Fortis: A Tool for Analysis and Repair of Robust Software Systems

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

Ian Dardik
Carnegie Mellon University
Pittsburgh, PA USA
idardik@andrew.cmu.edu

Rômulo Meira-Góes
The Pennsylvania State University
State College, PA USA
romulo@psu.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*—This paper presents Fortis, a tool for automated, formal analysis and repair of robust discrete systems. Given a system model, an environment model, and a safety property, the tool can be used to automatically compute *robustness* as the amount of *deviations* in the environment under which the system can continue to guarantee the property. In addition, Fortis enables automated repair of a given system to improve its robustness against a set of intolerable deviations through a process called *robustification*. With these techniques, Fortis enables a new process for developing *robust-by-design* systems. The paper presents the overall design of Fortis as well as the key details behind the robustness analysis and robustification techniques. The applicability and performance of Fortis are illustrated through experimental results over a set of case study systems, including a radiation therapy system, an electronic voting machine, network protocols, and a transportation fare system.**

I. Introduction

Typical verification tasks involve the following question: Given a model of a system (M) and an environment (E), does the system satisfy a desired property (P) under the environment (i.e., $M \| E \models P$)? The model E here captures various assumptions that the system makes about its environment to establish P. For example, such assumptions may state that a human operator in a safety-critical system (e.g., a medical device) performs a set of actions in an expected order, or that the underlying network in a distributed system is reliable and delivers messages correctly from one node to another.

In practice, once the system is deployed, the actual environment may *deviate* from this model, either due to modeling errors, faults, or natural changes in the environment. For example, the operator may inadvertently commit errors from time to time (e.g., omitting or repeating an action); the network might experience an unexpected disruption and fail to guarantee reliable delivery (e.g., losing or duplicating messages). Ideally, a system that is *robust* would continue to ensure its most critical properties even under possible deviations in the environment.

In this paper, we present Fortis[1], a tool for formal analysis and repair of robust software systems. Our tool is based on a formal definition of robustness for discrete systems introduced in our prior work [1]: Given system M, environment E (both specified as a labeled transition system (LTS)) and safety property P, the *robustness* of the system, denoted Δ, is defined as the set of all possible deviations in E under which M continues to satisfy P. More specifically, Δ consists of traces that do not belong to the trace set of E, capturing additional environmental behaviors beyond the normative environment. For example, if M describes the design of a medical system (e.g., radiation therapy system), E the expected behavior of the human operator, and P a safety requirement (e.g., "Patients should be protected from radiation overdose"), Δ would represent the set of possible operator errors under which the system can still ensure safety. Conceptually, Δ represents the safe operating envelope of the system: As long as the environmental deviations remain within this envelope, the system can guarantee P.

Building on this definition, Fortis provides various types of analysis tasks to support rigorous design and analysis of robust systems. First, given M, E, and P, Fortis can be used to automatically compute Δ as a qualitative measure of the system robustness. Our tool can also be used to compute deviations that lie outside of Δ (which we call *intolerable deviations*), showing how P may be violated when the environment moves outside of the operating envelope. In addition, given a pair of alternative system designs, M_1 and M_2, Fortis can also be used to formally compare the two with respect to their robustness (i.e., compute a set of deviations that one design can tolerate but the other cannot).

Once the above analysis reveals that the given system is not robust against certain deviations, the developer may wish to modify M to further improve its robustness. To support this task, Fortis also provides a type of system repair called *robustification* [2]: Given M, P, E, and a set of intolerable

