Thesis Proposal

Behavioral Robustness of Software System Designs

Changjian Zhang

Thesis Committee

Dr. Eunsuk Kang (Co-chair)
Software and Societal Systems
Carnegie Mellon University

Dr. David Garlan (Co-chair)
Software and Societal Systems
Carnegie Mellon University

Dr. Jonathan Aldrich
Software and Societal Systems
Carnegie Mellon University

Dr. Sebastian Uchitel
Department of Computing
Imperial College London

A thesis proposal presented for the degree of

Doctor of Philosophy

Software and Societal Systems
Carnegie Mellon University
Pittsburgh, PA

March 18, 2024

Abstract

Software systems are designed and implemented with assumptions about the environment. However, once a system is deployed, the actual environment may *deviate* from its expected behavior, potentially leading to violations of desired properties. Ideally, a system should be *robust* to continue establishing its most critical requirements even in the presence of possible *deviations* in the environment. To enable systematic design of robust systems against environmental deviations, this work proposes a rigorous behavioral notion of robustness for software systems. Then, it presents a technique called *behavioral robustification*, which involves two tactics to systematically and rigorously improve the robustness of a system design against potential deviations.

Specifically, the robustness of a system is defined as the largest set of deviating environmental behavior under which the system is capable of guaranteeing a desired property. Then, we present an approach to compute robustness based on this definition. On the other hand, the system is not robust against an environment when the environment exhibits deviations causing a violation of the desired property. The robustification method finds a re-design that is capable of satisfying the property under such a deviated environment. In particular, two tactics, namely *wrapper* and *specification weakening*, are introduced. We show that how the robustification problem can be formulated as a *multi-objective optimization* problem with the goal of guaranteeing the desired property, while maximizing the amount of existing functionality and minimizing the cost of changes to the original design.

The proposed robustness computation and robustification methods are implemented in a tool, named Fortis. The applicability and efficiency of these approaches are evaluated through experimental results across five case studies, including a radiation therapy machine, an electronic voting machine, network protocols, a transportation fare system, and an infusion pump machine.

Contents

1 Introduction **4**
- 1.1 What is Robustness? . . . 4
- 1.2 Robust-by-Design Software . . . 5
- 1.3 Robustness Assessment and Improvement . . . 6
- 1.4 Thesis Statement . . . 8
- 1.5 Elaboration on Thesis Statement . . . 8
- 1.6 Approach Overview . . . 9
- 1.7 Expected Contributions . . . 10
- 1.8 Problem Scope and Plan . . . 11

2 Related Work **13**
- 2.1 Robustness Definition and Measurement . . . 13
- 2.2 Design Robustification . . . 14

3 Current Work **16**
- 3.1 Background and Motivating Example . . . 16
- 3.2 A Behavioral Notion of Robustness . . . 17
- 3.3 Robustification by Wrapper . . . 21
- 3.4 Evaluation of Current Approach . . . 26

4 Proposed Work **29**
- 4.1 Robustification by Weakening . . . 29
- 4.2 Fortis: A Tool for Robustness Analysis . . . 30
- 4.3 Large-Scale Case Study: Open EMR . . . 30

Bibliography **31**

List of Figures

1.1 Robust-by-design development process. 6

3.1 Labeled transition systems for a radiation therapy system ($M = M_I||M_B||M_S$). . . . 17
3.2 Labeled transition system for the operator task model (E). 18
3.3 Illustration of behavioral relationships between machine M, environment E, and robustness $\Delta(M, E, P)$. 18
3.4 Overview of the robustness computation process. 20
3.5 A deviation model D for the radiation therapy machine. 21
3.6 A deviated environment model E' with *commission errors* for the radiation therapy machine, under which the original design M is not robust. 23
3.7 A redesign of the radiation therapy machine. In particular, we show only the redesigned interface where the operator can fire the beam until the mode switch has completed. 23
3.8 The overall process for solving optimal robustification problems. 25

List of Tables

1.1 Scope and current progress of this work. Green : The task is completed. Red : The task has not started yet. Gray : The task is out of the scope. 12

3.1 Evaluation results of robustness computation. All problems were run on a Linux Machine with a 3.6GHz CPU and 24GB memory under a 30-minute timeout. 27
3.2 Evaluation results of robustification. All problems were run on a Linux Machine with a 3.6GHz CPU and 24GB memory under a 30-minute timeout. 28

Chapter 1

Introduction

1.1 What is Robustness?

A software system is designed and implemented with respect to a specification, which typically, both explicitly and implicitly, makes assumptions about the operating environment. When these assumptions are satisfied, the system is expected to ensure particular functionalities and quality attributes. Nowadays, software systems are applied in diverse domains such as fiance, healthcare, manufacturing, aerospace, autonomous vehicles, and online services. With the increasing capability of software systems, both the systems and their operating environments grow in complexity. It is increasingly common that once a system is deployed, the actual environment may deviate from its expected behavior as described in the specification. For example, an online web application might experience message loss or disruption; a user interacting with a medical device might inadvertently perform actions in the wrong order; or an aircraft might operate in extreme weather conditions, causing sensors to produce inaccurate observations. In such scenarios, the system may not be able to continue providing its assured functionalities or maintaining its specified quality, thereby exposing it to potential risks or failures. From a high-level, *robustness* characterizes the capability of a system to consistently fulfill its commitments even under unexpected circumstances. Therefore, the assurance of software robustness becomes increasingly crucial as system complexity grows, especially for mission-critical or safety-critical systems, such as financial systems, aircrafts, or medical devices [56].

IEEE standards define robustness as *the ability of a software system to continue functioning correctly in the presence of invalid inputs or stressful environment* [1]. Avizienis et al. [5] further characterizes robustness as *a secondary attribute of dependability, i.e., dependability with respect to external faults*. However, these definitions are overly abstract in that they cannot be directly used to help developers analyze the robustness of a system; and the term "robustness" tends to carry different interpretations in various sub-domains of software.

We classify software systems into three categories: *Conventional software systems*, *Machine learning (ML) systems*, and *Cyber-physical systems (CPS)*. Here, conventional systems refer to systems such as operating systems, communication systems, distributed systems, or web services, whose fundamental behavior can be conceptualized through discrete transition systems [37, 25]. ML systems refer to systems with ML components that are statistical models trained against certain datasets. CPS are systems that closely interact with the physical world such as autonomous vehicles. While clear boundaries within this classification do not exist, it is a widely adopted framework in the literature; and robustness definition and evaluation techniques vary significantly across these domains.

Based on this classification, the emphasis of this work is on the robustness of conventional software systems, particularly the robustness with respect to "the correctness of system behavior". This form of software robustness has been widely investigated in the literature, with correct system behavior often characterized as the correctness of the system output or the absence of system failures.

Behavioral Robustness and System-Level Property. Nevertheless, the term "software system behavior" is often used vaguely. It generally refers to how a system reacts and responds to various inputs or events. To precisely delineate the type of robustness studied in this work, it is necessary for us to define the meaning of software behavior.

- *Input-Output Relationships*: In this perspective, a software system often corresponds to a functional procedure, such as a system call in an operating system or an API of a web service. The system behavior is characterized by how it processes inputs and generates corresponding outputs, with the unexpected behavior of the environment characterized by invalid inputs. Specifically, a developer makes assumptions about the input values of a function, e.g., a non-negative integer input for computing a factorial. Then, an *invalid input* is an input value outside the assumed range. Thus, robustness of this kind studies whether the system would produce erroneous outputs given certain inputs that are outside the assumed range.

- *States and Transitions*: In this view of software behavior, a software system is explicitly modeled as a discrete transition system. An execution of the system is defined as a sequence of the system states and their transitions, and the system behavior is the set of all possible executions. Under this definition, two types of properties are often used to specify the correctness of behavior: *safety property* and *liveness property*. A safety property defines the bad states that a system should avoid, while a liveness property defines the desirable states that the system should reach [43]. Robustness of this type assesses the ability of the system to maintain the desired property under unexpected behavior from the environment. Additionally, we specifically focus on *unexpected behavior* as sequences of environmental events (transitions) that occur during an actual operation and are not defined in the assumed environment when designing the software.

We use the term *behavioral robustness* to denote robustness with respect to the behavior of system states and transitions, and use *IO robustness* to indicate the robustness of input-output relationships. In this work, our focus is on the behavioral robustness.

System, Machine, and Environment. Another concern is the term "system". In the context of system behavior as states and transitions, there is often an explicit distinction made between the software and its environment, collectively forming a *closed* system. A closed system is one that does not interact with other elements in the external world. However, in the input-output perspective, a system refers to a procedure that takes certain inputs and produces outputs, with the environment generating the inputs implicitly defined. To avoid this potential confusion, we will use the term *machine* to represent the software and use *system* to denote the composition of a machine and its operating environment.

Therefore, we also make a clear distinction between the properties associated with these two perspectives on behavior. We use *IO property* to indicate a property of the input-output relationships, and use *system-level property* to denote a safety or liveness property at the level of the system (i.e., machine and environment) as a whole.

Finally, we define that this work investigates the behavioral robustness that captures *the ability of a machine to maintain a desired system-level property in the presence of unexpected sequences of events from its environment.*

1.2 Robust-by-Design Software

The objective of this research is to investigate how developers can construct behaviorally robust software, with a specific emphasis on software design. Although in software engineering practices, the significance of system design is often overlooked [56] with engineers placing greater emphasis on implementation and testing phases, it is also widely acknowledged by industrial practitioners and academics that *the longer an issue lingers in the system, the more effort it requires to resolve* [13]. Moreover, for safety-critical or mission-critical systems, establishing a "correct" design that ensures

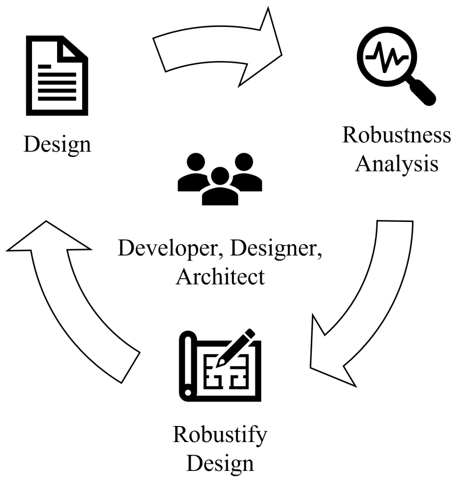

Figure 1.1: Robust-by-design development process.

robustness is even more crucial, given that a failure may be deemed unacceptable, and fixing defects in later phases can become exceedingly costly or, in some cases, unfeasible. Therefore, this work specifically focuses on the behavioral robustness of software designs.

We adopt a *robust-by-design* development process, as illustrated in Figure 1.1. In this process, developers start with an initial machine design that operates effectively under normal environmental conditions. Subsequently, they conduct an analysis to evaluate the robustness of the machine. This analysis may not only reveal the invalid inputs or faults that lead to robustness violations but may also quantitatively or qualitatively quantify the degree of robustness. Consequently, with these analysis results, developers then modify or redesign the machine to enhance its robustness, especially if the initial design does not meet the specified requirements. Additionally, developers can measure the robustness of the new design and compare it with the old design, uncovering potential design trade-offs, such as additional costs or the sacrifice of certain functionalities for ensuring robustness. This iterative process may be repeated multiple times until a satisfactory level of design robustness is achieved.

To support this development process, we argue that a methodology with the following capabilities is crucial. Specifically, it should be able to:

- *systematically and rigorously quantify the degree of robustness of a machine, compare the robustness of two machines, and identify robustness vulnerabilities if they exist*,
- *systematically and rigorously improve machine robustness in response to the identified robustness vulnerabilities.*

While such design and development practices are prevalent in other well-established engineering disciplines such as aerospace, civil, and manufacturing [58], existing techniques in software robustness lack sufficient support for this process.

