Behavioral Robustness of Software System Designs

Changjian Zhang

CMU-S3D-24-111

November 2024

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Thesis Committee:
Eunsuk Kang, Co-Chair
David Garlan, Co-Chair
Jonathan Aldrich
Sebastian Uchitel (Imperial College and Universidad de Buenos Aires)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This work was supported in part by the NSF awards 2144860, 2319317, 1918140, 1801546, DoD under Contract No. FA8702-15-D-0002, and DARPA under Contract No. FA87501620042. It was also supported by the CAMELOT project (reference POCI-01-0247-FEDER045915) co-financed by the European Regional Development Fund and the Portuguese Foundation for Science and Technology under CMU Portugal. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsoring agencies.

Keywords: Software Design, Software Requirements, Software Specifications, Robustness, Safety, Model-Driven Engineering, Formal Methods, Formal Software Verification, Model Checking, Discrete Systems, Labeled Transition Systems, Supervisory Control, Linear Temporal Logic, LTL Learning.

For my wife and parents.

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 behaviors 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 redesign that is capable of satisfying the property under such a deviated environment. In particular, two tactics, namely *robustification-by-control* and *robustification-by-specification-weakening*, are introduced. The robustification-by-control tactic formulates the robustification problem 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. Then, the specification-weakening tactic is used alongside the control tactic, which allows weakening the property to generate more feasible redesigns that retain more functionality or have a lower cost.

The proposed robustness computation and robustification method 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.

Acknowledgments

When I first set foot in the United States in 2017, it was primarily following my parents' advice—to explore a different culture, learn advanced technologies, and potentially find a job in Silicon Valley after earning a master's degree. At that time, I did not have a strong motivation or clear goal to pursue a Ph.D. degree. When I was a child, people would often ask, "What do you want to be when you grow up?" I vaguely remember answering with an astronaut or a scientist. However, as I grew older, these seemed more like the "fantasies" of a child. Yet, approaching my thirties, I am also surprised to realize that I can now proudly call myself a scientist. Achieving this milestone as well as my childhood's fantasy would not have been possible without the support and guidance of many people, and I want to express my deepest gratitude to them.

First and foremost, I want to thank my two advisors, Eunsuk Kang and David Garlan. Their support began even before I officially started my Ph.D. studies. In 2017, when my goal was still to find a software engineering job in the industry, it was David's course on Models that introduced me to Formal Methods and sparked my interest in pursing a Ph.D. in this field. In 2018, David and Eunsuk co-advised me on my first academic paper. Although it was rejected—after all, I didn't even know what ICSE was at that time—it marked my very first step from being just a programmer to becoming a software scholar. Within just two years, I managed to publish two FSE papers and even received a Distinguished Paper Award. Beyond my personal efforts and a bit of luck, my rapid growth was also heavily due to their guidance. Even though I have decided to return to the industry, I still look forward to the possibility of any future collaborations.

I would also like to thank my other two committee members, Jonathan Aldrich and Sebastian Uchitel, for their invaluable insights and feedback on my thesis. It is easy for one's mindset to be limited by their past work, and Jonathan's and Sebastian's perspectives significantly broadened my views to my thesis. Their suggestions greatly enriched this work.

I am also grateful to many other faculties in the department, including Bradley Schmerl, Ruben Martins, Christian Kästner, Matt Fredrikson, Jeffery Gennari, Manuel (Mel) Rosso-Llopart, David Root, Anthony Lattanze, Matthew Bass, Eduardo Miranda, Vijay Sai Vadlamudi. Some of them mentored me during my master's program at CMU, and although many have since left the department, without their help, I would not have quickly grasped the state of software engineering research and its challenges, nor would I have had the opportunity to purse my Ph.D. studies at CMU. Others not only helped me during my master's studies, but also continued to support me during my Ph.D.

I also want to thank my colleagues, Rômulo Meira-Góes, Parv Kapoor, Ian Dardik, Simon Chu, Yining She, Hongbo Fang, Ao (Leo) Li, Wode (Nimo) Ni, Tobias, Dürschmid, Chenyang Yang, Jane Hsieh, Tarang Saluja, Ryan Wagner, Maria da Loura Casimiro, Ivan Ruchkin, Nianyu Li, Cody Kineer, Roykrong Sukkerd, Javier Cámara, Gabe Moreno, Andres Diaz-Pace, Sumon Biswas, Yiliang Liang, Andy Hammer, Saloni Sinha, Qishen Zhang. I am thankful for all the research and personal support they provided in various situations. Special thanks to Rômulo for helping me overcome a semester-long struggle on the robustification problem. Special thanks to Parv and Ian for their work on those papers. Special thanks to Ivan for his support when I first decided to pursue a Ph.D. and for his ongoing insights into my research. Special thanks to Simon, Yining, Hongbo, Leo, and Nimo for our conversations about life beyond only research.

Finally, I want to express my deepest gratitude to my wife and parents. Without their emotional and material support, I cannot imagine how I would have faced all those challenges of the Ph.D. journey alone. A survey indicates that nearly one-third of Ph.D. students experience depression, anxiety, and other mental health issues. I consider myself fortunate to have the support from my family, allowing me to always face difficulties with optimism. Therefore, this thesis is not only a summary of my six years of doctoral work but also a gift to my family.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 What is Behavioral 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 finance, 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, aircraft, or medical devices [2].