[1]https://github.com/cmu-soda/Fortis

https://doi.org/

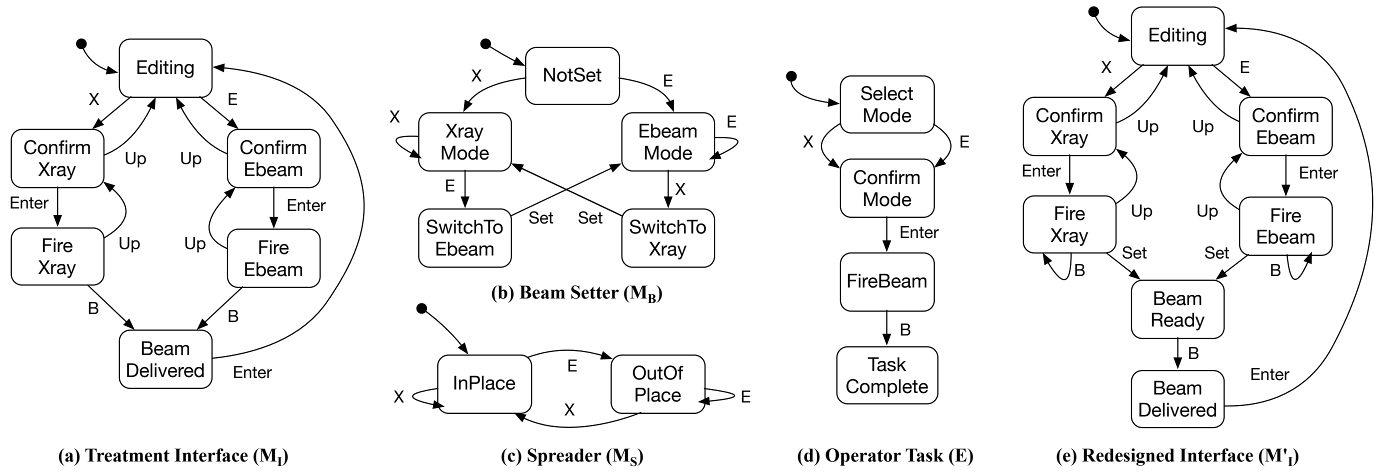

Fig. 1. Labeled transition systems for a radiation therapy system ($M = M_I \| M_B \| M_S$).

deviations, $\bar{\delta}$ such that $M \| E' \not\models P$ (where $E' = E \oplus \bar{\delta}$ is the deviated environment), the goal is to synthesize a more robust system, M', such that $M' \| E' \models P$.

As far as we are aware, Fortis is the first tool that is capable of providing the types of robustness analysis and repair described above. Compared to the prototypes presented in our prior publications, additional engineering effort has been taken to integrate them into a uniform framework. Together with these techniques, we believe Fortis enables a new methodology for developing *robust-by-design* systems. Developers can start with an initial design that guarantees its desired properties under the normative environment. Then, they can use Fortis to understand the robustness of the initial design and generate the deviations that it cannot tolerate. Finally, developers can decide which of these deviations the system should be able to tolerate, and use Fortis to automatically generate a more robust design. Developers may iterate this process for multiple times until a satisfactory design is met.

To evaluate the tool, it has been applied to a wide range of case study systems, including a radiation therapy system (similar to the well-known Therac-25 system [3]), an electronic voting system, network protocols, an infusion pump, and a fare collection protocol used in a public transportation system. Our experiments show that Fortis can automatically compute robustness for complex system models under several seconds, and also synthesize repairs for most of the case studies under a set timeout.

The rest of the paper is structured as follows. We first demonstrate use cases of Fortis using an example involving a radiation therapy system (Section II). We then present an overview of the tool architecture (Section III) and describe the key details of the analysis and robustification techniques (Section IV). Next, we illustrate the applicability and performance of Fortis over the case studies (Section V). We conclude with the related work (Section VI) and a discussion of limitations and possible extensions (Section VII).

II. Motivating Example

Consider a radiation therapy machine similar to the well-known Therac-25 machine [3]. Figure 1 shows the labeled transition systems of the main system components, including (a) *Treatment Interface (M_I)*, which allows the operator to choose the radiation mode and fire the beam, (b) *Beam Setter (M_B)*, which switches between the two radiation modes (Electron and X-ray), and (c) *Spreader (M_S)*, which is inserted during the X-ray mode to attenuate the effect of the high-power X-ray beam and limit possible overdose (X-ray delivers roughly 100 times higher level of current than the Electron beam). The overall system is the composition of the three components, i.e., $M = M_I || M_B || M_S$.

An important safety requirement for the system is that the spreader must be in place when the beam is delivered in the X-ray mode. This requirement can be formally defined in linear temporal logic [4] as: $\mathbf{G}(BeamDelivered \land XrayMode \Rightarrow InPlace)$. Furthermore, the task to be carried out by the operator is specified as an environment model (E) in Figure 1(d): In the normal treatment process, the therapist selects the correct mode for a given patient by pressing either *X* or *E*, confirms the mode by pressing *Enter*, and finally initiates the therapy by pressing *B*.

Applying a verification technique such as model checking [5] would show that the above system satisfies the safety property under the normative operator behavior, i.e., $M || E \models P$. Beyond this standard verification task, Fortis offers the following additional tasks:

a) Computing robustness: In addition to stating that M satisfies P under E, Fortis can be used to generate the set of all deviations (i.e., environmental traces that do not belong to the behavior of E) under which the system can still guarantee P. This set captures the overall *robustness* of the system, and can aid the developer in understanding the system's ability in handling deviations in the environment. In the radiation therapy system example, one of the deviations that Fortis generates is $\langle X, B \rangle$, which depicts the operator omitting to confirm the radiation mode before firing; the system guarantees P even under a new environment E' where this trace has been added as an additional behavior.

b) Generating intolerable deviations: Complementary to the previous analysis, Fortis can also be used to generate deviations for which the system is not able to guarantee P. In the radiation therapy example, one such deviation is $\langle X, Up, E, Enter, B\rangle$, depicting a scenario where the operator accidentally selects the X-ray mode and corrects the mistake by pressing *Up* and then *E*. When the operator presses *B* to fire the beam, the beam setter might still be in transition from X-ray to the Electron mode (in state SwitchToEbeam in M_B, Figure 1(b)) while the spreader is out of place, causing P to be violated. The output from this analysis can help identify parts of the system design that can be made more robust.