1.3 Robustness Assessment and Improvement

As outlined in the robust-by-design process, it comprises two crucial activities: robustness assessment and robustness improvement. According to the surveys by Shahrokni et al. [63] and Laranjeiro et al. [46], robustness assessment techniques for conventional software are predominantly experimental, with a primary focus on robustness testing against a software implementation rather than a software design. Furthermore, most of these techniques concentrate on the IO robustness of a machine, which contrasts with the behavioral robustness that we investigate in this study.

For improving software robustness, the prevalent approach involves using wrappers over existing functions and components to mask and prevent the propagation of errors. However, these methods also predominantly center around software implementation and IO robustness. On the other hand, the self-adaptive system community [48] and the run-time assurance community [31] explore how to ensure the run-time correctness of a system, which aligns more closely with our notion of behavioral robustness.

Robustness Assessment Techniques. According to the surveys [63, 46], existing testing-based robustness assessment methods typically answer the following questions:

Is the machine (implementation) robust against certain type of invalid inputs or faults?
Will the machine crash or generate erroneous outputs?

To address these questions, researchers commonly adopt a testing pattern that involves: (1) generating and providing invalid inputs or faults to the machine under test; and (2) assessing whether the machine produces erroneous outputs or exhibits unexpected behavior, such as crashes. The most prevalent evaluation methods include fault injection [7, 54], where developers intentionally introduce specific types of faults to the machine or its environment, and model-based testing [68], where a formal model is employed to generate test cases executed on the concrete machine implementation. Additional methods include fuzzing [14, 51], which automatically and randomly generates extensive inputs, model-based analysis, which formally models and verifies the robustness of a machine [4], and mutation testing, which aims to enhance the quality of test cases for detecting more faults [40, 61, 28].

However, robustness testing methods such as these fail to address our objective, i.e., the assessment method should be able to *systematically and rigorously quantify the degree of robustness of a machine*. In other words, they fail to answer the following question:

What is the set of all invalid inputs or faults that the machine is robust against?

While testing-based methods excel at identifying robustness violations, they often cannot offer robustness guarantees when no violations are detected. In such instances, it remains unclear whether we should allocate additional resources to conduct the robustness tests, or if the machine can be confidently deemed robust. Moreover, determining which machine is more robust than another based solely on testing results is challenging. Additionally, a large portion of these methods focus on IO robustness, i.e., detecting robustness violations caused by invalid inputs, while behavioral robustness remains relatively under-explored [63].

Robustness Improvement Techniques. In their survey, Shahrokni et al. [63] conclude that existing works mostly focus on the use of wrappers and encapsulation of existing software components to improve robustness. These studies often delve into the source code of the software, examining how error detection code and explicit exception handlers can filter out invalid inputs and erroneous outputs and prevent the propagation of errors. Similar methods are also advocated in programming guidelines such as defensive/robust programming [9] and practitioner-oriented books like [52, 10].

However, these methods also take the view of input-output relationships. While improving the IO robustness of individual components can eventually impact the robustness of the machine as a whole, they fail to provide *a rigorous and systematic way of improving the behavioral robustness of the machine.*

On the other hand, contributions from the self-adaptive system community [48] and the run-time assurance community [31] present methods on how to ensure a machine to maintain its functionalities or quality attributes at run time against uncertainties. Some of these approaches can be seen as ways of improving behavioral robustness of a machine, even though the term "robustness" may not be explicitly mentioned, or they may consider "robustness" with respect to a broader set of properties other than pure safety and liveness, such as performance or availability [48, 22]. While these methods can be utilized to improve the behavioral robustness of a machine and indeed inspire our approach, one significant difference is that we focus on design-time robustness improvement whereas they assume run-time adaption or assurance.

In light of the current state of software robustness research, this work explores a method designed to systematically and rigorously analyze, measure, and improve the behavioral robustness of a machine design. It addresses a gap in the current research, manifested as a lack of systematic reasoning and enhancing methods for software design robustness. Furthermore, this method can facilitate a robust-by-design process for software.

1.4 Thesis Statement

Given a machine and its environment that can be formally modeled as a state-transition system, we can systematically measure and improve the behavioral robustness of the machine with respect to the environment and a system-level safety property.

1.5 Elaboration on Thesis Statement

Behavioral Robustness of Software Design. As detailed in the preceding sections, this work focuses on the behavioral robustness of software designs. Specifically, a *system* is closed and consists of a *machine* (i.e., the software) and its *environment*. The system is considered as a state-transition system, and its *behavior* is characterized by a set of execution traces, each being a sequence of states and transitions. A *system-level* property, i.e., a safety property or a liveness property, specifies the intended behavior of a system. The *unexpected behavior* from the environment is characterized by sequences of environmental events that occur at run time and are not defined in the assumed environment. Therefore, this work investigates the *behavioral robustness* of a machine design that captures the ability of it to continue satisfying a desired system-level property in the presence of unexpected sequences of events from its actual operating environment.

Formal Reasoning. Our approach leverages formal methods to achieve our objectives of systematically and rigorously quantifying and improving robustness. Specifically, a formal state-transition system serves as a natural representation for the behavior of a software system, and each trace in such a formal model indicates a particular execution scenario. For instance, in a network protocol, a sequence of events $\langle send, receive, acknowledge,\ get\_acknowledge\rangle$ depicts a normative execution where the client sends a request and receives the acknowledgement from the server. In contrast, an event sequence $\langle send, message\_lost\rangle$ illustrates a faulty scenario where the client's request is lost during transmission. With such a formal model, we can mathematically analyze the behavioral robustness of a machine as a set of traces. In addition, it also enables systematic and automated enhancement of robustness by generating a more robust machine model through modification or synthesis.

Safety vs. Liveness. This work primarily focuses on behavioral robustness with respect to a safety property. A safety property specifies the bad states that a system should avoid. We prioritize safety for several reasons: (1) it is widely studied in the literature and aligns intuitively with developers' objective of preventing bad behavior in a system, (2) ensuring safety is particularly crucial for safety-critical and mission-critical systems, and (3) robustness challenges differ substantially between safety and liveness properties, leading us to prioritize one over the other given the limited time available for this research. Nevertheless, we do not completely disregard liveness in this work, especially concerning robustness improvement. Given that a system simply losing all its functionalities and entering a termination state is deemed safe but practically useless, we recognize the importance of incorporating liveness as a crucial dimension when enhancing robustness.

1.6 Approach Overview

The expected contributions of this work are: (1) a behavioral notion of robustness for software designs and its computation method based on labeled transition systems, namely *robustness analysis*, and (2) an approach for improving the behavioral robustness of a system design, namely *design robustification*.

We consider a *system* S to consist of a *machine* M and an *environment* E, the composition of the machine and the environment forms a *closed* system. Then, a typical software design task involves answering the following question: Given machine M and environment E, does the composed system satisfy a desired property P given the machine operating under the environment, i.e., $M||E \models P$?

Then, in terms of robustness, we say the machine M should continue to satisfy the desired property even under an environment E' that *deviates* from the expected one specified by E, i.e., $M||E' \models P$. Specifically, E' and E should contain the same set of events but differ in the set of event traces they prescribe. The distinctions in the sets of traces are denoted as *deviations*, represented by δ. These deviations may due to errors or natural changes in the actual environment. Finally, the robustness is measured as the maximum possible set of deviations, denoted by Δ, such that the machine continues to satisfy the desired property under these deviations. Conceptually, Δ represents the safe operating envelope of a machine, i.e., as long as the environmental deviations remain within this envelope, the machine can guarantee property P.

Formalism. We utilize labeled transition systems (LTS) to model the discrete behavior of a machine and its environment. In an LTS, we explicitly model the sequences of events that occur in a system, with the system states being implicitly defined. A trace in an LTS is a sequence of events, and the behavior is the set of all traces. Furthermore, we consider safety properties in terms of LTS in this work. A safety property is also an LTS that describes a set of traces, and we say a model T satisfies a safety property P when the set of traces of T is a subset of those defined by P.

Robustness Analysis. We propose a formal behavioral notion of software robustness based on LTS. Given a machine M, an environment E, and a safety property P, all modeled in an LTS, a deviation of the environment is represented as a sequence of events (a trace). We measure robustness Δ as a set of traces that are not defined in E under which M continues to satisfy P. For example, in a network protocol, the trace $\langle send, receive, acknowledge, get\_acknowledge \rangle$ is a normative scenario defined in E, assuming a perfect communication channel, whereas the trace $\langle send, message\_lost \rangle$ is a deviation that can occur with an imperfect channel. Then, if M still satisfies property P under this deviation trace, it belongs to robustness Δ.

We present an approach to compute robustness Δ that contains all deviation traces under which the machine M continues to satisfy property P. The computation process also facilitates robustness comparisons. Additionally, in general, robustness Δ may contain an infinite number of traces, which is not easily comprehensible by the developers. Thus, we present an approach to partition Δ into a finite set of *equivalence classes* and sample *representative traces* from them, each trace represents a group of traces that describe the same type of deviations. Finally, we use a *deviation model* to generate explanations for those representative traces. An explanation describes what environmental faults cause the environment to deviate from its expected behavior. The final output from this analysis is a set of pairs of a representative trace and its corresponding explanation. Section 3.2 describes our approach in more detail.

Design Robustification. We propose an approach to synthesize new machine designs based on an existing design, a safety property, and a deviated environment. Compared to the assumed environment, a *deviated environment* is one that shares the same set of events but contains additional deviation traces where the old design fails to satisfy the property. The goal of robustification is to find a new design that is robust against the deviations in the deviated environment. Specifically, our method considers four dimensions of robustification: safety, liveness, observability and controllability, and cost.

As defined in the thesis statement, our primary focus is on robustness with respect to safety. However, ensuring safety may compromise liveness in the sense of losing certain desired functionalities. In an extreme case, safety can be trivially achieved by letting a machine enter a termination state and do nothing. Therefore, while full support for robustness regarding liveness is beyond the scope of this work, we aim for the robustification process to guarantee as much liveness as possible. However, when certain safety and liveness properties cannot be satisfied simultaneously, one issue may be the lack of observability and controllability of the machine. Intuitively, a machine that can observe and control more events in the system can achieve more fine-grained control to better prevent the propagation of errors and recover from the faults. Nevertheless, enhanced observability and controllability often come at a higher cost. Therefore, our method needs to consider these four dimensions.

Our proposed method consists of two components. The first component employs a *wrapper* tactic, utilizing supervisory control theory [20] to generate new designs. It improves robustness by monitoring events from the machine and the environment, disabling specific events to ensure the safety property. To prevent excessive restrictions on liveness, we allow the extension of observability and controllability of the new machine, however, at an additional cost. Therefore, the robustification process is framed as a multi-objective optimization problem with two quality goals: (1) preserving behavior from the original design and (2) minimizing cost of changes, measured by the extended observing and control abilities. We introduce a novel algorithm for searching optimal redesigns, and Section 3.3 provides a detailed description of this technique.

The second robustification component is the ongoing work of this thesis. The wrapper technique may potentially disable certain critical machine functionalities even with extended observability and controllability, especially when the desired safety property is exceptionally stringent. Therefore, the second robustification process explores *specification weakening* as an additional tactic alongside the wrapper tactic. Intuitively, by employing a weaker safety property, the system should be capable of tolerating more deviations, as certain deviations would no longer lead to a safety violation. Consequently, the new tactic involves weakening a potentially too-strong safety property, allowing for improved robustness through the application of the wrapper tactic without disabling critical system functionalities. Section 4.1 provides a detailed description of this technique.

Implementation and Evaluation. We implement all our proposed approaches in a tool named *Fortis*. The tool includes a simple Graphical User Interface (GUI) for users to specify system models and properties and run our methods to compute robustness and robustify a machine. We evaluate the applicability and efficiency of our approaches through five case studies, which include a radiation therapy machine, an electronic voting machine, network protocols, a medical infusion pump machine, and a public transportation fare system.