IEEE standards define robustness as *the ability of a software system to continue functioning correctly in the presence of invalid inputs or a stressful environment* [3]. Avizienis et al. [4] further characterize 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 [5, 6]. ML systems refer to systems with ML components that are statistical models trained on 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 thesis is on the robustness of conventional software systems, particularly 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 to and responds to various inputs or events. To precisely delineate the type of robustness studied in this thesis, 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's behavior is typically characterized by the relationship between the inputs and the corresponding outputs, with 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, and an *invalid input* is an input value that falls outside the assumed range. Thus, robustness of this kind studies whether the system (program) would crash, abort abnormally, or produce unexpected outputs (e.g., an irrelevant error code) given certain inputs that are outside the assumed range [7, 8].
- *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 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 [9]. The 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 actual operation and are not included 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 *IO robustness* to indicate the robustness of input-output relationships. In this thesis, our focus is on 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 *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 *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 state that this thesis investigates behavioral robustness, which 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. It is widely acknowledged by industrial practitioners and academics that *the longer an issue lingers in the system, the more effort it requires to resolve* [10]. However, historically, researchers have observed that the significance of system design is often overlooked in software engineering practices, with engineers placing greater emphasis on the implementation and testing phases [2]. More recent research shows that, on average, engineers spend 25% of their discussions on design topics [11, 12]. However, these design discussions are scattered across commits, issues, and pull requests [13, 14]. Developers often do not produce specific design artifacts, and systematic design analysis is still lacking.

The importance of software design can be stated as follows: a good design fosters the quality of a software system at a time when errors, omissions, and inconsistencies are relatively cheap and easy to correct [15]. In our context of software robustness, systematic design analysis allows developers to identify potential robustness vulnerabilities at an early stage of the development process. Changes made to the design to improve robustness are generally less expensive than those made during implementation or testing. Moreover, for safety-critical or mission-critical systems, establishing a "correct" design that ensures 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 thesis specifically focuses on the systematic analysis of behavioral robustness in 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 invalid inputs or faults

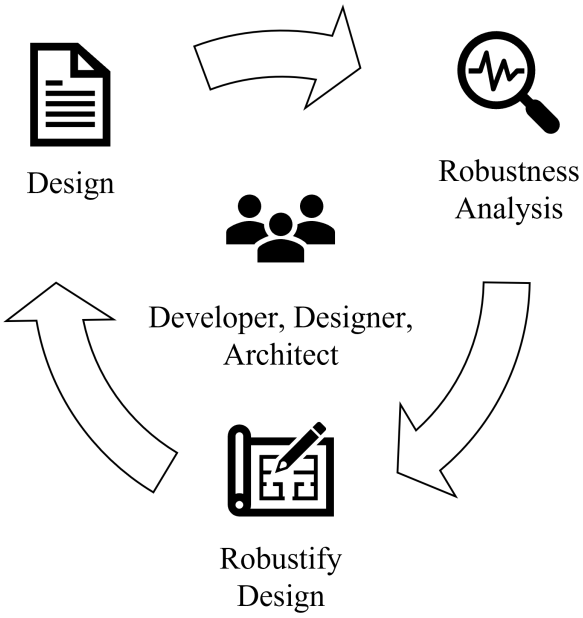


Figure 1.1: Robust-by-design development process.

that lead to robustness violations but may also quantitatively or qualitatively assess 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 [16], existing techniques in software robustness lack sufficient support for this process.

1.3 Robustness Assessment and Improvement

As outlined above, the robust-by-design process comprises two crucial activities: robustness assessment and robustness improvement. According to the surveys by Shahrokni et al. [17] and Laranjeiro et al. [18], robustness assessment techniques for conventional soft-

ware 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 thesis.

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. In contrast, the self-adaptive system community [19] and the run-time assurance community [20] explore how to ensure the run-time correctness of a system, which aligns more closely with our notion of behavioral robustness.

Robustness assessment techniques. From the surveys [17, 18], existing testing-based robustness assessment methods typically answer the following questions:

> *Is the machine (implementation) robust against certain types of invalid inputs or faults? Will the machine crash, abort abnormally, or return unexpected 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 unexpected outputs or exhibits unexpected behavior, such as crashes. The most prevalent evaluation methods include fault injection [21, 22], where developers intentionally introduce specific types of faults to the machine or its environment, and model-based testing [23], where a formal model is employed to generate test cases executed on the concrete machine implementation. Additional methods include fuzzing [24, 25], which automatically and randomly generates extensive inputs, model-based analysis, which formally models and verifies the robustness of a machine [26], and mutation testing, which aims to enhance the quality of test cases for detecting more faults [27, 28, 29].

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 more 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 [17].

Robustness improvement techniques. In their survey, Shahrokni et al. [17] 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 outputs and prevent the propagation of errors. Similar methods are also advocated in programming guidelines such as defensive/robust programming [30] and practitioner-oriented books like [31, 32]. 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.*

Moreover, much recent research has shifted from general robustness techniques to more domain-specific robustness techniques, as different domains often face diverse challenges in robustness [33], such as robust machine learning systems [34] and robust CPS [35]. However, they are outside the scope of our focus on the robustness of conventional software systems.