c) Comparing designs: Given two different versions of a system (e.g., Therac-25 and its predecessor, Therac-20, which was known to be safer thanks to an additional hardware interlock that was subsequently removed [3]), Fortis can also be used to formally compare the two with respect to their robustness. For example, Fortis would show that Therac-20 is strictly robust than Therac-25 by generating a deviation (e.g., $\langle X, Up, E, Enter, B\rangle$) for which the former can guarantee P while the latter cannot.

d) Robustifying the system: Fortis can be used to automatically improve an existing design through a process called *robustification*: If M is not robust against some given set of deviations ($\bar{\delta}$), generate a more robust design, M', that can satisfy P under $\bar{\delta}$ (i.e., M' is strictly more robust than M). For example, given deviation $\langle X, Up, E, Enter, B\rangle$, Fortis automatically synthesizes $M' = M_I' \| M_B \| M_S$; this new design, M', guarantees the safety property by preventing the operator from firing the beam until the mode switch has completed (i.e., state BeamReady), as shown in Figure 1(e).

III. Overview of Fortis

Figure 2 shows the overall architecture of Fortis. Given LTS-based specifications of machine M, its normative environment E, and safety property P as input, and the *Model Parser* compiles them into our internal data structure for LTS. Depending on the type of task that the user wishes to perform, the input models are then passed onto *Robustness Analysis* or *Robustification*.

a) Robustness Analysis: To compute robustness, we first generate the *weakest assumption* of M with respect to environment E and property P. In assume-guarantee style of reasoning [6], the weakest assumption captures the largest possible environmental behavior under which the machine satisfies a given property. Then, robustness, denoted Δ, is computed as the set of traces that are in the weakest assumption but not in the expected environment E. In general, Δ may be infinite, and not in a form that is easily comprehensible by the user. Thus, we partition Δ into a finite set of *equivalence classes*, each of which contains traces that describe the same type of deviation (e.g., the type of user error where one omits an action), and sample *representative traces* from those classes. Finally, if a *deviation model* is provided, we use it to generate *explanations* that describe how the environment may deviate from its expected behavior in a particular way. The final output is a set of pairs of a representative trace and its corresponding explanation. Section IV-A describes some of these steps in more detail.

b) Robustification: To robustify a machine, the user specifies a set of intolerable deviations ($\bar{\delta}$), which are then used to transform the normative environment (E) into a deviated environment (E'). Note that the robustness analysis module can be used to generate *all* the intolerable deviations $\bar{\Delta}$, which can help designers identify the undesirable deviations of interest. Optionally, the user can also specify the *preferred behaviors* (i.e., execution traces) expected to be retained in the new design and the *costs* to control and observe events, to generate repairs that are optimal with respect to these two metrics.

Internally, Fortis leverages supervisory control theory [7] to synthesize new designs; in particular, it currently uses the state-of-the-art controller synthesizer called Supremica [8]. To find optimal repairs, the *Design Optimizer* repeatedly invokes the synthesizer for different combinations of the preferred behaviors and the event costs, exploring the multi-objective space to generate *Pareto-optimal* solutions [9] as the final output of the tool (more details in Section IV-B).

c) Bridging Robustness and Robustification: In addition to the implementation of the two techniques proposed in our prior work, Fortis also provides an integration of them that bridges the gap to close the loop for robust-by-design development process, represented as the dashed line connecting the two modules in Figure 2. Specifically, it enables the user to first compute the robust deviations (Δ) and intolerable deviations ($\bar{\Delta}$); and after the user has decided on the deviations that they want the system to be robust against, it can then generate the corresponding deviated environment model (E'). Finally, this deviated model can be used as the input to robustify the system design.

IV. Analysis and Robustification Methods

In this section, we provide key implementation details behind the robustness analysis, robustification techniques, and their integration in Fortis.

A. Robustness Analysis

a) Robustness Computation: In our definition [1], the *robustness* of machine M with respect to environment E and property P (denoted $\Delta(M, E, P)$) is defined as the *maximal* set of traces that do not cause a property violation and do not belong to the trace set of the normative environment. This set is computed by calculating the difference between (1) the *weakest assumption* of M w.r.t. E and P, and (2) the environment E, i.e., $\Delta(M, E, P) = beh(W_{M,E,P}) \setminus beh(E)$, where $beh(\cdot)$ is the set of all traces of a given LTS and $W_{M,E,P}$ is the weakest assumption. Specifically, Fortis uses the approach developed by Giannakopoulou et al. [10] to compute the weakest assumption as an LTS.