The evaluation demonstrates the applicability of our robustness computation method across diverse case studies originating from various software application domains. The computed results, specifically deviations, correspond to real-world erroneous scenarios that have been previously investigated in other domains. The evaluation also demonstrates that our robustification process can successfully find optimal new designs robust against a deviated environment, across software applications from different domains. Lastly, the efficiency of our robustness computation and robustification methods is demonstrated through benchmarking against a set of problems derived from our five case studies.

1.7 Expected Contributions

The expected contributions of this thesis include:

- **Behavioral Notion of Robustness**: We propose a behavioral notion of robustness for software systems based on labeled transition systems, defining robustness as a set of event traces not specified in the assumed environment, under which the machine continues to satisfy a desired safety property.

- **Robustness Computation Approach**: We provide an approach to compute and represent the behavioral robustness of a system. Additionally, the computation process facilitates robustness comparisons.
- **Robustification Approach**: We introduce an approach to robustify a machine with respect to a deviated environment and a safety property, employing two tactics: wrapper and specification weakening.
- **Implementation and Evaluation**: We implement the proposed robustness computation and robustification approaches in a tool, evaluating their applicability and efficiency through five case studies.

1.8 Problem Scope and Plan

ML and CPS robustness. The initial focus of this work revolves around robustness for conventional software systems and its robustification, making LTS a suitable formalism. However, this formalism cannot be directly applied to ML systems or CPS. ML systems typically involve statistical models trained and optimized against certain datasets, and their concrete behavior may be challenging to discern, especially for complex structures like Deep Neural Networks (DNN) [38]. On the other hand, CPS interacts closely with the physical world, such as aircraft or autonomous vehicles, often involving modeling, control, and manipulation of physical processes characterized by differential equations or hybrid logic [47].

While our research interest extends to the robustness of ML systems and CPS, our preliminary investigation [72] indicates that these domains may require significantly different methodologies. The notion of robustness and the robustification techniques presented in this work would need non-trivial theoretical extensions, such as the incorporation of probabilistic modeling and properties [6] or leveraging statistical model checking [2]. Therefore, considering the time constraints of this thesis, the extension of our proposed robustness methods to ML systems and CPS is beyond the scope of this work.

Type of Uncertainties. This work investigates a specific type of robustness, termed behavioral robustness. In this framework, uncertainties (deviations) are defined as sequences of environmental events that are not specified by the assumed environment. Specifically, these sequences should only differ in the order of events, while the set of events should remain unchanged. Hence, dealing with other types of uncertainties, such as unknown events from the environment, is out of the scope of this research. In addition, we also do not consider the probability associated with the occurrence of these deviations.

Liveness Properties. The comprehensive support of liveness properties is beyond the scope of this work. Introducing support for robustness computation and robustification concerning liveness properties would demand significant theoretical extensions to our existing approach. Moreover, our intuition suggests that robustness with respect to liveness properties poses a more intricate challenge. One of the challenges is that our current robustification approach primarily employs the idea of disabling actions to enhance robustness. However, addressing liveness properties may require the addition of new behavior to the machine, such as incorporating retries in a network protocol. This could potentially lead to a much larger solution space, making it a much more challenging task. Therefore, we decided to concentrate on safety properties in this work.

Large-Scale Evaluation. While we implement our approaches and evaluate them on five case studies that span diverse domains with real-world counterparts, it is important to acknowledge that these case studies may not fully encapsulate the complexities inherent in real-world software systems. Evaluating our approaches on large-scale real-world systems would require modeling such systems with appropriate abstractions and potentially validating our analysis results against concrete

	Description	Est. Time Remains
Must-Haves	A behavioral notion of robustness for conventional software systems in LTS.	Completed
	The computation of robustness w.r.t. safety properties.	Completed
	Robustification w.r.t. safety properties by wrapper.	Completed
	Robustification w.r.t. safety properties by specification weakening.	3 months
May-Haves	Evaluation over a large-scale, real-world system.	3 months
	Improving and evaluating the quality of our robustness analysis tool, e.g., usability.	2 months
Won't-Haves	Robustness computation and robustification w.r.t. liveness properties.	N/A
	Extensions of our approaches to ML systems or CPS.	N/A

Table 1.1: Scope and current progress of this work. Green : The task is completed. Red : The task has not started yet. Gray : The task is out of the scope.

implementations. Given the substantial engineering and research effort involved, it is considered a stretch goal of this work.

Evaluation of the Tool. Lastly, our implementation, encapsulated in a tool, serves as a demonstration of our proposed approaches. Substantial engineering effort will be invested in integrating the robustness computation and robustification techniques into a unified framework, with a focus on improving computation efficiency. A simple GUI is implemented to provide essential interaction capabilities for users. However, enhancing other quality attributes of the tool, such as usability, demands non-trivial engineering effort. Therefore, evaluating the overall quality of the tool is also considered a stretch goal beyond the primary objectives of this work.

Given the problem scope, Table 1.1 summarizes the research plan, where *Must-Haves* are the essential components that must be delivered of this work, *May-Haves* are the components that we would potentially deliver, and *Won't-Haves* are the out-of-the-scope components.

Chapter 2

Related Work

2.1 Robustness Definition and Measurement

According to the survey by Shahrokni et al. [63] and Laranjeiro et al. [46], most of the prior works on robustness for conventional software systems focus on testing. Popular methods such as fault injection [7, 54], model-based testing [68], and fuzzing [14, 51] are designed to evaluate the robustness of a system by identifying invalid inputs or environmental faults that cause undesirable system behavior, often measured by crashes or failures. A exemplar work is by Koopman et al. [42]. The approach generates invalid inputs for a set of identified system calls in an operating system and uses a five-scale categorization, named CRASH, to categorize the severity of failures. The survey conducted by Laranjeiro et al. in 2021 [46] provides a more comprehensive and up-to-date review of the current state of research in software robustness assessment.

In contrast, we compute robustness as an intrinsic characteristic of the software, i.e., we systematically compute all the deviations that a machine can tolerate. In addition, we believe our robustness metric (i.e., a set of deviation traces) can potentially be used to complement existing robustness testing approaches. For instance, we could generate test scenarios based on traces in robustness Δ to verify that the implementation of the machine is robust against certain types of environmental deviations.

Our formal robustness definition assumes discrete transition systems. Various formal definitions of robustness for discrete systems have been investigated [64, 35, 36, 12]. One common characteristics of these prior definitions is that they are all *quantitative* in nature, in that they all define certain kind of function to measure the *distance* between traces and system behavior. For example, Bloem et al. [12] propose a notion of robustness that defines, for a robust system, the ratio of the degree of incorrect machine outputs over the degree of incorrect inputs should be small, where the degree is measured by a function that maps every possible trace to a value indicating how "close" the behavior is to a correct behavior. Similarly. Tabuada et al. [64] propose a notion that assigns costs to certain input and output traces (e.g., a trace that deviates significantly from the expected behavior should have a high cost) and stipulates that an input trace with a small cost should only result in an output trace with a proportionally small cost. Henzinger et al. [35, 36] adopt the notion of *Lipshitz continuity* from control theory to define robustness, where a system is K-(Lipschitz) robust if the deviation (measured by a *similarity function*) in its output is at most K times the deviation in its input.

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

Tabuada and Neider propose an extension of linear temporal logic called *robust linear temporal logic* (rLTL) [65]; similarly, Nayak et al. propose *robust computation tree logic* (rCTL) [55]. Both of them use a multi-valued semantics to capture the different levels of satisfaction of a property;

e.g., given an expected property $\mathbf{G}\phi$, i.e., ϕ should always be true, then $\mathbf{FG}\phi$ is considered a weaker version of it, i.e., eventually ϕ should always be true. Therefore, robustness can be measured as: a "small" violation of the environment assumption must cause only a "small" violation of the property satisfaction degree. In our work, we say a machine is robust against a deviation when the desired property continues to be satisfied, following a binary criterion. Thus, our notion of robustness could potentially be extended with rLTL or rCTL to compute robustness as an ordered set of deviations.

In safety engineering and risk management, *operating envelope* or *safety envelope* has been used to represent the boundary of environmental conditions under which the system is capable of maintaining safety [59]. This concept has been adopted in a number of engineering domains such as aviation, robotics, and manufacturing, but as far as we know, has not been rigorously defined in the context of software engineering. Therefore, our notion of robustness can be considered as one possible definition of the safety envelope for software systems.

There exist alternative notions of software robustness that significantly differ from both IO robustness and behavioral robustness. Schulte et al. [61] propose *software mutational robustness* where robustness is measured by the fraction of random mutations to program code that leave a program's behavior unchanged. They focus on deviations (mutations) that occur in the code whereas we focus on deviations as sequences of events from the environment. Petke et al. [57] argue that a robust program should be able to stop the propagation of failures; and according to information theory, they prospect that robustness might be captured as entropy loss in the code region succeeding the code region where the faults occur. The higher is the entropy loss, the higher the likelihood the propagation of the failure could be prevented.

2.2 Design Robustification

Research on robustness for discrete systems in control theory has not only provided formal definitions of robustness but has also introduced methods for synthesizing a robust controller [12, 64, 35, 11, 41]. These works rely on a *quantitative* notion of robustness, involving numerical measures of deviations. In contrast, our approach relies on a *qualitative* definition of robustness that centers around deviations as discrete events, aligning more closely with the nature of software systems. However, the foundational technique for our wrapper robustification method, namely supervisory control synthesis [20], also originates from a sub-field of control theory dedicated to discrete event systems.

Similar concepts of control have also been applied in the context of self-adaptive systems [48] and run-time assurance [29, 31, 30] to dynamically enforce system requirements. For instance, in the self-adaptive systems community, the MAPE-K adaption framework [48], including tools like Rainbow [32], also employs a monitoring and actuation control loop to ensure a system to continue satisfying certain properties against environmental uncertainties at run time. However, these run-time approaches typically assume fixed sensing and actuating capabilities. By comparison, our work focuses on robustifying a machine at *design time*, providing developers with the flexibility to extend the sensing and actuating abilities by introducing additional observable and controllable events.

Robustification by wrapper. Our robustification by wrapper approach solves the problem of synthesizing a new machine model that satisfies a desired property. Thus, it shares similarities with model repair. *Model repair* addresses the problem that: Given system M and property P where $S \not\models P$, generates a new system S' such that $S' \models P$. Buccafurri et al. [18] propose a formal definition of model repair for Computation Tree Logic (CTL) and present an approach to find repairs using abductive reasoning. Similarly, Menezes et al. [53], Chatzieleftheriou et al. [21], and Ding et al. [26] present repair approaches for CTL, α-CTL (which considers actions behind transitions), Kripke Modal Structure (which contains *must-transitions* and *may-transitions*), and Linear Temporal Logic (LTL), respectively. Our wrapper robustification approach can be considered as a kind of model repair. However, it addresses how to enhance the machine M to tolerate deviations in the environment E, whereas prior model repair works do not make distinction between the machine and the environment of a system. Moreover, the existing works do not consider the cost of a repair or consider the cost based on only the syntactic changes of the model, e.g., adding or removing states or transitions. However, our approach considers multiple semantic-based quality metrics of repairs, i.e., the value of preserved behavior and the cost of events for observing or controlling the machine and the environment.

We leverage supervisory control synthesis to generate new machine designs, where the closet work is presented by Tun et al. [67] which also uses controller synthesis to generate new designs that satisfy a desired property. Although their work does not explicitly aiming at improving the robustness of a software system, they have the similar goal to revise a machine M to fulfill a security requirement P in an environment E where the users might deviate from the expected behavior causing security violations. Our wrapper approach and OASIS differ in the way they explore and generate new designs: OASIS uses an *abstraction-based* technique to allow changing the sequence of events in an machine to generate new designs that satisfy certain property, while our approach allows adding events to be observed or controlled by the new designs. OASIS's approach could potentially be a complementary exploring method for us to search optimal robustification solutions. In addition, OASIS does not consider optimizing designs for the two quality goals., i.e., minimizing the cost of changes and preserving behavior from the old design.