On the other hand, contributions from the self-adaptive systems community [19] and the run-time assurance community [20] are closer to our research focus. They present methods on how to ensure that a machine maintains its functionalities or quality attributes at run time against uncertainties. Some of these approaches can be seen as ways of improving the 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 [19, 36]. For example, one of their primary interests is studying how to improve a system's resilience or recoverability in the face of uncertainties, which can also be seen as a type of robustness. 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 adaptation or assurance.

1.4 Thesis Statement

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

The *thesis statement* is:

> **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.**

We further elaborate on the thesis statement.

Behavioral robustness of software design. This thesis 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 (specifically, a safety property) defines the intended behavior of the 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. In sum, this thesis investigates the *behavioral robustness* of a machine design, which captures its ability to continue satisfying a desired system-level safety property in the presence of unexpected sequences of events from its actual operating environment.

Formal reasoning. This thesis leverages formal methods for systematically and rigorously quantifying and improving robustness. Specifically, a formal state-transition system serves as a mathematical representation of 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 an acknowledgment 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, our approach mathematically analyzes the behavioral robustness of a machine as a set of traces. Moreover, the formal model enables systematic and automated enhancement of robustness by generating a more robust machine model through model modification and synthesis.

Safety property vs. Liveness property. This thesis primarily focuses on behavioral robustness with respect to a safety property. A safety property specifies the bad states that a system should avoid (e.g., an overdose in a medical device or inconsistent data records in a distributed system). It is widely studied in the literature and is often used to express developers' objectives of preventing bad behavior in a system. We study the robustness problem of whether a machine can continue to avoid those bad states even under unexpected sequences of events from the environment, which is particularly crucial for safety-critical and mission-critical systems. Nevertheless, we do not completely disregard liveness in our approach, especially concerning robustness improvement. Given that a system simply losing all its functionalities and entering a termination state is also deemed safe but practically useless, we recognize the importance of incorporating liveness (or retaining critical functionalities) as an important dimension when enhancing robustness.

1.5 Contributions Overview

This thesis makes the following contributions: (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 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 can 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 primarily consider safety properties in terms of LTS—a safety property in LTS is an LTS that describes a set of "good" 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.

1.5.1 Robustness Definition and Analysis

In this thesis, we propose a formal behavioral notion of software robustness based on LTS. Given a machine M, a normal environment E, and a safety property P, each modeled as an LTS, we assume that the machine satisfies the safety property under the normal 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. Finally, 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. 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 communication 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 rigorous robustness comparisons, which are challenging to conduct using testing-based methods. Additionally, in general, robustness Δ may contain an infinite number of traces, which are not easily comprehensible by developers. Thus, we present an approach to partition Δ into a finite set of *equivalence classes* and sample *representative traces* from them, where 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 consisting of a representative trace and its corresponding explanation. Chapter 2 describes our approach in more detail. This work has also been published at ESEC/FSE 2020 [37].

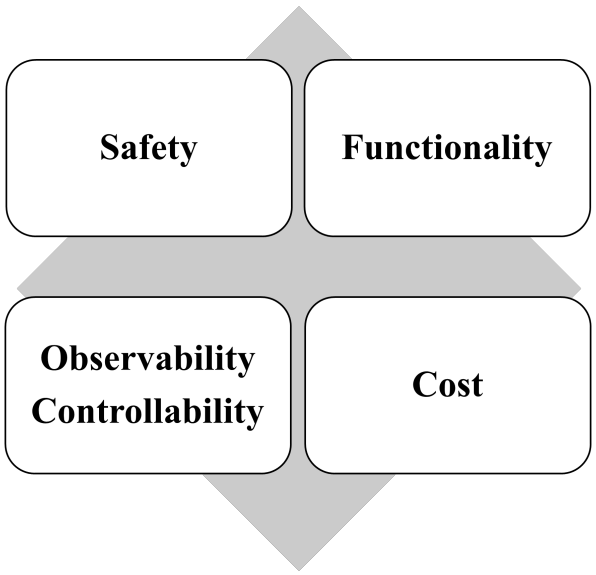


Figure 1.2: Trade-off dimensions of our robustification approach.

1.5.2 Design Robustification

We present 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* 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 trade-off dimensions of robustification: safety, functionality, controllability and observability, and cost.

As defined in the thesis statement, our primary focus is on robustness with respect to a safety property. However, ensuring safety may compromise system functionality by losing certain desired functions. In an extreme case, safety can be trivially achieved by allowing a machine to enter a termination state and do nothing. Therefore, our robustification method also considers maintaining the critical functionality of a system as much as possible.

However, there are scenarios where certain safety properties and system functionality cannot be satisfied simultaneously. One of the issues we consider in this thesis is the lack of controllability and observability 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 faults. More specifically, a machine with more controllability and observability can better distinguish between safe and unsafe executions and prevent the machine from entering unsafe scenarios. However, enhanced controllability and observability often come at a higher cost. Therefore, our method also considers balancing improved controllability and observability (for ensuring safety and retaining more functionality) with implementation costs.