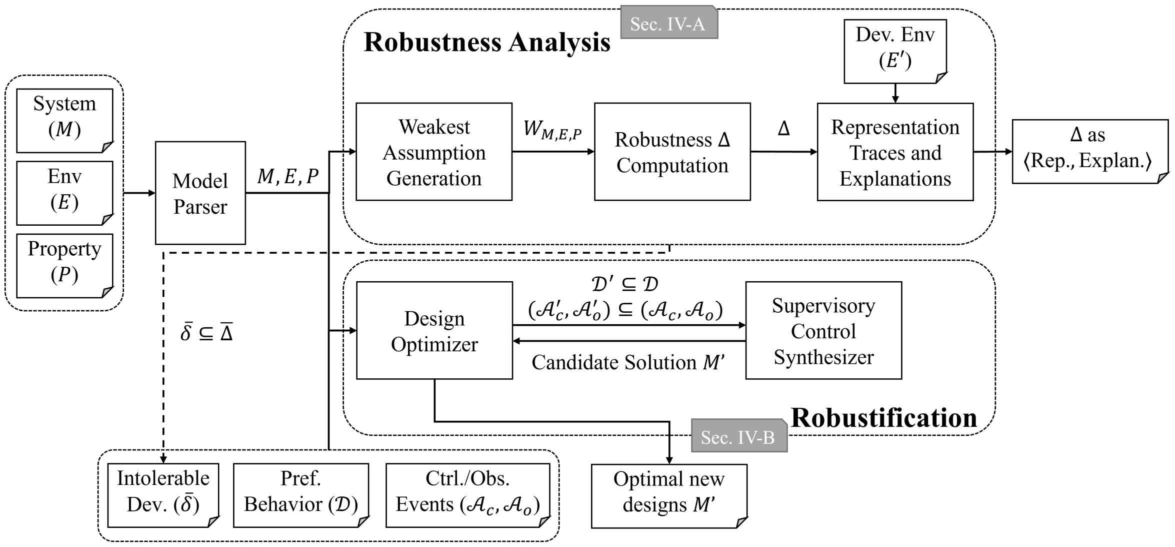
Robustness Analysis
Sec. IV-A
Dev. Env (E')
System (M)
Env (E)
Property (P)
Model Parser
M, E, P
Weakest Assumption Generation
$W_{M,E,P}$
Robustness Δ Computation
Δ
Representation Traces and Explanations
Δ as ⟨Rep., Explan.⟩
$\bar{\delta} \subseteq \bar{\Delta}$
Design Optimizer
$\mathcal{D}' \subseteq \mathcal{D}$
$(\mathcal{A}'_c, \mathcal{A}'_o) \subseteq (\mathcal{A}_c, \mathcal{A}_o)$
Candidate Solution M'
Supervisory Control Synthesizer
Robustification
Sec. IV-B
Intolerable Dev. ($\bar{\delta}$)
Pref. Behavior ($\mathcal{D}$)
Ctrl./Obs. Events ($\mathcal{A}_c, \mathcal{A}_o$)
Optimal new designs M'

Fig. 2. The architecture of Fortis.

b) Robustness Comparison: Similar to robustness computation, given two designs M_1 and M_2, Fortis leverages their weakest assumptions to compare their robustness. In particular, given machine M_1, M_2, and the same environment E and property P, the robustness comparison is achieved by:

$$\Delta(M_1) - \Delta(M_2) = beh(W_{M_1,E,P}) \setminus beh(W_{M_2,E,P})$$

where $W_{M_1,E,P}$ and $W_{M_2,E,P}$ are the weakest assumptions for M_1 and M_2 with respect to E and P, respectively.

c) Robustness Representation: In general, the set of traces that represent Δ may be infinite, and an LTS-based representation may not be readily comprehensible by the user, even for relatively simple models like the radiation therapy machine. To address this, Fortis generates a succinct and finite representation of Δ. It groups the traces in Δ into a finite set of *equivalence classes* $\Pi_{s,a}$, where s is a state that directly leads to a violation of E by taking the transition a (note that Δ contain traces that belong to $W_{M,E,P}$ but not in E). Therefore, $\Pi_{s,a}$ describes a class of robust traces that share the same normative behaviors in E that all end in state s and deviate from E by the same event a.

Finally, from each equivalence class, we sample a single *representative* trace that represents this class. In particular, the representative trace for $\Pi_{s,a}$ is generated by finding the shortest trace in $W_{M,E,P}$ from the initial state to state s and then appending event a to it. For example, in the radiation therapy machine, $\langle X, B\rangle$ represents the equivalence class of behaviors that deviate in action B from state s_X, where s_X is the state reached by the normative behavior $\langle X\rangle$ and B is the first deviated action leading to a trace not defined in E. The final output from this step is a finite set of representative traces that describe different types of environmental deviations.

d) Deviation Explanation: Representative traces describe how the environment deviates from the expected behavior as *observed* by machine M. However, they do not describe the internal *faults* within the environment that cause these deviations in the first place. For example, it is unclear what kind of faults cause the deviation of $\langle X, B\rangle$. If a *deviation model* that contains these internal events is provided by the user, Fortis uses it to generate an *explanation* of how a particular deviation arises due to an internal behavior of the environment. In particular, given deviation model D and representative trace σ, an explanation is generated by finding trace σ_{exp} in D that is equivalent to σ when projected over the observable events in M but contains additional faulty events.

