those for the electronic voting were from [12]. For the infusion pump, robustification was performed with the most general set of deviations, in that the resulting environment (E') allowed the user to perform actions in any order. In addition, we chose the preferred behaviors based on what we deemed to be common and important behaviors in these systems (e.g., the voter being able to navigate back and change their vote for a voting machine).

[4]We used a stronger safety property that the recorded vote should be the one that was selected by the corresponding voter, whereas in [12], it is only required that the vote not be changed by the official.

F. Discussion

Our experiment shows that FORTIS is able to generate a robustified design that (1) retains user-specfied preferred behaviors and (2) minimizes the cost of change, at a performance that is comparable to OASIS. In addition, unlike the two other approaches, FORTIS can generate the set of *all* Pareto-optimal solutions, which allows the developer to explore the trade-offs between the two qualities.

Vanilla can only restrict, but not extend, the machine behavior; thus, its ability to generate an optimal robustification is limited. FORTIS can extend the behavior by extending the controllability and observability of environmental events. OASIS does so by changing the sequence of events. However, such a reordering may prevent it from preserving the behavior of the original design (e.g., *Voting-1* and *Pump*) or sometimes result in an unusual design (e.g., in *Pump*, "starts dispensing" after the system "turns off").

On the other hand, by abstracting the machine and changing its event sequencing, OASIS can produce alternative designs that are not in the solution space of FORTIS. An approach that combines the event-based method of FORTIS with the abstraction-based strategy of OASIS may enable a more powerful robustification process, and is an interesting direction that we plan to investigate as future work.

VIII. Related Work

Model repair addresses the following problem: Given system M and property P where $M \not\models P$, finds a new M' such that $M' \models P$. Buccafurri et al. [27], Menezes et al. [28], Chatzieleftheriou et al. [29], and Ding et al. [30] present repair approaches for CTL, α-CTL, Kripke Modal Structure, and LTL, respectively. Our approach can be considered as a kind of model repair, although robustification addresses how to enhance the system design (M) to tolerate deviations in the environment (E), whereas the prior works do not make a distinction between M and E. Moreover, the existing works do not consider the cost of a repair or consider costs based on only the syntactic changes to the model (e.g., adding or removing transitions), whereas our approach considers multiple quality metrics that are semantic-based (i.e., behaviors preserved and events added to control the environment). Among this class of works, OASIS [12] is the closest to our work, for which we provide a more thorough comparison in Section VII.

Prior works have investigated synthesizing systems that are robust against environmental disturbances [31]–[35]. These works differ from our approach in that they rely on a notion of robustness that is *quantitative* in nature (e.g., a numerical amount by which an input deviates). In comparison, we adopt a *qualitative* definition of robustness from [8], which is applicable to the types of discrete deviations that are common in software systems (e.g., user omitting an action). Moreover, our deviation model generalizes the approach in [36], [37], where deviations are defined only in terms of additional transitions.

Control theory has also been applied in the context of self-adaptive systems [38] and run-time verification [39]–[41] to dynamically enforce system requirements. These run-time approaches typically assume a fixed sensing and actuating capability. By comparison, our work focuses on robustifying a system at *design time*, which gives developers flexibility to extend the sensing and actuating ability (by adding observable and controllable events, respectively).

Alrajeh et al. proposes an approach that leverages a learning technique to automatically adapt a system to changes in the environment [42]. Their approach targets adapting system *requirements* (specified as goal models [43]) to handle environmental changes, whereas our work involves modifying the system itself. However, in certain domains, it may also be possible to improve the robustness of the system by *weakening* a requirement [44] (i.e., given $M||E' \not\models P$, derive P' such that $M||E' \models P'$). We plan to study combining these two types of approaches as an integrative approach to robustifying a system.

D'Ippolito et. al proposes a *multi-tier control* approach to self-adaptation, where the developer provides a hierarchy of environment models that embody different levels of uncertainty, and at run time, the system dynamically switches between different controllers that best correspond to the current environment [45]. Their approach is similar to ours in that it also involves synthesizing different machines (M', M'',...) for different environments (E', E'',...). However, there are also some notable differences: (1) their approach aims to achieve *graceful degradation* by progressively weakening the system goal (i.e., property) under different environments (e.g., P', P'', ...) whereas our aim is to preserve the property, and (2) they do not specifically consider the relationship between a pair of machines (i.e., M and M') with respect to quality metrics.