On the other hand, even with sufficient controllability and observability, certain safety requirements and functional requirements still cannot be satisfied simultaneously due to

the underlying conflicts in requirements. Such inconsistencies in requirements may arise from multiple causes, such as conflicts of interests among different stakeholders, requirements evolution, or unforeseen technical constraints [38]. Therefore, rather than always choosing to sacrifice system functionality, we also present an approach to resolve requirement conflicts by weakening the safety specifications. With a weaker safety requirement, we are then able to find a new design that is robust against environmental deviations with respect to the new safety property while retaining more functionality.

Specifically, our proposed robustification method consists of two components:

- *Robustification by control.* 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 (i.e., system functionality), we allow the extension of controllability and observability of the new machine; however, this comes 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 the cost of changes, measured by the extended observing and control abilities. We leverage supervisory control theory [39] and introduce a novel algorithm and several heuristics to search for optimal redesigns. Chapter 3 provides a detailed description of this technique. This work has also been published at ICSE 2023 [40].
- *Robustification by specification weakening.* The robustification-by-control method may potentially disable certain critical machine functionalities, even with extended controllability and observability. We observe that one reason for such situations is the conflicts between safety requirements and functional requirements. In particular, we focus on the cases where the desired safety property is exceptionally strong (restrictive). Therefore, the second method explores *specification weakening* as an additional tactic alongside the robustification-by-control tactic. Intuitively, by employing a weaker safety property, the system should be capable of tolerating more deviations, as certain deviations would no longer be deemed to cause a safety violation. Consequently, the new method involves weakening a potentially overly strong safety property, allowing for improved robustness through the application of the robustification-by-control method with respect to the weakened safety property, without disabling critical system functionalities. We leverage our work on constrained LTL learning [41, 42] to weaken a safety property. Chapter 4 provides a detailed description of this technique.

1.5.3 Implementation and Evaluation

We implement all our proposed approaches in a tool named *Fortis*, which was published at FMCAD 2023 [43]. 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

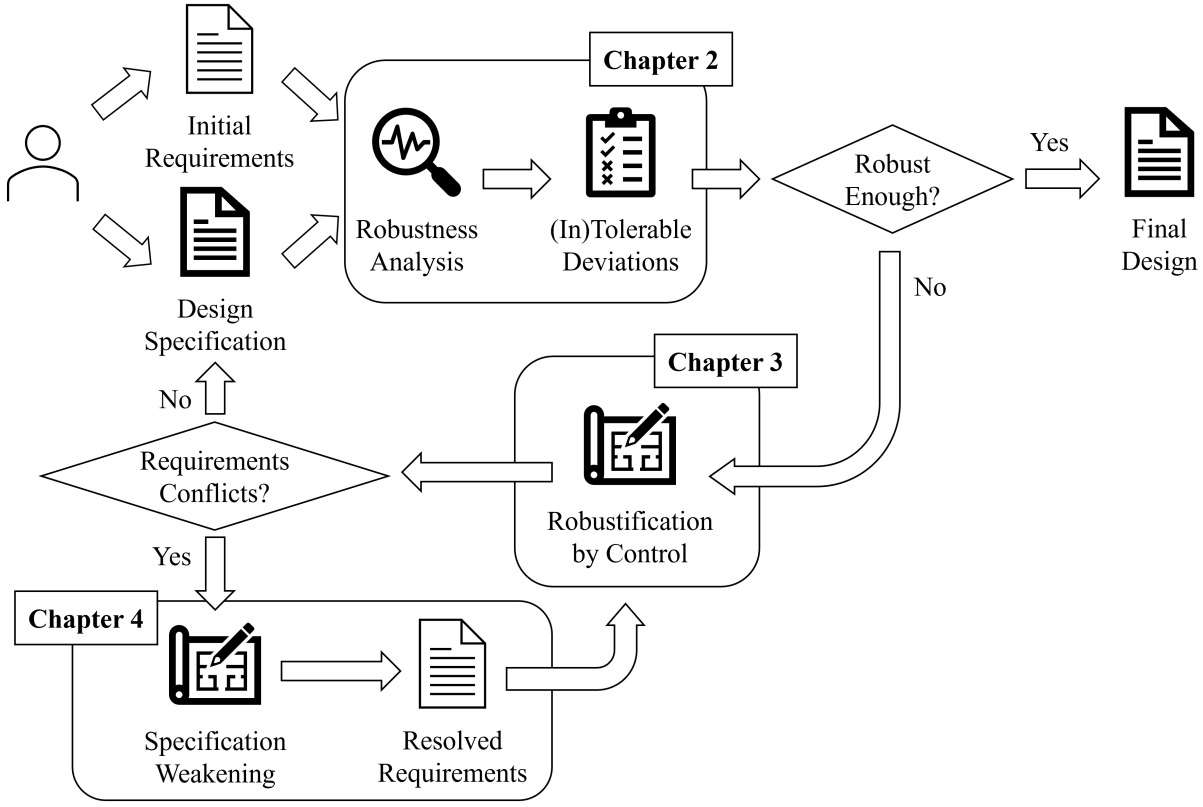


Figure 1.3: Overview of robustness analysis and design robustification process.