A deviation model is created by augmenting the normative environmental model E with additional transitions on faulty events. For example, a deviation model for the radiation therapy machine may specify that from state ConfirmMode in E (Figure 1(d)), the operator might commit a type of error called *omission error* [11], i.e., omitting *Enter* and pressing *B*; this would be specified as an additional transition from ConfirmMode to FireBeam on an internal faulty event *Omission*. Then, Fortis would generate an explanation for the representative trace $\langle X, B\rangle$ as $\langle X, Omission, B\rangle$. In [1], it is further described how a deviation model can be automatically generated by applying domain-specific patterns of deviations (e.g., patterns of common human errors [11]) to the normative environment E.

B. Robustification

Fortis finds not just any solution to the robustification problem, but *optimal* repairs of M. In particular, it attempts to optimize two different quality metrics: (1) the amount of behaviors retained from M to new design M' and (2) the cost of modifications needed to achieve M'. Specifically, for (1), we introduce the notion of *preferred behaviors* $\mathcal{D}$, which are specified as traces and represent operational scenarios that the user wishes the new design to retain. For (2), we introduce the notion of *controllable* $\mathcal{A}_c$ and *observable* $\mathcal{A}_o$ events that indicate whether M' can control and observe additional events, respectively, for the purpose of robustification. In addition, the user can optionally assign a cost to these events, to distinguish events that are more costly to control or observe.

These two metrics lead to conflicting objectives: With additional controllable and observable events, the new design can preserve more behaviors but the modification cost also increases. Thus, finding optimal repairs is a multi-objective optimization problem, where the goal is to generate a set of Paret-optimal solutions [9]. Fortis implements a novel algorithm to generate such solutions, using controller synthesis as a primitive operation. At high-level, the Design Optimizer first attempts to synthesize a controller that preserves the maximum number of preferred behaviors ($\mathcal{D}_{max} \subseteq \mathcal{D}$) with all controllable and observable events ($\mathcal{A}_c$ and $\mathcal{A}_o$). Then, the optimizer incrementally removes elements from $\mathcal{D}_{max}$ to find solutions with a lower modification cost. In particular, given a particular subset of preferred behaviors $\mathcal{D}' \subseteq \mathcal{D}_{max}$, it searches for a controller that uses a minimal subset of the controllable and observable events.

For the radiation therapy system, one possible repair that Fortis generates results in the machine simply disabling action Up in state ConfirmXray, to prevent the operator error from occurring in the first place. While this solution technically achieves the safety property, it is arguably undesirable, since it prevents the operator from switching the radiation mode. To rule out such solutions, the Fortis user may specify preferred behaviors as traces $\langle X, Up \rangle$, $\langle E, Up \rangle$, stipulating that the operator should be able to select Up after X or E to change the mode. In addition, the user may assign a cost to event Set to reflect the cost of making it controllable or observable by the interface (M_I) or the spreader component (M_S).

Given the additional inputs on preferred behaviors and costs, as an alternative solution, Fortis generates a repair like the one in Figure 1(e); this solution retains the ability for mode switching, while being more costly since it involves the system observing an additional event, Set, to synchronize on the completion of mode switching. The user of Fortis can examine these alternative solutions and select the final one depending on the trade-offs between the two metrics that they are willing to make.

C. Integration

Compared to the prototype implementations in our prior work, Fortis integrates the two techniques into a uniform framework, i.e., they accept the same formats of input model files and can produce results in a uniform manner. Also, we leverage Automatalib [12], a well-maintained open source library for transition systems, to provide a common interface for our internal representations for LTS, which implements the CompactFSM data structure for efficient model manipulations. Moreover, we replace some algorithm implementations with more efficient data structures like BitSet (for NFA to DFA conversion) to further improve the performance of the computation process.

Fortis also bridges the gap between the robustness computation and robustification. In particular, after generating the representative traces for robustness and intolerable deviations, and their explanations given a deviation model D, the user can select the deviations δ' that the new design should be robust against. For example, $\langle X, Commission, Up, E, Enter, B \rangle$ is the explanation of an intolerable deviation for the radiation therapy machine, which could be included in δ' as the user wishes the new design to be robust against it.

To robustify the system against δ', Fortis can automatically generate a new deviation model D' such that it only includes the deviations in δ' (as the original D may include more than one kinds of human errors, e.g., *commission error*, *omission error*, or *repetition error*). Then, the user can use D' as the input to the robustification process to synthesize the new design M'.