IX. Discussions

We have introduced the notion of *robustification*, and an approach that leverages supervisory control to robustify a system design against possible deviations in the environment. Our approach can be used not only to find new, robust designs but also support design decisions based on trade-offs between the developer's preferences (i.e., what behavior the new design should retain and whether it is cost-effective).

In this paper, we evaluated our approach mainly in the domain of human-machine interfaces since the class of deviations that manifest as human errors have been well studied and codified into formal models [46], which is well-suited for our illustration. However, in general, our robustification technique can be applied to any domain where (1) the system and the environment can be modeled in LTSs and (2) deviations in the environment can be captured as additional transitions and states of the model. Other examples of such domains include network protocols [8], security protocols [47], and cyber-physical systems [36], [37].

Our approach relies on identification of relevant deviations in the environment, which typically requires domain knowledge (e.g., [48] in human factors) and cannot be fully automated. However, FORTIS has been integrated with the robustness analysis technique in [8], which can automatically generate and classify possible deviations into different categories (e.g., user omitting or repeating an action) and aid the process of deviation identification.

Currently, our approach focuses on safety properties only. In the future, we plan to explore the problem of robustification for liveness properties as well. In particular, during robustification, new behaviors may need to be added to the system (e.g., adding retries in a network protocol, instead of restricting its behavior as currently done with supervisory control), possibly leading to a much larger search space and requiring additional heuristics beyond those presented in this paper. In addition, we plan to explore a notion of robustness that is stochastic in nature (e.g., where an environment model E is specified as a Markov chain) and investigate the robustification problem under this setting.

Acknowledgments

This work was supported in part by the National Science Foundation award CCF-2144860, the NSA under Award No. H9823018D0008, and the Office of Naval Research under Award N00014172899. It was also supported by the CAMELOT project (reference POCI-01-0247-FEDER-045915) which is co-financed by the European Regional Development Fund and the Portuguese Foundation for Science and Technology under CMU Portugal. Any views, opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the organizations.

References

[1] M. Jackson, "The world and the machine," in *17th International Conference on Software Engineering (ICSE)*, 1995, pp. 283–292.
[2] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Software*, vol. 17, no. 3, pp. 37–43, 2000.
[3] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.
[4] D. Gage and J. McCormick, "We did nothing wrong: Why software quality matters," *Baseline Magazine*, 2004.
[5] H. W. Thimbleby, "Ignorance of interaction programming is killing people," *Interactions*, vol. 15, no. 5, pp. 52–57, 2008.
[6] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 399–314.
[7] S. Sun and K. Beznosov, "The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 378–390.
[8] C. Zhang, D. Garlan, and E. Kang, "A behavioral notion of robustness for software systems," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, p. 1–12.
[9] Y. Collette and P. Siarry, *Multiobjective Optimization: Principles and Case Studies*, ser. Decision Engineering. Springer Berlin Heidelberg, 2013.

[10] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 3rd ed. Springer, Cham, 2021.

[11] U.S. Attorney's Office Eastern District of Kentucky, "Clay county officials and residents convicted on racketeering and voter fraud charges," Mar 2010. [Online]. Available: https://archives.fbi.gov/archives/louisville/press-releases/2010/lo032510.htm

[12] T. T. Tun, A. Bennaceur, and B. Nuseibeh, "OASIS: Weakening user obligations for security-critical systems," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 113–124.

[13] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.

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

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

[16] J. Bergstra, A. Ponse, and S. Smolka, Eds., *Handbook of Process Algebra*. Amsterdam: Elsevier Science, 2001.

[17] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs, 2nd Edition*. London: Wiley, 2006.

[18] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012.

[19] J. N. Tsitsiklis, "On the control of discrete-event dynamical systems," in *26th IEEE Conference on Decision and Control*, vol. 26, 1987, pp. 419–422.

[20] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 179–190. [Online]. Available: https://doi.org/10.1145/75277.75293

[21] R. Su and W. M. Wonham, "Supervisor reduction for discrete-event systems," *Discrete Event Dynamic Systems*, vol. 14, no. 1, pp. 31–53, 2004.