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 that are robust against a deviated environment across software applications from different domains. Lastly, the efficiency of our robustness computation and robustification methods is demonstrated through experiments on a set of benchmark problems derived from our five case studies.

1.5.4 Summary of Contributions

Figure 1.3 shows an overview of our proposed robustness analysis and design robustification process. At the beginning of this process, developers provide the initial machine design and system requirements (including safety requirements, functional requirements, controllability and observability, and cost information) to our analysis tool. Our tool can analyze the robustness of the machine design with respect to the safety specification and inform the developers about the deviations that the system can or cannot tolerate (i.e., is or is not robust against). If the developers are not satisfied with the design, they can use the robustification-by-control method to robustify the design, which fixes the safety

requirement and finds a redesign that balances the trade-offs between functionality, controllability and observability, and implementation costs. However, if there are conflicts between the safety requirements and functionality that cannot be addressed by enhancing controllability and observability, the developers can choose to weaken the safety requirement and redo the robustification-by-control process to find another redesign. This process can iterate multiple times until a satisfactory design is generated.

The list of contributions of this thesis include:

- **Behavioral Notion of Robustness**: We present 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 by Control**: We present a method to improve the robustness of a machine design with respect to a safety property by controlling its actions. The method also considers the goal of retaining as much functionality as possible and minimizing the cost of changes.
- **Robustification by Specification Weakening**: When a robustification-by-control process cannot find satisfactory redesigns of a machine because the safety property is excessively strong, we present a method to weaken the safety property to allow more feasible solutions through robustification-by-control.
- **Implementation and Evaluation**: We implement the proposed robustness computation and robustification approaches in an open-source tool named Fortis, evaluating their applicability and efficiency through five case studies.

Chapter 2

A Behavioral Notion of Robustness

2.1 Introduction

In this chapter, we present our approach for systematic robustness analysis based on a mathematically rigorous notion of behavioral robustness for software. In particular, we say that a software machine is *robust* with respect to a *property* and a particular set of *environmental deviations* if the machine continues to satisfy the property even when the environment exhibits those deviations. Furthermore, we define the *behavioral robustness* of a software machine as the set of all deviations under which the machine continues to satisfy the property. Based on these definitions, we then present an analysis technique for automatically computing the behavioral robustness of a machine.

The goal of a typical verification method is to check the following: Given machine M, environment E, and property P, does the machine satisfy the property under the environment (i.e., $M||E \models P$)? Our notion of robustness enables the formulation of new types of analyses beyond this. For instance, we could ask whether a machine is robust against a particular set of environmental deviations; given two alternative machine designs (both satisfying property P), we could rigorously compare them by generating deviations against which one design is robust but the other is not; and given multiple system properties (some of them more critical than others), we could compare the environmental deviations under which the machine can guarantee them.

We demonstrate the application of our approach through five case studies. In particular, we focus on two of them, featuring two application domains: (1) human-machine interactions, where we adopt well-studied models of human errors from industrial engineering and human factors research [44, 45] and show how our method can be used to rigorously evaluate the robustness of safety-critical interfaces against such errors, and (2) computer networks, where our method is used to rigorously compare the robustness of network protocols against different types of failures in the underlying network. Furthermore, we build a set of benchmark problems from these case studies and evaluate the performance of our approach.

The rest of this chapter is organized as follows:

- Section 2.4 defines our formal notion of robustness for software machines and a set

of analysis problems that evaluate machine designs with respect to their robustness;

- Section 2.5 presents algorithmic techniques for automatically computing the robustness of a machine and generating succinct representations of robustness;
- Section 2.6 presents the implementation of our robustness analysis and the evaluation on five case studies.

2.2 Motivating Example

This section illustrates how our proposed notion of robustness may be used to support a new type of design analysis and aid in the systematic development of software machines that are robust against failures or changes in the environment.