V. Experiments

In this section, we illustrate the applicability and performance of Fortis through experiments over a set of case studies.

A. Implementation and Usage

Fortis leverages Automatalib [12] and LTSA [13] for model specification and manipulation, and Supremica [8] for supervisory control synthesis. It supports commonly used specification languages such as AUT and FSM (through Automatalib) and FSP (language used by LTSA) to specify and output system models.

Currently, Fortis implements a command-line interface. Users provide system and property specifications through command-line arguments or a JSON configuration file, and the tool produces computation results into command-line outputs. For example, Figure 3 shows the input and output for robustness computation and robustification of the radiation therapy example.

B. Case Studies

a) Voting Machine: We consider a simplified design of an electronic voting system (called ES&S iVotronic, described in more detail in [14]) that was used in several state-wide elections in the US. This system is particularly interesting to study from the perspective of robustness, as it was susceptible to an election fraud involving malicious election officials [15]. In this machine, for the last step of a voting process, the voters were asked to confirm their vote by pressing the *confirm* button. However, some voters would inadvertently forget to do so before exiting the voting booth (committing what is generally called *post-completion error* [16]). This would then allow a malicious official to enter the booth, press *back*, and then modify the vote to their liking. In our model, the property to guarantee is that the machine should record each voter's selection exactly as made by that voter, and the intolerable deviation of interest is voters omitting the confirmation step. We successfully used Fortis to generate alternative designs that would prevent malicious officials from modifying the vote (e.g., keeping track of who enters the booth and disabling *confirm* if an official is in the booth).

```
# Run robustness computation
java -jar fortis.jar robustness -s sys.lts -e env0.lts -p p.lts -d env.lts
[INFO] BaseCalculator - Generating robust behavior representation traces by equivalence classes...
[INFO] BaseCalculator - Generating the weakest assumption...
[INFO] SubsetConstructionGenerator - Compose System and Property...
[INFO] SubsetConstructionGenerator - System: #states = 22, #transitions: 44
[INFO] SubsetConstructionGenerator - S||P: #states = 20, #transitions: 40
[INFO] SubsetConstructionGenerator - Pruning and determinising the model...
[INFO] Robustness - Total time: 00:00:00:021
[INFO] Robustness - Equivalence class 'EquivClass(s=1, a=up)':
[INFO] Robustness -     RepTrace(word=x,up, deadlock=false) => x,up
[INFO] Robustness - Equivalence class 'EquivClass(s=8, a=up)':
[INFO] Robustness -     RepTrace(word=e,up, deadlock=false) => e,up
...

# Run robustification
java -jar fortis.jar robustify config-fast.json
...
[INFO] SolutionIterator - Start iteration 1...
[INFO] SolutionIterator - Try to weaken the preferred behavior by one of the 0 behavior sets:
[INFO] SolutionIterator - This iteration completes, time: 00:00:00:093
[INFO] SolutionIterator - Number of controller synthesis process invoked: 5
[INFO] SolutionIterator - New solution found:
[INFO] SolutionIterator -       Size of the controller: 63 states and 130 transitions
[INFO] SolutionIterator -       Number of controllable events: 4
[INFO] SolutionIterator -       Controllable: [enter, fire_ebeam, fire_xray, setMode]
[INFO] SolutionIterator -       Number of observable events: 8
[INFO] SolutionIterator -       Observable: [b, e, enter, fire_ebeam, fire_xray, setMode, up, x]
[INFO] SolutionIterator -       Number of preferred behavior: 4
[INFO] SolutionIterator -       Preferred Behavior:
[INFO] SolutionIterator -               x,up,e,enter,b
[INFO] SolutionIterator -               e,up,x,enter,b
[INFO] SolutionIterator -               x,enter,up,up,e,enter,b
[INFO] SolutionIterator -               e,enter,up,up,x,enter,b
[INFO] SolutionIterator - Utility Preferred Behavior: 48
[INFO] SolutionIterator - Utility Cost: -1
...
```

Fig. 3. Fortis' input and ouput for robustness computation and robustification of the radiation therapy system. For robustness computation, the log indicates two of the representative traces found by Fortis, i.e., $\langle x, up\rangle$ and $\langle e, up\rangle$. For robustification, the log indicates one redesign found by Fortis where all preferred behaviors are satisfied under the events given in the solution to be controlled and observed. The concrete redesign model is written to a model specification file in the AUT format.