Robustification by weakening The idea of weakening in the context of requirements engineering [44, 3] is rooted in the recognition that environmental conditions may change over time and space. As a result, original requirements might become inadequate or inconsistent with the new environment, necessitating adaptation or weakening. This concept has been further explored in self-adaptive systems [69, 19, 23].

In the work by Alrajeh et al. [3], the focus is on adapting *system requirements* to address environmental deviations. The authors propose an approach that utilizes a learning technique to automatically adapt system requirements specified in a goal model [45] with Metric Temporal Logic (MTL) to changes in the environment. Similarly, Chu et al. [23] explore similar ideas in the context of CPS with Signal Temporal Logic (STL). They introduce an extension to STL called *weakened STL*, where the time bounds of temporal operators can be weakened to make a formula easier to be satisfied. At run time, when a CPS is about to violate its system specification, they synthesize a new control action that satisfies a minimally weakened formula. Additionally, Buckworth et al. [19] present a run-time adaptation technique involving the weakening of LTL. In their work, when a self-adaptive system violates its specification at run time, they learn a weakened specification that aligns with the current environmental conditions and synthesize a new controller based on this adjusted specification.

D'Ippolito et. al [27] also adopts the weakening concept and proposes a *multi-tier control* approach for self-adaptation, where the developer provides a hierarchy of environment models $(E', E'', \ldots)$ that embody different levels of uncertainty, and synthesizes different machines $(M', M''$, $\ldots)$ to satisfy gradually weakened system goals $(P', P'', \ldots)$. Then, at run time, the system dynamically switches between different controllers that best correspond to the current environment.

The adoption of the weakening and its extension to robustness in our work aligns with the intuition that a weaker property can improve a system's ability to tolerate more environmental deviations, thereby improving its robustness. By combining this concept with the wrapper tactic, we are able to identify additional feasible new designs that are robust against a given deviated environment. Through this approach, the new designs may preserve more behavior from the original design or incur lower costs.

Chapter 3

Current Work

This chapter describes our current progress on behavioral robustness of software systems. Specifically, Section 3.1 introduces the necessary background on labeled transition systems and an example to illustrate our approach. Section 3.2 describes our robustness definition and its computation. Section 3.3 describes our approach on robustification by wrapper. Finally, Section 3.4 describes the evaluation of our current approach.

3.1 Background and Motivating Example

We first introduce the necessary background on labeled transition systems. Then, we introduce a radiation therapy machine example to illustrate our approach through out this chapter.

3.1.1 Labeled Transition Systems

A *labeled 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 $beh(T)$, which can also be referred as the language of T.

Operators. For LTS $T = \langle S, \alpha T, R, s_0 \rangle$, the *projection* operator $\upharpoonright$ exposes a subset of the alphabet of T. Given $T \upharpoonright A = \langle S, \alpha T \cap A, R', s_0 \rangle$, for any $(s, a, s') \in R$, if $a \notin A$, then $(s, \tau, s') \in R'$; otherwise, $(s, a, s') \in R'$. The $\upharpoonright$ operator also applies to traces; $\sigma \upharpoonright A$ 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 [8]. Given $T_1 = \langle S^1, \alpha T^1, R^1, s_0^1 \rangle$ and $T_2 = \langle S^2, \alpha T^2, R^2, s_0^2 \rangle$, $T_1 || T_2$ is LTS $T = \langle S, \alpha T, R, s_0 \rangle$ where $S = S^1 \times S^2$, $\alpha T = \alpha T^1 \cup \alpha T^2$, $s_0 = (s_0^1, s_0^2)$, and R is defined as: For any $(s^1, a, s^{1'}) \in R^1$ and $a \notin \alpha T^2$, we have $((s^1, s^2), a, (s^{1'}, s^2)) \in R$; for any $(s^2, a, s^{2'}) \in R^2$ and $a \notin \alpha T^1$, we have $((s^1, s^2), a, (s^1, s^{2'})) \in R$; and for $(s^1, a, s^{1'}) \in R^1$ and $(s^2, a, s^{2'}) \in R^2$, we have $((s^1, s^2), a, (s^{1'}, s^{2'})) \in R$.

Properties. In this work, we consider a class of properties called *safety properties* [43], 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 $beh(T \upharpoonright \alpha P) \subseteq beh(P)$.

We check whether an LTS T satisfies a safety property $P = \langle S, \alpha P, R, s_0 \rangle$ by automatically deriving an *error* LTS $P_{err} = \langle S \cup \{\pi\}, \alpha P, R_{err}, s_0 \rangle$ where π denotes the error state, and $R_{err} =$

$R \cup \{(s, a, \pi) | a \in \alpha P \wedge \nexists s' \in S : (s, a, s') \in R\}$. With this P_{err} LTS, we test whether the error state π is reachable in $T||P_{err}$. If π is not reachable, then we can conclude that $T \models P$.

3.1.2 Example: Radiation Therapy Machine

We use a radiation therapy machine similar to the well-known Therac-25 machine [49] to illustrate our approach on robustness computation and robustification. Figure 3.1 shows the labeled transition systems of the main components of the machine, including (a) *Treatment Interface* (M_I), which allows an operator to choose the radiation mode (Electron or X-ray) and fire the beam, (b) *Beam Setter* (M_B), which switches the physical component for the two radiation modes, 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 (as X-ray delivers roughly 100 times higher level of current than the Electron beam). The overall behavior of the machine is the composition of the three components, i.e., $M = M_I||M_B||M_S$.

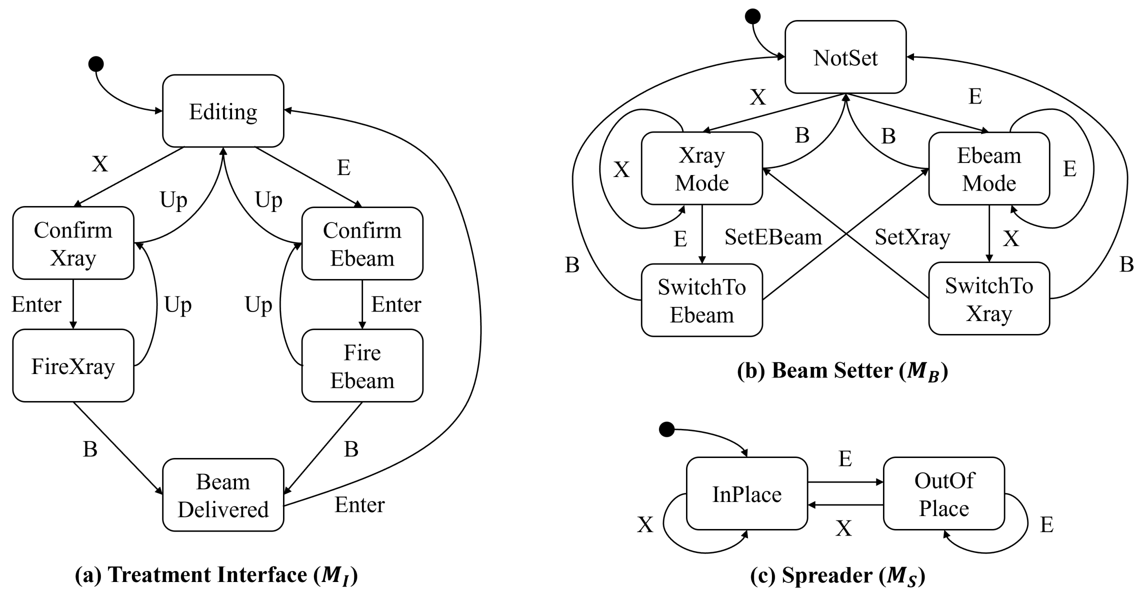

Figure 3.1: Labeled transition systems for a radiation therapy system ($M = M_I||M_B||M_S$).

We consider an important safety requirement of the system that is the spreader must be in place when the beam is delivered in X-ray mode. This requirement can be formally defined as a LTL formula P[1]:

$$\mathbf{G}(BeamDelivered \wedge XrayMode \Rightarrow InPlace)$$

Moreover, the task to be carried by an operator is specified as an environment model (E) in Figure 3.2. In particular, it describes: 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. By using a model checker [24], we can tell that the machine satisfies the property under the normative task model, i.e., $M||E \models P$.

3.2 A Behavioral Notion of Robustness

This section introduces our notion of behavioral robustness. Section 3.2.1 presents our definition of robustness. Then, Section 3.2.2 presents the design questions that our analysis can answer. Finally, Section 3.2.3 outlines the approach we have taken to compute and represent robustness.

[1]This property is later translated into a deterministic LTS for safety check.

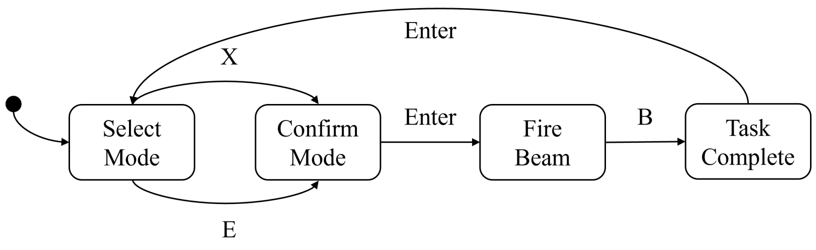

Figure 3.2: Labeled transition system for the operator task model (E).

3.2.1 Robustness Definition

In our work [71], we propose a new definition of robustness based on labeled transition systems. It captures robustness as a set of environmental behavior under which the system can continue to guarantee certain safety property.

Specifically, let M be a machine, E be its operating environment, both modeled in LTS, and P be a desired safety property, where M satisfies the property under E, (i.e., $M||E \models P$). Machine M is said to be *robust* against a set of environmental traces δ if and only if the machine continues satisfying the desired property P under a new environment E' that is capable of additional behaviors in δ compared to the original environment E. Formally, we have:

Definition 3.2.1 (Robust System) *Machine M is robust against a set of traces $\delta \subseteq \alpha E^*$ with respect to environment E and property P if and only if $M||E \models P$, $\delta \cap beh(E) = \emptyset$, and for E' such that $beh(E') = beh(E) \cup \delta$, $M||E' \models P$.*

The traces in δ are also called *deviations* with respect to the original environment E. For example, in the radiation therapy machine, trace $\langle X, B\rangle$ is a deviation to the normative environment, and it is easy to see that when applying this trace to the machine, the machine still satisfies the safety property (as action B is disabled in state ConfirmXray). Therefore, we say the machine is robust against the deviation $\langle X, B\rangle$.

Then, the *robustness* of machine M is defined as the largest set of deviations under which the machine continues to satisfy property P. Formally, we have:

Definition 3.2.2 (Robustness) *The robustness of machine M with respect to environment E and property P, denoted by $\Delta(M, E, P)$, is the set of traces δ such that M is robust against δ with respect to E and P, and there exists no δ' such that $\delta \subset \delta'$ and M is also robust against δ'.*

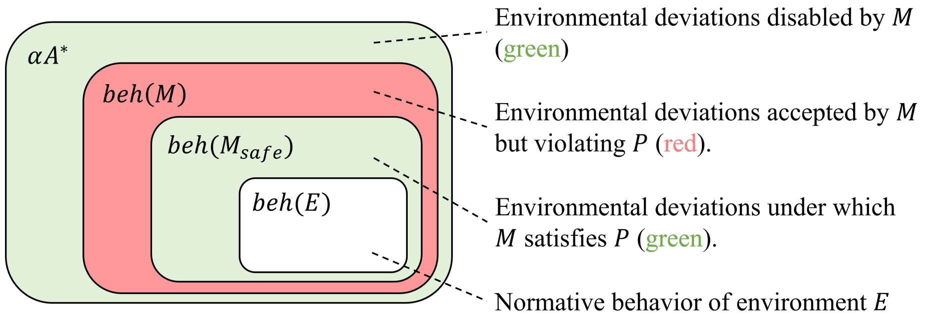
αA^*
$beh(M)$
$beh(M_{safe})$
$beh(E)$
Environmental deviations disabled by M (green)
Environmental deviations accepted by M but violating P (red).
Environmental deviations under which M satisfies P (green).
Normative behavior of environment E

Figure 3.3: Illustration of behavioral relationships between machine M, environment E, and robustness $\Delta(M, E, P)$.