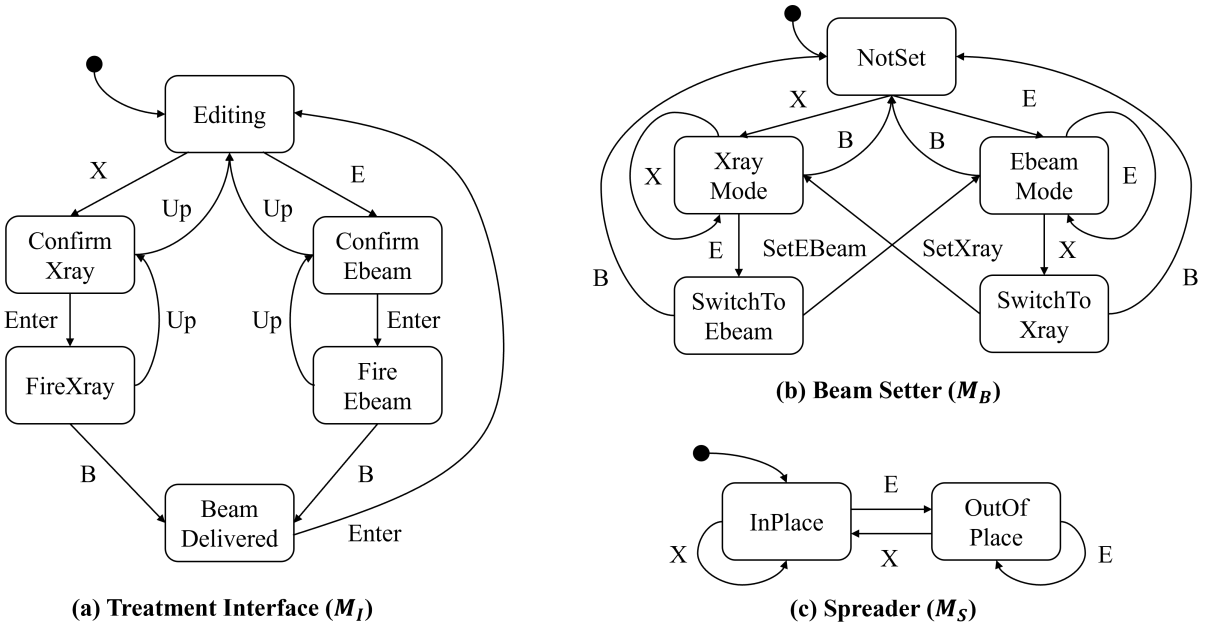


Figure 2.1: Labeled transition systems for a radiation therapy system.

Analysis under the normative environment. As a motivating example, consider the design of a radiation therapy machine similar to the well-known Therac-25 machine [46]. The state machines in Figure 2.1 describe the three components of the machine[1], including (a) the *Treatment Interface* (M_I), which allows an operator to choose the radiation mode (Electron or X-ray) and fire the beam; (b) the *Beam Setter* (M_B), which switches the physical component for the two radiation modes; and (c) the *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

[1]For simplicity of illustration, some states and events are omitted from the diagrams.

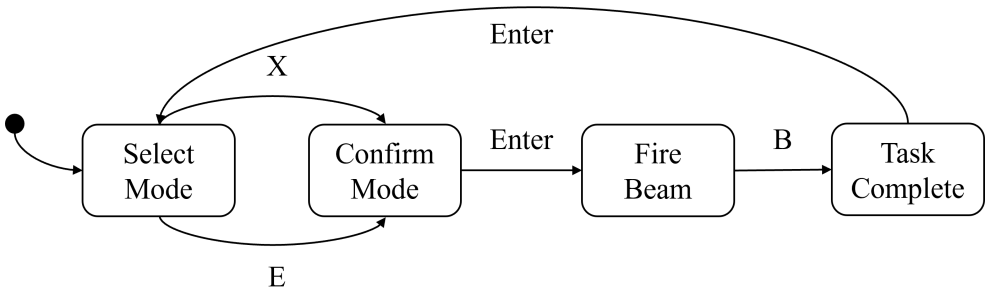


Figure 2.2: Labeled transition system for the operator task model (E).

the Electron beam). The overall behavior of the machine is the composition of the three components, i.e., $M = M_I||M_B||M_S$.

The radiation therapy machine is associated with a number of safety requirements, one of which states: *The spreader must be in place when the beam is delivered in the X-ray mode.* Otherwise, the high power of the X-ray will lead to an overdose and cause fatal injuries to patients [46].

During the normal treatment process, a therapist is expected to perform the following tasks: Select the correct therapy mode for the current patient by pressing either X or E, confirm the treatment data by pressing Enter, and then finally initiate the beam delivery to the patient by pressing B. Figure 2.2 shows the state machine for this normative operator behavior.

Suppose the designer of the machine wishes to check whether the therapy machine satisfies its safety requirements, assuming that an operator carries out the tasks as expected. More generally, this can be formulated as the following common type of analysis task:

> *Does the machine, under an environment that behaves as expected, satisfy a desired property?*

To perform this task, one may apply a verification technique such as model checking [47] to check whether the composition of the machine and the environment satisfies the desired property. Performing this analysis confirms that the machine indeed satisfies the safety property that the spreader is always in place during the X-ray mode.

Analysis of undesirable environmental deviations. In complex systems, the environment may not always behave as expected and may possibly undermine the assumptions that the machine relies on to fulfill its requirements. For instance, in interactive systems, human operators are far from perfect and inadvertently make mistakes from time to time while performing a task (e.g., performing a sequence of actions in the wrong order) [45]. In the context of a safety-critical system such as medical devices, some of these operator errors, if permitted by the interface, may result in a safety violation.

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

What are possible ways in which the environment may deviate from its expected behavior and cause a violation of the property?

Given the therapy machine model M and property P, the designer can use an existing analysis tool (e.g., a model-checking tool like LTSA [48]) to check whether $M \models P$. The analyzer may return a counterexample trace that demonstrates how the operator could deviate from their normative behavior (as captured by E) and cause a violation of P.

Suppose that one such trace contains the following sequence of operator actions: ⟨X, Up, E, Enter, B⟩. This trace depicts a scenario in which the operator accidentally selects the X-ray mode, corrects the mistake by pressing Up and selecting the Electron beam mode, and then carries on with the rest of the treatment as intended (by confirming the mode and firing the beam). This sequence of operator actions, however, may lead to a violation of the safety property P in the following way: When the operator presses B, the beam setter may still be in the process of mode switching (i.e., in state SwitchToBeam), causing the beam to be delivered in the X-ray mode while the spreader is out of place. This scenario corresponds to one type of failure that caused fatal overdoses in the Therac-25 system [46].