b) Network Protocols: Consider the problem of transmitting a sequence of messages between a pair of nodes (*sender* and *receiver*) in a specific order. We consider two protocols for network communication: (1) A naive protocol where the sender assumes a perfectly reliable communication channel, and (2) the Alternate Bit Protocol (ABP) [17], which is designed to guarantee integrity of messages over unreliable channels (e.g., message loss or duplication). The normative environment (E) here is the reliable channel, which relays (1) a message from the sender to the receiver and (2) then an acknowledgement from the receiver back to the sender. Our notion of deviations can be used to capture different ways in which an unreliable channel might behave, such as reordering (e.g., from expected trace $t = \langle msg_1, msg_2\rangle$ to deviation $t' = \langle msg_2, msg_1\rangle$), losing ($t' = \langle msg_2\rangle$), or duplicating messages ($t' = \langle msg_1, msg_1, msg_2\rangle$). In particular, we used Fortis to formally compare the robustness of the two protocols, to compute faults in the channel that ABP can handle but the naive one cannot.

c) Oyster: We consider the *Oyster* card fare collection protocol used in public transportation in London, UK (described in [18]). In this system, the user taps their card on the entry gate at the beginning of their journey and on the exit gate at the end. The protocol also allows the user to pay their fare through other means such as credit cards and mobile payments. In the normative case, the user chooses the appropriate method of payment, and taps in and out with the same method. The property of interest here is avoiding *card collision*, where two different methods of payment are used in the same journey. For instance, the user may tap the Oyster card at the entry gate but then (by mistake) use their mobile phone at the exit, possibly being charged a higher fare than required.

d) Infusion Pump: We model an infusion pump machine for dispensing medication to patients through tube lines [19]. The machine also includes a built-in battery that activates when the power cable is unplugged, and an alarm that goes off when the battery is low. Normally, the operator plugs in the device, configures the medication dose and starts the dispensation. Deviations here correspond to operator errors or power loss. In particular, if the cable is accidentally unplugged and battery runs out during dispensation continues, this might cause serious medical accidents, such as overdose. Thus, the property to be guaranteed here is that *if the machine loses power, it should immediately stop any on-going dispensation*. This case study is the most complex out of the ones that we have done so far and is intended to demonstrate the scalability of Fortis.

C. Experimental Results

For each of the above case studies, we used Fortis to (1) compute the robustness of the system and (2) synthesize

TABLE I
EVALUATION RESULTS. ALL PROBLEMS WERE RUN ON A LINUX MACHINE WITH A 3.6GHZ CPU AND 24GB MEMORY UNDER A 30 MINUTE TIMEOUT.

	Robustness Computation[*]		**Robustification**[†]						
						Naive		With heuristics	
	$\|M\|\|P\|$	Time (s)	$\|\mathcal{D}\|$	$\|\mathcal{A}\|$	$\|M\|\|E'\|$	#Syn.	Time (s)	#Syn.	Time (s)
Therapy	20	0.025	4	5	21	32	0.812	6	0.469
Naive	41	0.029	2	8	14	1	0.226	1	0.242
ABP[‡]	23	0.033	-	-	-	-	-	-	-
Voting-1[**]	53	0.033	1	13	12	2,576	24.100	9	0.507
Voting-2	277	0.066	1	23	31	-	T/O	16	1.908
Voting-3	821	0.106	1	32	44	-	T/O	21	20.172
Voting-4	1,829	0.188	1	41	57	-	T/O	-	T/O
Pump-1	163	0.036	2	12	104	2,304	59.584	13	1.129
Pump-2	1,679	0.149	4	16	760	-	T/O	17	10.817
Pump-3	19,435	1.227	6	20	6,248	-	T/O	21	457.839
Oyster	1,729	0.280	2	4	900	16	1.799	1	0.686

[*] $|M||P|$ is the number of states of the composition of machine M and property P, and the worst-case complexity of the computation is $O(2^{|M||P|})$.
[**] In Voting-n, n represents the number of voters and officials in the system; similarly, n in Pump-n is the number of the dispensation lines connected to the pump.
[†] $|\mathcal{D}|$ is the number of preferred behaviors, $|\mathcal{A}|$ the number of controllable and observable events with cost, $|M||E'|$ the number of states of machine M composed with deviated environment E'. The size of the search space is approximately $O(2^{|\mathcal{D}|+|\mathcal{A}|+|M||E'|})$. #Syn. is the number of calls to the controller synthesizer.
[‡] Robustification is not applicable to ABP as it already satisfies P under the given deviations.

robustified designs against a given set of intolerable deviations. Table I summarizes the results from the experiments. For robustness computation, although the worst-case complexity is exponential to the size of the machine M and property P. i.e., $O(2^{|M||P|})$, Fortis can efficiently compute robustness even for a very large model like *Pump-3* with 19,435 states in 1.227 seconds.

On the other hand, robustification is a much more complex problem to solve with a much larger search space, where the worst case complexity is exponential to the number of states of machine M and deviated environment E', plus the number of preferred behaviors $\mathcal{D}$ and controllable/observable events $\mathcal{A}$, i.e., $O(2^{|\mathcal{D}|+|\mathcal{A}|+|M||E'|})$. For example, the worst-case complexity for robustifying Pump-2 is about 6x10^{234}.