Figure 3.3 illustrates the relationships between the behaviors of the machine, the environment, and the robustness. For illustration purpose, we assume that they all have the same alphabet αA.

The outermost box represents all the possible finite traces given αA. Then, $beh(M)$ represents all the possible behaviors permitted (accepted) by the machine; and $beh(E)$ represents all the behaviors of the normative environment.

Given safety property P, the behaviors of M can be classified into two sets: $beh(M_{safe})$, the set of *all* behaviors that are permitted by the machine M and under which the machine satisfies P, and the rest unsafe behavior (i.e., $beh(M) \setminus beh(M_{safe})$) that lead to a violation (red area). Therefore, for $\delta_1 = beh(M_{safe}) \setminus beh(E)$, it represents the set of all deviations that the machine accepts and is robust against. Meanwhile, $\delta_2 = \alpha A^* \setminus beh(M)$ is the set of all deviations that the machine does not accept, or in other words, disables.[2] Thus, given Definition 3.2.1, we say machine M is also robust against δ_2. Hence, the robustness of the machine should consist of both δ_1 and δ_2, i.e., $\Delta(M, E, P) = \delta_1 \cup \delta_2$, the union of the two green areas.

For example, trace $\langle X, Enter, Up, Enter, B\rangle$ is a deviation that accepts by the machine and under which the machine satisfies the property, which belongs to the deviation set δ_1. On the other hand, $\langle X, B\rangle$ is a trace disabled by the machine and thus under which the machine also satisfies the property, which belongs to the deviation set δ_2. Both of these deviations should be included in the robustness set $\Delta(M, E, P)$.

3.2.2 Design Questions

Given our robustness definition, we can answer the following design questions with respect to robustness.

First of all, we can compute the robustness as defined in 3.2.2, i.e.,

Problem 3.2.1 (Robustness Computation) *Given a machine M, an environment E, and a safety property P, compute the robustness $\Delta(M, E, P)$.*

Moreover, we can answer robustness comparison questions as follows.

Problem 3.2.2 (Design Comparison) *Given machines M_1 and M_2, an environment E, and a safety property P such that $\alpha M_1 = \alpha M_2$, compute set $\mathcal{X} = \Delta(M_2, E, P) - \Delta(M_1, E, P)$.*

This analysis allows us to compare a pair of designs on their robustness given the same environment and property. M_2, for example, may be an evolution of M_1; and thus the result of this analysis indicates precisely how M_2 is robust against some deviations that M_1 is not. On the other hand, $\Delta(M_2, E, P)$ may not necessarily subsume $\Delta(M_1, E, P)$ indicating the design trade-offs that being robust against certain deviations may lead to violations under other deviations.

Another similar type of analysis is to compare the robustness of a single machine under different properties:

Problem 3.2.3 (Property Comparison) *Give a machine M, an environment E, and safety properties P_1 and P_2, compute set $\mathcal{X} = \Delta(M, E, P_2) - \Delta(M, E, P_1)$.*

A use case of this analysis is that given a stronger safety property P_1 and a weaker property P_2 where P_2 might be easier to satisfy, the result of this analysis can tell us exactly which deviations the system becomes robust against under the weaker safety property. It indicates the design trade-offs between safety and fault tolerance and may be useful in the context of requirements weakening [23]. In other words, since in general improving robustness might introduce additional complexity to a system, it may be more cost-effective to design a system to be robust against the most critical requirements [39].

[2]There isn't a unified interpretation of these "unaccepted" behaviors in LTS. It could be interpreted as "undefined" or "disabled". In this work, we assume the "disabled" interpretation.

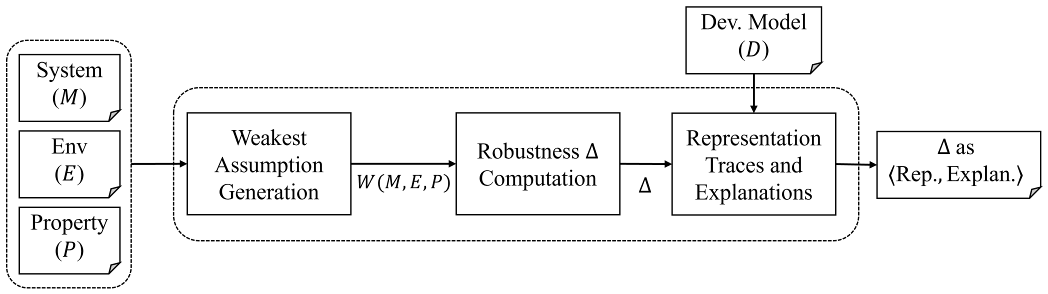
System
(M)
Env
(E)
Property
(P)
Weakest Assumption Generation
$W(M,E,P)$
Robustness Δ Computation
Δ
Dev. Model
(D)
Representation Traces and Explanations
Δ as
⟨Rep., Explan.⟩

Figure 3.4: Overview of the robustness computation process.

3.2.3 Robustness Computation

Overview. Figure 3.4 shows the overall process for the robustness computation. Given the LTS of the machine M, the environment E, and the safety property P, we first compute the *weakest assumption* of M with respect to E and P, which then is used to compute the model of the robustness Δ. In general, Δ may be infinite, and not in a form that can be easily comprehensible by the user. Thus, we generate a succinct representation of it. Specifically, we partition Δ into a finite set of equivalence classes, each of which contains traces that describe the same type of deviation, and then sample *representative traces* from those classes. Finally, we take a *deviation model D* as input to generate *explanations* that describe how the environment could deviate from the normative behavior in a particular way. The final output of the process is a set of pairs of a representative trace and its explanation.

Weakest assumption and robustness. According to our robustness definition (Definition 3.2.2) and the illustration in Figure 3.3, robustness consists of two sets of behaviors: (1) the deviations accepted by the machine and under which the machine can guarantee the safety property, and (2) the deviations that are disabled by the machine. In [71], we present an approach to compute it using the weakest assumption of a machine.

In assume-guarantee style of reasoning [34], given a machine M, the environment E, and a property P, the *weakest assumption* is the largest possible environmental behavior under which the machine satisfies the property. More formally:

Definition 3.2.3 (Weakest Assumption) *The weakest assumption of machine M with respect to environment E and property P, denoted by $W(M,E,P)$, is an LTS such that*

$$\forall E' : E' || M \models P \Leftrightarrow E' \models W(M,E,P)$$

We will simply use W to refer $W(M,E,P)$ if the context is unambiguous. Therefore, given this definition, the weakest assumption W should include all the behaviors in $beh(M_{safe})$ plus the behaviors in $\delta_2 = \alpha A^* \setminus beh(M)$, as shown in Figure 3.3. Thus, the robustness of a machine is equivalent to its weakest assumption *minus* the behaviors of the normative environment, i.e.,

$$\Delta(M,E,P) = beh(W) \setminus beh(E) \tag{3.1}$$

In our work, we leverage the approach by Giannakopoulou et al. [33] to generate the weakest assumption.

Robustness comparison. Base on the robustness computation process, we can also define the robustness comparison process. Specifically, to compare the robustness of two machine designs, the set $\mathcal{X} = \Delta(M_2,E,P) - \Delta(M_1,E,P)$ can be computed by:

$$\mathcal{X} = beh(W(M_2,E,P)) \setminus beh(W(M_1,E,P)) \tag{3.2}$$

Similarly, to compare the robustness of one machine under two different properties, the set $\mathcal{X} = \Delta(M, E, P_2) - \Delta(M, E, P_1)$ can by computed by:

$$\mathcal{X} = beh(W(M, E, P_2)) \setminus beh(W(M, E, P_1)) \tag{3.3}$$

Representative traces. Given the computation process, robustness is computed as $\Delta = beh(W) \setminus beh(E)$. In general, this Δ may represent infinite number of traces and may not be readily comprehensible by the user without a proper representation. Therefore, we propose a succinct, finite representation of robustness. The key idea behind this is that many traces in Δ capture a similar type of deviation (e.g., message loss in a network communication system or a human operator mistakenly skipping an action), and thus can be grouped into a same *equivalence class*. Then, we can generate one or multiple *representative trace(s)* from each equivalence class to represent a group of behaviors.

For example, in the radiation therapy machine, we can use trace $\langle X, B\rangle$ to represent an equivalence class that contains a group of deviations that all have $\langle X, B\rangle$ as their prefix, where $\langle X\rangle$ is the shortest normative trace and B is the first action to deviate from the normative environment. More details about the representative-trace generation can be found in our paper [71].

Deviation explanations. Representative traces describe how the environment deviates from the normative environment as *observed* by machine M. However, they do not capture how the internal *faults* of the environment could have caused this deviation. Therefore, we propose a method for augmenting the representative traces with additional domain-specific information about the underlying causes behind the deviations by introducing a *deviation* model.

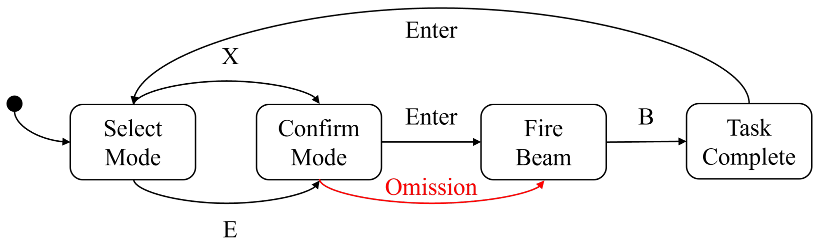

Figure 3.5: A deviation model D for the radiation therapy machine.

For example, Figure 3.5 is a deviation model for the radiation therapy machine. It specifies that from state ConfirmMode, the operator might commit a type of error called *omission error* [17, 16], i.e., omitting *Enter* action and pressing B instead, which is specified as the additional transition from ConfirmMode to FireBeam on an faulty event *Omission*. Then, we would generate an explanation for the representative trace $\langle X, B\rangle$ as $\langle X, Omission, B\rangle$.

3.3 Robustification by Wrapper

This section provides an overview of our work on robustifying system designs by wrapper [73]. In Section 3.3.1, we first introduce the necessary background on supervisory control, which is the underlying technique we rely on to synthesize new designs. Section 3.3.2 and Section 3.3.3 describes the definitions of our robustification problems. Finally, Section 3.3.4 presents the algorithm to solve the robustification problems.

3.3.1 Supervisory Control

Our proposed robustification approach leverages techniques from an area of control theory called *supervisory control* [20]. In the context of supervisory control, it assumes an "uncontrolled" system (also called *plant*), which in our context would be the composition of a machine and its environment

$(M||E)$, 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. The control or restriction is done by a component named *supervisory controller*, which can observe certain events in the plant and disable some events to occur.