Robustness analysis. Having discovered how the operator's mistake could lead to a safety violation, the designer (manually) modifies the treatment interface to *improve* its robustness against the possible error. In this redesign, shown in Figure 2.3, the operator can press B to fire the beam only after the mode switching has been carried out by the beam setter. As the next step, the designer wishes to ensure that the machine, as redesigned, is robust against the operator's mistake (i.e., it continues to satisfy the safety property even under the misbehaving operator).

The designer could check $M' \models P$, where M' is the redesign. If no errors are returned, it means that M' is robust against the mistake and that M' can work under *any* environment. However, this is not always the case. More likely, the analyzer may return another trace representing a new mistake, and it does not necessarily mean that the machine is robust against the previous one.

Instead, the designer can use our approach to perform the following *robustness analysis* task:

What are possible environmental deviations under which the new design satisfies the property, but the old design does not?

Given the original machine model M, modified model M', normative environment E, and property P, our analysis returns a set of traces (expressed over environmental events). Each trace describes a scenario where machine M' satisfies the property, but M does not. For example, one of the traces is the sequence of operator actions discussed above: ⟨X, Up, E, Enter, B⟩, confirming that the redesign has correctly addressed the risk of a possible safety violation due to this particular type of operator mistake.

The steps of analyzing undesirable environmental deviations, improving robustness, and analyzing robustness can be repeated to identify more potential safety violations due to other types of operator mistakes, further enhancing the machine's robustness. We call this

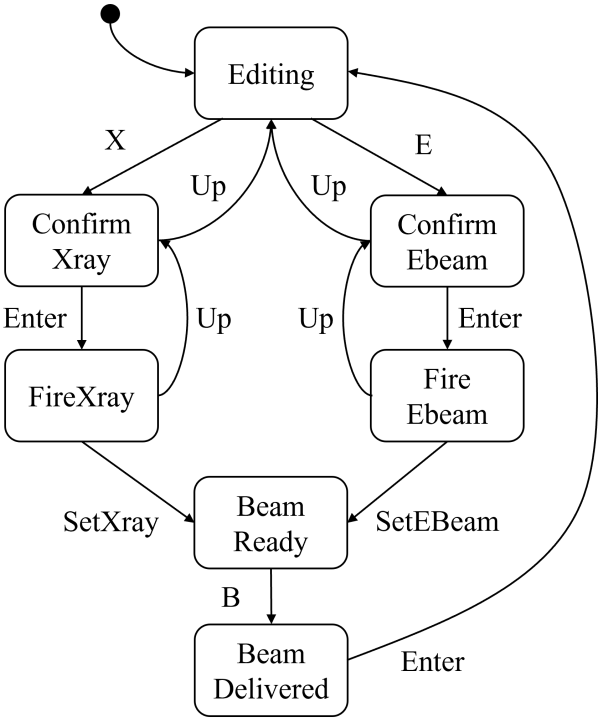


Figure 2.3: A redesign of the radiation therapy machine. In particular, we show only the redesigned interface where the operator can fire the beam until the mode switching has completed. For simplicity of illustration, some states and transitions are omitted.

iterative process a *robust-by-design* loop. This chapter presents our approach for analyzing robustness.

2.3 Preliminaries

In this work, we use labeled transition systems to model the behavior of machines and environment and define properties.

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. An LTS is *complete* if $\forall s : S, a : \alpha T : \exists s' \in S : (s, a, s') \in R$.

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. Moreover, the set $beh(T)$ is *prefix-closed* such that for any trace $\sigma \in beh(T)$ and any prefix u of σ, u is also in $beh(T)$.

Operators. For LTS $T = \langle S, \alpha T, R, s_0 \rangle$, the *projection* operator $\uparrow$ exposes a subset of the alphabet of T. Given $T{\uparrow}A = \langle S, \alpha T \cap A, R', s_0 \rangle$, for any $(s, a, s') \in R$, if $a \notin A$, then $(s, \tau, s') \in R'$; otherwise, $(s, a, s') \in R'$. The $\uparrow$ operator also applies to traces; $\sigma{\uparrow}A$ 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 [49]. 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* [9], 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 $beh(T{\uparrow}\alpha P) \subseteq beh(P)$.

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 $R_{err} = R \cup \{(s, a, \pi) | a \in \alpha P \wedge \nexists s' \in S : (s, a, s') \in R\}$. Thus, P_{err} is a complete LTS. 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 $T \models P$.