[22] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica–an efficient tool for large-scale discrete event systems," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5794–5799, 2017, 20th IFAC World Congress.

[23] M. L. Bolton and E. J. Bass, "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*, 2011, pp. 1788–1794.

[24] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *Journal of Computer and System Sciences*, vol. 78, no. 3, pp. 911–938, 2012, in Commemoration of Amir Pnueli.

[25] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *ACM Symp. POPL*, 1989.

[26] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Proceedings of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 96–106.

[27] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing model checking in verification by ai techniques," *Artificial Intelligence*, vol. 112, no. 1, pp. 57–104, 1999.

[28] M. V. de Menezes, S. do Lago Pereira, and L. N. de Barros, "System design modification with actions," in *Advances in Artificial Intelligence – SBIA 2010*, A. C. da Rocha Costa, R. M. Vicari, and F. Tonidandel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 31–40.

[29] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros, "Abstract model repair," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 341–355.

[30] Y. Ding and Y. Zhang, "A logic approach for LTL system modification," in *Foundations of Intelligent Systems*, M.-S. Hacid, N. V. Murray, Z. W. Raś, and S. Tsumoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 435–444.

[31] 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*, 2014, pp. 431–443.

[32] 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*, 2011, pp. 176–185.

[33] P. Tabuada, A. Balkan, S. Y. Caliskan, Y. Shoukry, and R. Majumdar, "Input-output robustness for discrete systems," in *International Conference on Embedded Software, *EMSOFT)*. ACM, 2012, pp. 217–226.

[34] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann, "Robustness in the presence of liveness," in *Computer Aided Verification (CAV)*, vol. 6174. Springer, 2010, pp. 410–424.

[35] T. Kobayashi, R. Salay, I. Hasuo, K. Czarnecki, F. Ishikawa, and S. Katsumata, "Robustifying controller specifications of cyber-physical systems against perceptual uncertainty," in *International Symposium on NASA Formal Methods (NFM)*, 2021, pp. 198–213.

[36] U. Topcu, N. Ozay, J. Liu, and R. M. Murray, "On synthesizing robust discrete controllers under modeling uncertainty," in *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '12. Association for Computing Machinery, 2012, p. 85–94.

[37] R. Meira-Góes, E. Kang, S. Lafortune, and S. Tripakis, "On tolerance of discrete systems with respect to transition perturbations," *arXiv:2110.04200 [eess.SY]*, 2021.

[38] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Camara, R. Calinescu, M. B. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, H. A. Müller, R. Rouvoy, C. M. F. Rubira, E. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. M. Villegas, T. Vogel, and F. Zambonelli, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds. Cham: Springer International Publishing, 2017, pp. 3–30.

[39] A. Easwaran, S. Kannan, and O. Sokolsky, "Steering of discrete event systems: Control theory approach," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 4, pp. 21–39, 2006, proceedings of the Fifth Workshop on Runtime Verification (RV 2005).

[40] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," *Formal Methods in System Design*, vol. 38, no. 3, pp. 223–262, 2011.

[41] Y. Falcone, J.-C. Fernandez, and L. Mounier, "What can you verify and enforce at runtime?" *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 349–382, 2012.

[42] D. Alrajeh, A. Cailliau, and A. van Lamsweerde, "Adapting requirements models to varying environments," in *International Conference on Software Engineering (ICSE)*. ACM, 2020, pp. 50–61.

[43] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[44] S. Chu, E. Shedden, C. Zhang, R. Meira-Góes, G. A. Moreno, D. Garlan, and E. Kang, "Runtime resolution of feature interactions through adaptive requirement weakening," in *Proceedings of the 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '23, 2023.

[45] N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: multi-tier control for adaptive systems," in *36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 688–699.

[46] E. Hollnagel, "The phenotype of erroneous actions," *International Journal of Man-Machine Studies*, vol. 39, no. 1, pp. 1–32, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737383710515

[47] D. Basin, S. Radomirovic, and L. Schmid, "Modeling human errors in security protocols," in *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, 2016, pp. 325–340.

[48] 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*, vol. 70, no. 11, pp. 888–906, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581912000997