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. Therefore, we can formally define a controller as follows:

Definition 3.3.1 (Supervisory Controller) *A supervisory controller is a function*

$$S : beh(G{\upharpoonright}\alpha G_o) \rightarrow 2^{\alpha G} \ \ s.t. \ \forall \sigma \in beh(G{\upharpoonright}\alpha G_o) : \alpha G_{uc} \subseteq S(\sigma)$$

From this definition, the control enforced by a controller can change only after some observable event occurs. Also, in our work, 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 3.3.2 *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 $S||G \models P$.*

The synthesis should generate a controller that is *minimally restrictive*, i.e., it should disable only the necessary transitions that would eventually result in a property violation, and retain as much behavior as possible of the original plant. Supervisory control theory provides algorithmic techniques for computing such a controller; more details can be found in [20].

3.3.2 Basic Robustification Problem

According to our robustness definition, we say a machine is robust against a set of deviations δ when give an environment E' such that $beh(E') = beh(E) \cup \delta$, the machine continues to satisfy the property, i.e., $M||E' \models P$. In contrast, robustification deals with the opposite case, i.e., given some *intolerable* deviations $\bar{\delta}$ such that $M||E' \not\models P$, find a new machine M' such that $M'||E' \models P$; and the process for finding the new design M' is called *robustification*.[3]

Formally, we define the process of *robustifying* a design as follows:

Definition 3.3.3 *Given machine M, environment E, intolerable deviations $\bar{\delta}$, and property P such that $M||E \models P$, and let E' be the deviated environment such that $beh(E') = beh(E) \cup \bar{\delta}$ and $M||E' \not\models P$, the goal of robustification, $Rb(M, E, \bar{\delta}, P)$, is to find an LTS M' such that $M'||E' \models P$.*

For example, Figure 3.6 is another deviated environment E' under which the original design M is not robust. Specifically, a counterexample $\langle X, Commission, Up, E, Enter, B\rangle$ depicts a scenario where the user mistakenly selects the X-ray mode and uses Up to correct the selection and then fire the beam. However, the beam mode may still be in transition from X-ray while the spreader is out of place which causes a safety violation. Therefore, the robustification problem defines the process

[3]To distinguish from the deviations δ that the machine is robust against, we will use $\bar{\delta}$ to represent deviations that the original machine is not robust against, namely intolerable deviations.

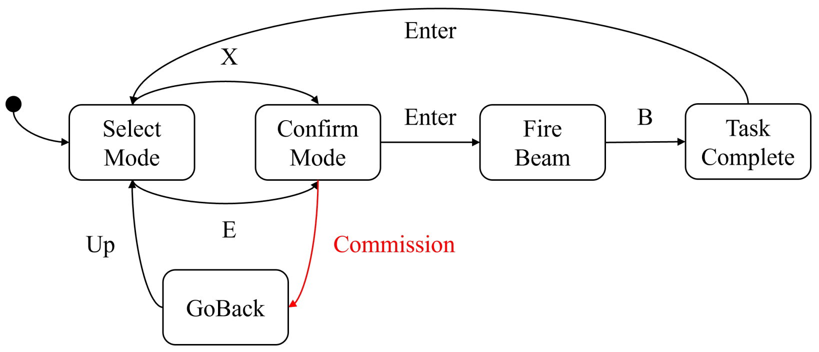

Figure 3.6: A deviated environment model E' with *commission errors* for the radiation therapy machine, under which the original design M is not robust.

to find a new therapy machine design M' such that M' is robust against this deviated environment. A solution to this problem is the redesign shown in Figure 3.7 where the interface will synchronize on the beam mode to be correctly set.

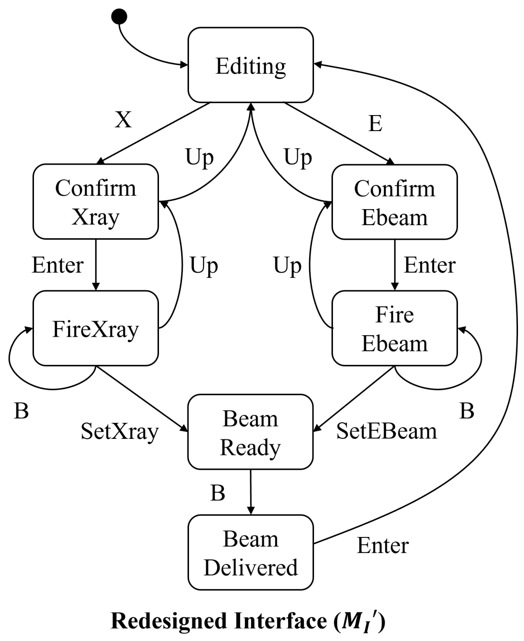

Figure 3.7: A redesign of the radiation therapy machine. In particular, we show only the redesigned interface where the operator can fire the beam until the mode switch has completed.

3.3.3 Optimal Robustification Problem

However, not every solution to the basic robustification problem may be desirable to the developer, especially when we focus on safety properties. Safety properties often specify that a system should not enter some bad states, a naive way to robustify any system would be removing behaviors; however, this may also result in a system that can do nothing useful.

For example, to robustify the radiation therapy machine against the deviated environment in Figure 3.6, one could remove all the B transitions; however, in this way, the machine can never fire the beam to start a treatment. One could also remove all the *Up* transitions from the interface; however, it does not allow the operator to ever change the beam mode. Neither of these redesigns would be desirable, in that although they can guarantee safety, they also give up on essential functionalities of the system.

Quality metrics for robustified designs. To enable generation of more "desirable" solutions, we consider two quality metrics for robustified redesigns: (1) the redesign M' should retain as much of the important functionality from the original design M as possible, and (2) the cost of modifying M to M' should be small.

For the first quality, we introduce the notion of *preferred behavior*.

Definition 3.3.4 *A preferred behavior v is an execution trace that represents an operational scenario that the developer wishes an LTS T to contain, i.e., $v \in beh(T \upharpoonright \alpha v)$, where αv refers to the events in trace v.*

For simplicity of notation, we denote it by $v \models T$. Then, retaining as much behavior from the original design as possible can be formulated as maximizing the number of preferred behavior v's such that $v \models M||E$ and $v \models M'||E'$. Formally:

Definition 3.3.5 *Give a set of preferred behaviors $V = \{v_1, v_2, \ldots, v_n\}$, we state $V \models T$ for some LTS T if and only if $\bigwedge_{v_i \in V} v_i \models T$.*

Then, we can associate each scenario v_i with a different importance value to quantitatively measure the amount of behavior retained by M' in terms of the total importance value of the subset of preferred behaviors $V' \subseteq V$ that satisfy $V' \models M'||E'$.

For example, in the radiation therapy machine, one can define preferred behaviors:

$$v_1 : \langle X, Up, E, Enter, B\rangle, \text{ and } v_2 : \langle E, Up, X, Enter, B\rangle$$

to specify the desire that the user should be able to switch from X-ray mode to Electron beam mode using the *Up* button, and vice-versa.

For the second property, we propose to use the set of environment and machine events that are observed or controlled by the machine to approximate the cost of a design. Intuitively, to observe certain events, the machine should implement proper detectors or sensors; and it is often more costly to control (enable or disable) certain events to occur. More precisely, we consider a pair of event sets, $A = (A_c, A_o)$, where $A_c, A_o \subseteq \alpha E \cup \alpha M$, that are controllable and observable, respectively, for the purpose of robustification. Furthermore, each event in A is associated with a cost measure to reflect the effort of implementing its actuation or sensing capability. Thus, the total cost of robustification can be measured as the sum of the individual costs of the events in $A' \subseteq A$ that are used to modify the machine.

For example, one could assign a moderate cost to event *SetXray* and *SetEBeam* for observability to reflect the cost of changes to implement sensing capability in the interface to synchronize on mode switching. They could also assign a minor cost to event B for controllability to reflect the cost to disable B accordingly to avoid accidentally firing the wrong beam.

Robustification as multi-objective optimization. Intuitively, a larger set of events for robustifying a machine allows a more fine-grained control, which can help retain more behavior from the original design but also leads to a higher cost. Thus, given the two quality metrics, the optimal robustification problem can be formulated as a multi-objective optimization problem where developers need to balance the trade-offs between maximizing preferred behaviors and minimizing cost of changes.

Specifically, given a robustification problem $Rb(M, E, \bar{\delta}, P)$, preferred behaviors V, and modifiable events A, let $\vec{R} = \langle M', V', A'\rangle$ be a solution such that M' is a valid robustified design, $V' \subseteq V$ is the satisfied preferred behavior, and $A' = (A'_c, A'_o)$ is the subset of events used for robustification. Then, we can define the following objective function:

$$\vec{U}(\langle M', V', A'\rangle) = \langle U_V(V'), U_A(A')\rangle$$

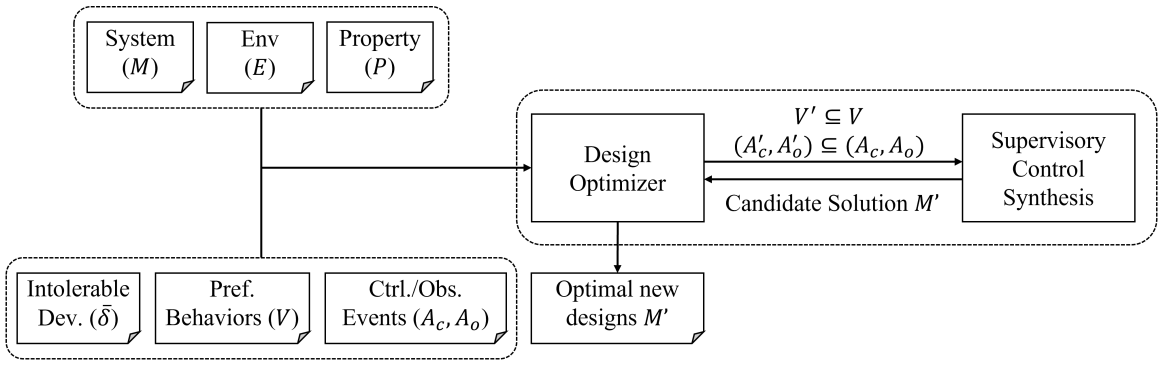

Figure 3.8: The overall process for solving optimal robustification problems.

where $U_V(V')$ is the amount of utility gained from fulfilling the preferred behaviors, which should be positive, and $U_A(A')$ is the total cost of events used to redesign M', which should be negative.

Finally, the optimal robustification problem can be defined as follows:

Definition 3.3.6 *Given a robustification problem Rb, a set of preferred behaviors V such that $V \models M||E$, and a set of available events for robustification A, the goal of $Opt(Rb, V, A)$ is to find one or more solutions $\vec{R} = \langle M', V', A' \rangle$ such that M' is a solution to problem Rb, $V' \models M'||E'$, and $\vec{R}$ maximizes the objective function $\vec{U}$.*

There could be more than one optimal solutions to this problem. It would be hard for developers to mentally compute the trade-offs between different design alternatives. Therefore, we then propose an algorithm that leverages supervisory control synthesis to generate a set of optimal redesigns.

3.3.4 Optimal Robustification Method

First, we show that a basic robustification problem can be reduced to a supervisory controller synthesis problem as follows:

Theorem 3.3.1 *Given a basic robustification problem $Rb(M, E, \bar{\delta}, P)$, let S be a solution to the controller synthesis problem $C(G, P, \alpha G_c, \alpha G_o)$, where $G = M||E'$ and $\alpha G_c \subseteq \alpha G_o \subseteq \alpha G$. Then, $M' = S||M$ is a solution to the robustification problem, where $\alpha M' = \alpha M \cup \alpha G_c \cup \alpha G_o$.*

Then, given the characteristics of supervisory control synthesis, by default, it generates the minimally restrictive controller (Definition 3.3.2), which aligns with our goal of retaining as much behavior as possible. Also, the use of controllable and observable events to synthesize a controller aligns with our goal of using actuation and sensing capability to measure the cost. More precisely, we have the following theorem:

Theorem 3.3.2 *Given an optimal robustification problem $Opt(Rb, V, A)$ and a corresponding controller synthesis problem $C(G, P, \alpha G_c, \alpha G_o)$, supervisory controller synthesis generates a controller S such that $M' = S||M$ satisfies the maximal possible $V' \subseteq V$ for $A' = (\alpha G_c, \alpha G_o)$.*

Therefore, we formulate an algorithm to solve the optimal robustification problem by using supervisory controller synthesis as a searching primitive. Figure 3.8 illustrates the overall process of our approach. At a high-level, a design optimizer generates the next searching target (V', A'). Subsequently, we solve a supervisory control synthesis problem with the given A' and verify if M' satisfies V'. The design optimizer stores this candidate solution M' and iteratively generates the next searching target. Ultimately, the optimizer produces all optimal new designs.