In addition, we found that controller synthesis is often the bottleneck. The time to solve one synthesis instance grows quickly with the increasing size of the system. Moreover, for the same problem, the synthesis becomes harder to solve when fewer controllable and observable events are provided (while minimizing the cost). Compared to naively searching solutions with brute-force (shown under the Naive columns), Fortis tackles the performance issue by introducing several search heuristics for pruning the search space and reducing the number of synthesis calls, which are described in more detail in [2]. While we believe that Fortis performs reasonably well on robustification of complex models, we plan to explore alternative synthesis techniques (such GR(1) synthesis [20], [21] or constraint-based methods [22]) to further improve its performance.

VI. RELATED WORK

Techniques for reasoning about system robustness have been investigated in prior works [23], [24], [25], [26], [27]. Most of these works adopt a *quantitative* notion of robustness (e.g., given bounded perturbations in its input, a robust system should ensure bounded changes in the output), while we use a definition that is *qualitative* (i.e., additional behaviors that a deviating environment may exhibit). We believe that the two types of definitions are complementary: A quantitative notion is well-suited for capturing numerical deviations in physical or hybrid systems (e.g., a sensor noise), while our approach is suitable for capturing deviations that occur in discrete systems (e.g., human operator errors). In addition, our notion of deviations generalizes the one in [28], [29], where deviations are defined as additional transitions that may be introduced into the environment.

Evrostos [30] is a tool for model checking systems against rLTL [31], a variant of LTL that captures robustness. rLTL enables specifications stipulating that "small" violations in the environmental assumption should result in proportionally "small" violations in the system guarantee. In particular, rLTL leverages a multi-valued semantics to capture different levels of property violation (e.g., given an expected property of form $\mathbf{G}\psi$, one possible weaker variant is $\mathbf{F}(\mathbf{G}\psi)$). Besides the difference in the definitions of robustness used, Evrostos and Fortis differ in their goals: The former is used to specify and verify robustness as a specification, while Fortis is used to extract robustness as a property of the system. However, rLTL could potentially be used to characterize certain types of environmental deviations that are temporal in nature.

Robustification in Fortis also shares similarities with *model repair* techniques [32], [33], [34], [35], [14]. Among these, the closest work to our approach is OASIS [14], which also leverages controller synthesis to revise a machine to satisfy a security property in a deviated environment. One major difference is that Fortis uses semantic-based metrics (i.e., preserved behaviors and costs of events) to qualify solutions, whereas these works do not take into account the cost of changes (e.g., OASIS), or consider only syntactic changes to the model (e.g., the number of modified transitions and states).

VII. Conclusions and Future Extensions

We have presented Fortis, an automated tool for formal analysis and repair of robust discrete systems. The tool supports a rigorous methodology for designing robust systems, where the developer starts with an initial design and a normative environment, and then iteratively improve its robustness by identifying undesirable environmental deviations and robustifying the system against them. As far as we know, Fortis is the first tool that provides these types of robustness analyses and repair by offering a seamless workflow packaged into a single tool.

There are a number of further tool extensions that we plan to explore. Currently, Fortis supports safety properties only. Adding liveness support will involve new theoretical extensions; in particular, during robustification, new behaviors may need to be added to the system (instead of restricting its behavior, as currently done for safety), possibly leading to a much larger search space. In addition, we also plan to add a capability for *synthesizing* a robust system (M) from scratch instead of modifying an existing one (M to M'). Finally, as discussed in Section V, we will explore alternative methods for controller synthesis to further improve its scalability.

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] 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.

[2] C. Zhang, T. Saluja, R. Meira-Góes, M. Bolton, D. Garlan, and E. Kang, "Robustification of behavioral designs against environmental deviations," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, to appear.

[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] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.

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

[6] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu, *Compositional Reasoning*. Springer International Publishing, 2018, pp. 345–383.

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

[8] 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.

[9] Y. Collette and P. Siarry, *Multiobjective Optimization: Principles and Case Studies*, ser. Decision Engineering. Springer Berlin Heidelberg, 2013.

[10] D. Giannakopoulou, C. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, 2002, pp. 3–12.

[11] 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.

[12] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Springer International Publishing, 2015, pp. 487–495.

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

[14] 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.

[15] 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

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

[17] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, 2000.

[18] D. Sempreboni and L. Viganò, "X-men: A mutation-based approach for the formal analysis of security ceremonies," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 87–104.

[19] 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.

[20] 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.

[21] 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.

[22] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 1–8.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] T. Anevlavis, D. Neider, M. Philippe, and P. Tabuada, "Evrostos: The RLTL verifier," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. Association for Computing Machinery, 2019, p. 218–223.

[31] P. Tabuada and D. Neider, "Robust Linear Temporal Logic," in *25th EACSL Annual Conference on Computer Science Logic (CSL)*, vol. 62, 2016, pp. 10:1–10:21.

[32] 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.

[33] 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.

[34] 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.

[35] 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.