We leverage Supremica [50], a state-of-the-art solver, to solve the synthesis problems, and the key contributions of our algorithm are heuristics for efficiently searching candidate tuples of (V', A') to improve the overall performance of the solving process. Details of our solving process and heuristics can be found in [73].

3.4 Evaluation of Current Approach

We implement our proposed robustness computation and robustification by wrapper approach into a tool named *Fortis*[4]. We then illustrate the applicability and performance of our approach through experiments over a set of case studies.

3.4.1 Case Studies

Except the radiation therapy machine, we briefly summarize other case studies we used. Details about the case studies can be found in [70, 71, 73].

Voting Machine. We consider a simplified design of an electronic voting system (called ES&S iVotronic, described in more detail in [67]), that was used in several state-wide elections in the U.S. and was involved in an election fraud [60]. 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. This would allow a malicious official to enter the booth and change the voter's vote. Thus, in our model, we set a safety property to guarantee that the machine should record each voter's selection exactly as made by that voter.

Network Protocols. Consider the problem of transmitting a sequence of messages between a pair of nodes (*sender* and *receiver*). 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) [66], which is designed to guarantee integrity of messages over unreliable channels (e.g., message loss or duplication). We specify the desired property as *the input and output should alternate*, i.e., the sender sends a new message only after the receiver receives the previous one.

Oyster. We consider the *Oyster* card fare collection protocol used in public transportation in London, UK (described in [62]). 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.

Infusion Pump. We model an infusion pump machine for dispensing medication to patients through tube lines [15]. The machine also includes a built-in battery that activates when the power cable is unplugged. Normally, the operator plugs in the device, configures the medication dose and starts the dispensation. However, 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 is to guarantee that *if the machine loses power, it should immediately stop any on-going dispensation.*

3.4.2 Evaluation Results

For each of the above case studies, we used Fortis to (1) compute the robustness of the system and (2) synthesize robustified designs against a deviated environment model.

Table 3.1 summarizes the results for robustness computation. It shows that although the worst-case complexity of the computation process 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.

[4]`https://github.com/cmu-soda/fortis-core`

Table 3.1: Evaluation results of robustness computation. All problems were run on a Linux Machine with a 3.6GHz CPU and 24GB memory under a 30-minute timeout.

	$\|M\|\|P\|^*$	Time (s)
Therapy	20	0.025
Naive	41	0.029
ABP	23	0.033
Voting-1**	53	0.033
Voting-2	277	0.066
Voting-3	821	0.106
Voting-4	1,829	0.188
Pump-1	163	0.036
Pump-2	1,679	0.149
Pump-3	19,435	1.227
Oyster	1,729	0.280

* $|M||P|$ 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.

Table 3.2 provides a summary of the results for robustification using the wrapper tactic. Robustification poses a significantly more complex problem with a much larger search space. The worst-case complexity is exponential in the number of states of machine M and the deviated environment E', in addition to the number of preferred behaviors V and controllable/observable events A—expressed as $O(2^{|V|+|A|+|M||E'|})$. Through our experiments, it became evident that controller synthesis often becomes the bottleneck. The time to solve one synthesis instance increases rapidly with the growing size of the system. Moreover, for the same problem, synthesis becomes more challenging as fewer controllable and observable events are provided, while minimizing the cost.

In comparison to a naive searching strategy using brute-force (depicted under the *Naive* columns), Fortis addresses the performance challenges by introducing several search heuristics. These heuristics aim to prune the search space and reduce the number of synthesis calls, as detailed in [73]. As shown in the *With heuristics* columns, our search heuristics effectively reduce the required number of synthesis calls and significantly improve the overall searching performance. Notably, in *Voting-2,3* and *Pump-2,3*, the naive search approach timed out after 30 minutes, while our search heuristics successfully found solutions.

Table 3.2: Evaluation results of robustification. All problems were run on a Linux Machine with a 3.6GHz CPU and 24GB memory under a 30-minute timeout.

				Naive		With heuristics	
	$\|V\|^*$	$\|A\|$	$\|M\|\|E'\|$	#Synth.	Time (s)	#Synth.	Time (s)
Therapy	4	5	21	32	0.812	6	0.469
Naive	2	8	14	1	0.226	1	0.242
ABP**	-	-	-	-	-	-	-
Voting-1	1	13	12	2,576	24.100	9	0.507
Voting-2	1	23	31	-	T/O	16	1.908
Voting-3	1	32	44	-	T/O	21	20.172
Voting-4	1	41	57	-	T/O	-	T/O
Pump-1	2	12	104	2,304	59.584	13	1.129
Pump-2	4	16	760	-	T/O	17	10.817
Pump-3	6	20	6,248	-	T/O	21	457.839
Oyster	2	4	900	16	1.799	1	0.686

* $|V|$ is the number of preferred behaviors, $|A|$ 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 $O(2^{|V|+|A|+|M||E'|})$. #Synth. is the number of calls to the controller synthesizer.

** Robustification is not applicable to ABP as it already satisfies P under the given deviations.

Chapter 4

Proposed Work

4.1 Robustification by Weakening

The wrapper robustification approach leverages supervisory control theory to synthesize a controller (wrapper) that observes and controls the inputs and outputs of a machine. Formally, for a given problem $M||E' \not\models P$, we synthesize a wrapper C such that $M'||E' \models P$, where $M' = M||C$. However, in cases with a strong safety property, we may not be able to find such a controller, or the resulting design M' might become overly restrictive, leading to the loss of certain functionalities. Another challenge arises when the wrapper demands additional observability and controllability of the machine, resulting in a high cost of changes.

To address these challenges, we introduce another robustification tactic alongside the wrapper tactic, called *specification weakening*. The goal of weakening is to identify a *weaker* property P' such that, after applying the wrapper tactic against P', we can find a new design that is not excessively restrictive or is cost-effective. Formally, for a given problem $M||E' \not\models P$, we aim to find a weaker property P' such that $P \Rightarrow P'$, and we can generate a new design M' with the wrapper tactic such that $M'||E' \models P'$. Additionally, this M' should be less restrictive or more cost-effective than a solution generated by only the wrapper tactic.

For instance, in the radiation therapy machine example, we can consider a stronger safety property that includes two aspects: (1) over-dose prevention, the spreader must be in place when the beam is delivered in X-ray mode, and (2) under-dose prevention, the spreader must be out of place when the beam is delivered in Electron beam mode. This can be formally defined as:

$$\mathbf{G}\big(\mathit{BeamDelivered} \Rightarrow (\mathit{XrayMode} \Rightarrow \mathit{InPlace}) \wedge (\mathit{EBeamMode} \Rightarrow \mathit{OutPlace})\big)$$

The initial design depicted in Figure 3.1 is not robust against the following two classes of deviations: (1) $\langle X, Up, E, Enter, B\rangle$, where the user switches from X-ray to Electron beam, potentially resulting in an over-dose issue; and (2) $\langle E, Up, X, Enter, B\rangle$, where the user switches from Electron beam to X-ray, potentially leading to an under-dose issue. By applying our robustification by wrapper tactic, feasible solutions include:

1. Observing $SetXray$ and $SetEBeam$ events to synchronize on the completion of mode switching and disabling Up button accordingly to avoid firing the wrong beam.

2. Disabling all Up transitions to disallow mode switching so that there's no need to observe the $SetXray$ and $SetEBeam$ events.

3. Observing only one of the Set event to allow mode switching from either X-ray to Electron beam or vice-versa.

The wrapper robustification method incorporates a simplified form of specification weakening. We have introduced the concept of preferred behavior, which is a lightweight representation of liveness properties. The robustification process permits giving up certain preferred behaviors to produce a wrapper with a reduced cost but that disables more transitions. In the aforementioned example, one could specify $\langle X, Up, E, Enter, B\rangle$ and $\langle E, Up, X, Enter, B\rangle$ as preferred behaviors, and solutions 2 and 3 choose to forgo these preferred behaviors.

However, for the specification weakening tactic, we allow the safety property to be weakened as well. For instance, the under-dose scenario caused by $\langle E, Up, X, Enter, B\rangle$ might be considered less critical and acceptable. Therefore, by weakening the safety property and removing the under-dose case, we could identify another feasible solution where the system does not need to observe the $SetEBeam$ event and disable Up for the under-dose scenario. In essence, the weakening of the safety property increases the space of feasible solutions, enabling us to find solutions with a lower cost or those that satisfy additional preferred behaviors.

The concept of weakening originates from requirements engineering [3], when environmental conditions are changed over time and space, the original requirements may not be adequate and consistent thus must be adapted. One key challenge of requirements weakening is how to formally define weakening and synthesize a weakened property. Prior research has explored weakening with goal modeling [3, 44], Fuzzy Temporal Logic [69], LTL GR(1) specification learning [19], and Signal Temporal Logic [23]. However, these approaches do not directly apply to our case as we consider safety properties defined in LTS. Therefore, a critical research question for us is to develop a new methodology for weakening a safety property defined in LTS.

4.2 Fortis: A Tool for Robustness Analysis

Fortis [70] is the tool that implements all our proposed robustness analysis and robustification techniques. We aim to make it an extensible toolbox and framework for robustness analysis. Its current implementation primarily focuses on realizing our proposed approaches efficiently. However, its extensibility and usability have not been adequately supported. Therefore, another stretch goal of this work is to enhance the quality of Fortis.

For instance, our existing implementation provides a simple GUI to allow basic interaction capabilities for users. We could expand this GUI to improve its usability and integrate visualizations for system models and repairs. However, this not only involves engineering effort but also poses a research challenge in intelligently visualizing a synthesized repair and effectively explaining it to users.

4.3 Large-Scale Case Study: Open EMR

Another stretch goal of this work is to apply our approaches to a large-scale, real-world case study. One promising case study is Open EMR[1], an open source electronic health records and medical practice management software. Over years of development contributed and maintained by the open source community, it provides a rich set of features including: scheduling, e-Prescribing, medical billing, clinical decision rules, lab integration, and reporting. Therefore, we can explore how well our proposed techniques can be applied to such a real-world medical software.

The research task involves understanding the usage and implementation of Open EMR and creating proper abstract models of it. Finally, we also need to identify meaningful safety properties for our robustness analysis. New research challenges might also emerge during our investigation.

[1]`https://www.open-emr.org/`

Bibliography

[1] Ieee standard glossary of software engineering terminology. IEEE Std 610.12-1990 pp. 1–84 (1990). https://doi.org/10.1109/IEEESTD.1990.101064

[2] Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1) (jan 2018). https://doi.org/10.1145/3158668, `https://doi.org/10.1145/3158668`

[3] Alrajeh, D., Cailliau, A., van Lamsweerde, A.: Adapting requirements models to varying environments. In: International Conference on Software Engineering (ICSE). pp. 50–61. ACM (2020)

[4] Arcile, J., Devillers, R., Klaudel, H., Klaudel, W., Woźna-Szcześniak, B.: Modeling and checking robustness of communicating autonomous vehicles. In: Omatu, S., Rodríguez, S., Villarrubia, G., Faria, P., Sitek, P., Prieto, J. (eds.) Distributed Computing and Artificial Intelligence, 14th International Conference. pp. 173–180. Springer International Publishing, Cham (2018)

[5] Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1**(1), 11–33 (2004). https://doi.org/10.1109/TDSC.2004.2

[6] Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model Checking Probabilistic Systems, pp. 963–999. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28, `https://doi.org/10.1007/978-3-319-10575-8_28`

[7] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos engineering. IEEE Software **33**(3), 35–41 (2016). https://doi.org/10.1109/MS.2016.60

[8] Bergstra, J., Ponse, A., Smolka, S. (eds.): Handbook of Process Algebra. Elsevier Science, Amsterdam (2001). https://doi.org/https://doi.org/10.1016/B978-044482830-9/50019-9

[9] Bishop, M., Elliott, C.: Robust programming by example. In: Dodge, R.C., Futcher, L. (eds.) Information Assurance and Security Education and Training. pp. 140–147. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[10] Bloch, J.: Effective Java: Best practices for the Java Platform. Addison-Wesley Professional (2017)

[11] Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Robustness in the presence of liveness. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification. pp. 410–424. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

[12] Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Jobstmann, B.: Specification-centered robustness. In: 2011 6th IEEE International Symposium on Industrial and Embedded Systems. pp. 176–185 (2011). https://doi.org/10.1109/SIES.2011.5953660

[13] Boehm, B.W.: Software engineering economics. IEEE Transactions on Software Engineering **SE-10**(1), 4–21 (1984). https://doi.org/10.1109/TSE.1984.5010193

[14] Boehme, M., Cadar, C., ROYCHOUDHURY, A.: Fuzzing: Challenges and reflections. IEEE Software **38**(3), 79–86 (2021). https://doi.org/10.1109/MS.2020.3016773

[15] Bolton, M.L., Bass, E.J.: Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking. In: 2011 IEEE International Conference on Systems, Man, and Cybernetics. pp. 1788–1794 (2011). https://doi.org/10.1109/ICSMC.2011.6083931

[16] Bolton, M.L., Bass, E.J.: Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking. IEEE Transactions on Systems, Man, and Cybernetics: Systems **43**(6), 1314–1327 (2013). https://doi.org/10.1109/TSMC.2013.2256129

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

[18] Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by ai techniques. Artificial Intelligence **112**(1), 57–104 (1999)

[19] Buckworth, T., Alrajeh, D., Kramer, J., Uchitel, S.: Adapting specifications for reactive controllers. In: 2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). pp. 1–12 (2023). https://doi.org/10.1109/SEAMS59076.2023.00012

[20] Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer, Cham, 3 edn. (2021)

[21] Chatzieleftheriou, G., Bonakdarpour, B., Smolka, S.A., Katsaros, P.: Abstract model repair. In: Goodloe, A.E., Person, S. (eds.) NASA Formal Methods. pp. 341–355. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_32

[22] Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). Shanghai, China (21-22 May 2006)

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

[24] Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)

[25] Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to Model Checking, pp. 1–26. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_1, `https://doi.org/10.1007/978-3-319-10575-8_1`

[26] Ding, Y., Zhang, Y.: A logic approach for LTL system modification. In: Hacid, M.S., Murray, N.V., Raś, Z.W., Tsumoto, S. (eds.) Foundations of Intelligent Systems. pp. 435–444. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11425274_45

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

[28] Duraes, J.A., Madeira, H.S.: Emulation of software faults: A field data study and a practical approach. IEEE Transactions on Software Engineering **32**(11), 849–867 (2006). https://doi.org/10.1109/TSE.2006.113

[29] Easwaran, A., Kannan, S., Sokolsky, O.: Steering of discrete event systems: Control theory approach. Electronic Notes in Theoretical Computer Science **144**(4), 21–39 (2006), proceedings of the Fifth Workshop on Runtime Verification (RV 2005)

[30] Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer **14**(3), 349–382 (2012). https://doi.org/https://doi.org/10.1007/s10009-011-0196-8

[31] Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design **38**(3), 223–262 (2011). https://doi.org/https://doi.org/10.1007/s10703-011-0114-4

[32] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10), 46–54 (2004). https://doi.org/10.1109/MC.2004.175

[33] Giannakopoulou, D., Pasareanu, C., Barringer, H.: Assumption generation for software component verification. In: Proceedings 17th IEEE International Conference on Automated Software Engineering,. pp. 3–12 (2002). https://doi.org/10.1109/ASE.2002.1114984

[34] Giannakopoulou, D., Namjoshi, K.S., Păsăreanu, C.S.: Compositional Reasoning, pp. 345–383. Springer International Publishing (2018)

[35] Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz Robustness of Finite-state Transducers. In: Raman, V., Suresh, S.P. (eds.) 34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014). Leibniz International Proceedings in Informatics (LIPIcs), vol. 29, pp. 431–443. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.431, `http://drops.dagstuhl.de/opus/volltexte/2014/4861`

[36] Henzinger, T.A., Otop, J., Samanta, R.: Lipschitz robustness of timed i/o systems. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 250–267. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)

[37] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (1969)

[38] Huang, X., Kroening, D., Ruan, W., Sharp, J., Sun, Y., Thamo, E., Wu, M., Yi, X.: A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. Computer Science Review **37**, 100270 (2020). https://doi.org/https://doi.org/10.1016/j.cosrev.2020.100270, `https://www.sciencedirect.com/science/article/pii/S1574013719302527`

[39] Jackson, D., Kang, E.: Separation of concerns for dependable software design. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. p. 173–176. FoSER '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1882362.1882399, `https://doi.org/10.1145/1882362.1882399`

[40] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62

[41] Kobayashi, T., Salay, R., Hasuo, I., Czarnecki, K., Ishikawa, F., Katsumata, S.y.: Robustifying controller specifications of cyber-physical systems against perceptual uncertainty. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) NASA Formal Methods. pp. 198–213. Springer International Publishing, Cham (2021)

[42] Koopman, P., Sung, J., Dingman, C., Siewiorek, D., Marz, T.: Comparing operating systems using robustness benchmarks. In: Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems. pp. 72–79 (1997). https://doi.org/10.1109/RELDIS.1997.632800

[43] Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering **SE-3**(2), 125–143 (1977). https://doi.org/10.1109/TSE.1977.229904

[44] van Lamsweerde, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. IEEE Transactions on Software Engineering **24**(11), 908–926 (1998). https://doi.org/10.1109/32.730542

[45] van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley Publishing, 1st edn. (2009)

[46] Laranjeiro, N., Agnelo, J.a., Bernardino, J.: A systematic review on software robustness assessment. ACM Comput. Surv. **54**(4) (may 2021). https://doi.org/10.1145/3448977, `https://doi.org/10.1145/3448977`

[47] Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems: A Cyber-Physical Systems Approach. The MIT Press, 2nd edn. (2016)

[48] de Lemos, R., Garlan, D., Ghezzi, C., Giese, H., Andersson, J., Litoiu, M., Schmerl, B., Weyns, D., Baresi, L., Bencomo, N., Brun, Y., Camara, J., Calinescu, R., Cohen, M.B., Gorla, A., Grassi, V., Grunske, L., Inverardi, P., Jezequel, J.M., Malek, S., Mirandola, R., Mori, M., Müller, H.A., Rouvoy, R., Rubira, C.M.F., Rutten, E., Shaw, M., Tamburrelli, G., Tamura, G., Villegas, N.M., Vogel, T., Zambonelli, F.: Software engineering for self-adaptive systems: Research challenges in the provision of assurances. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Software Engineering for Self-Adaptive Systems III. Assurances. pp. 3–30. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-74183-3_1

[49] Leveson, N.G., Turner, C.S.: An investigation of the therac-25 accidents. Computer **26**(7), 18–41 (1993). https://doi.org/10.1109/MC.1993.274940

[50] Malik, R., Åkesson, K., Flordal, H., Fabian, M.: Supremica–an efficient tool for large-scale discrete event systems. IFAC-PapersOnLine **50**(1), 5794–5799 (2017), 20th IFAC World Congress

[51] Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563

[52] Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2008)

[53] de Menezes, M.V., do Lago Pereira, S., de Barros, L.N.: System design modification with actions. In: da Rocha Costa, A.C., Vicari, R.M., Tonidandel, F. (eds.) Advances in Artificial Intelligence – SBIA 2010. pp. 31–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16138-4_4

[54] Natella, R., Cotroneo, D., Madeira, H.S.: Assessing dependability with software fault injection: A survey. ACM Comput. Surv. **48**(3) (feb 2016). https://doi.org/10.1145/2841425, `https://doi.org/10.1145/2841425`

[55] Nayak, S.P., Neider, D., Roy, R., Zimmermann, M.: Robust computation tree logic. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods. pp. 538–556. Springer International Publishing, Cham (2022)

[56] Ogheneovo, E.E.: Software dysfunction: Why do software fail? Journal of Computer and Communications **2014** (2014)

[57] Petke, J., Clark, D., Langdon, W.B.: Software robustness: A survey, a theory, and prospects. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1475–1478. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3473133, `https://doi.org/10.1145/3468264.3473133`

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

[59] Rasmussen, J.: Risk management in a dynamic society: a modelling problem. Safety Science **27**(2), 183–213 (1997). https://doi.org/https://doi.org/10.1016/S0925-7535(97)00052-0, `https://www.sciencedirect.com/science/article/pii/S0925753597000520`

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

[61] Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software mutational robustness. Genetic Programming and Evolvable Machines **15**, 281–312 (2014)

[62] Sempreboni, D., Viganò, L.: X-men: A mutation-based approach for the formal analysis of security ceremonies. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 87–104 (2020). https://doi.org/10.1109/EuroSP48549.2020.00014

[63] Shahrokni, A., Feldt, R.: A systematic review of software robustness. Information and Software Technology **55**(1), 1–17 (2013). https://doi.org/https://doi.org/10.1016/j.infsof.2012.06.002, `https://www.sciencedirect.com/science/article/pii/S0950584912001048`, special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010

[64] Tabuada, P., Caliskan, S.Y., Rungger, M., Majumdar, R.: Towards robustness for cyber-physical systems. IEEE Transactions on Automatic Control **59**(12), 3151–3163 (2014). https://doi.org/10.1109/TAC.2014.2351632

[65] Tabuada, P., Neider, D.: Robust linear temporal logic (2015)

[66] Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press, 2 edn. (2000). https://doi.org/10.1017/CBO9781139168724

[67] Tun, T.T., Bennaceur, A., Nuseibeh, B.: OASIS: Weakening user obligations for security-critical systems. In: 2020 IEEE 28th International Requirements Engineering Conference (RE). pp. 113–124 (2020). https://doi.org/10.1109/RE48521.2020.00023

[68] Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability **22**(5), 297–312 (2012). https://doi.org/https://doi.org/10.1002/stvr.456, `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.456`

[69] Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H., Bruel, J.M.: Relax: Incorporating uncertainty into the specification of self-adaptive systems. In: 2009 17th IEEE International Requirements Engineering Conference. pp. 79–88 (2009). https://doi.org/10.1109/RE.2009.36

[70] Zhang, C., Dardik, I., Meira-Góes, R., Garlan, D., Kang, E.: Fortis: A tool for analysis and repair of robust software systems. In: Formal Methods in Computer-Aided Design (FMCAD) (2023)

[71] Zhang, C., Garlan, D., Kang, E.: 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). p. 1–12 (2020)

[72] Zhang, C., Kapoor, P., Meira-Goes, R., Garlan, D., Kang, E., Ganlath, A., Mishra, S., Ammar, N.: Investigating robustness in cyber-physical systems: Specification-centric analysis in the face of system deviations (2023)

[73] Zhang, C., Saluja, T., Meira-Góes, R., Bolton, M., Garlan, D., Kang, E.: Robustification of behavioral designs against environmental deviations. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (2023)