2.4 Robustness Analysis

2.4.1 Robustness Definition

Let M be the LTS of a machine, E be the LTS of the environment, and $\alpha E_M = \alpha M \cap \alpha E$ be the common actions between the machine and the environment. Then, we say $beh(M{\uparrow}\alpha E_M)$ represents the set of all environmental behaviors that are *permitted* by the machine M. Machine M is said to be *robust* against a set of traces $\delta \subseteq {\alpha E_M}^*$ if and only if the machine satisfies a desired property under a new environment E' that is capable of additional behaviors in δ compared to the original environment E. Formally, we have:

Definition 2.1. *Machine M is robust against a set of traces $\delta \subseteq {\alpha E_M}^*$ with respect to the environment E and property P if and only if $M||E \models P$, $\delta \cap beh(E{\uparrow}\alpha E_M) = \emptyset$, and for every E' such that $beh(E'{\uparrow}\alpha E_M) = beh(E{\uparrow}\alpha E_M) \cup \delta$, $M||E' \models P$.*

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

Definition 2.2. *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'.*

In addition, although this definition is general enough for both safety and liveness properties, this work particularly focuses on safety properties.

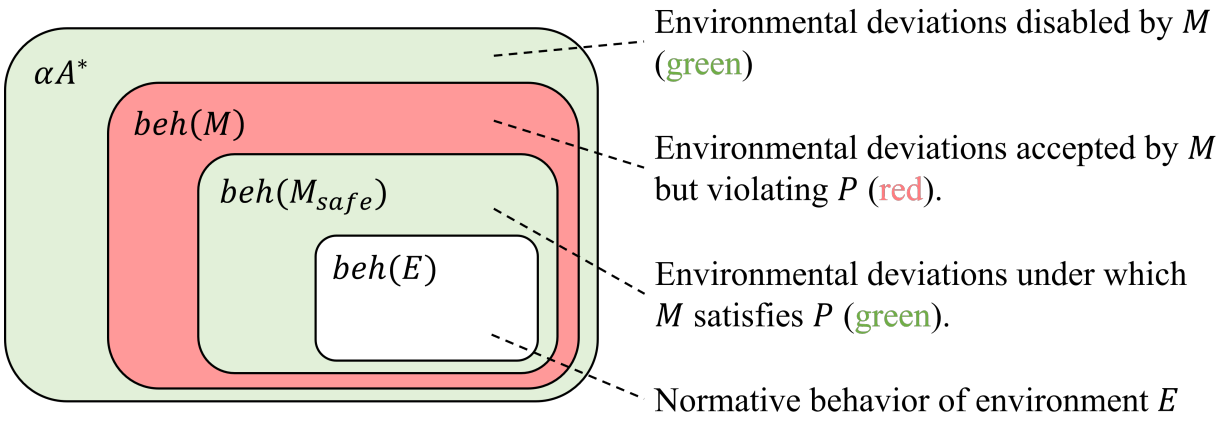


Figure 2.4: Illustration of behavioral relationships between machine M, environment E, and robustness $\Delta(M, E, P)$.

Figure 2.4 illustrates the relationships between the behaviors of the machine, the environment, and robustness. For simplicity, we assume that all behaviors share the same alphabet αA. The outermost box represents the set of all possible finite traces over αA. Within this, $beh(M)$ represents all behaviors permitted by the machine, and $beh(E)$ represents the normative environment's behaviors. For ease of illustration, we assume that the normative environment's behaviors are a subset of the machine's behaviors, i.e., $beh(E) \subseteq beh(M)$.

Given a safety property P, the behaviors of the machine M can be classified into two sets: $beh(M_{safe})$, the set of *all* behaviors that are permitted by machine M and under which the machine satisfies P, and the rest of the unsafe behaviors (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 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.

Example. In the radiation therapy machine, trace $\langle$X, Enter, Up, Enter, B$\rangle$ is a deviation that is accepted 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 deviation disabled by the machine and 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 $\Delta(M, E, P)$.

[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.

2.4.2 Analysis Problems

Given our robustness definition, we can answer the following analysis problems with respect to robustness. First of all, we can compute the robustness as defined in 2.2:

Problem 2.1. *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 2.2. *Given machines M_1 and M_2, an environment E, and a safety property P such that $\alpha M_1 \cap \alpha E = \alpha M_2 \cap \alpha E$, compute set $\mathcal{X} = \Delta(M_2, E, P) - \Delta(M_1, E, P)$.*

This analysis allows us to compare a pair of machine 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 that being robust against certain deviations may lead to violations under other deviations, which reflects design trade-offs.

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

Problem 2.3. *Give a machine M, an environment E, and safety properties P_1 and P_2, compute set $\mathcal{X} = \Delta(M, E, P_2) - \Delta(M, E, P_1)$.*

For instance, suppose that P_1 is a stronger safety property stating that "the radiation machine should always deliver the correct amount of dose to each patient", while P_2 is weaker stating that "the machine never overdoses patients by delivering X-ray while the spreader is out of place". The result of this analysis can tell us, for example, that the machine is capable of guaranteeing P_2 (weaker but arguably more critical of the two) even under certain operator mistakes, while P_1 may be violated under similar deviations. It indicates the design trade-offs between safety and robustness and may be useful in the context of requirements weakening [50, 38, 51, 52]. In other words, since, in general, improving robustness might introduce additional complexity to a machine, it may be more cost-effective to design a machine to be robust against the most critical requirements [53].

2.5 Robustness Computation

2.5.1 Overview

Figure 2.5 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 is then used to compute the model of the robustness Δ. In general, Δ may be infinite and not in a form that can be easily comprehended 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 deviations, 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

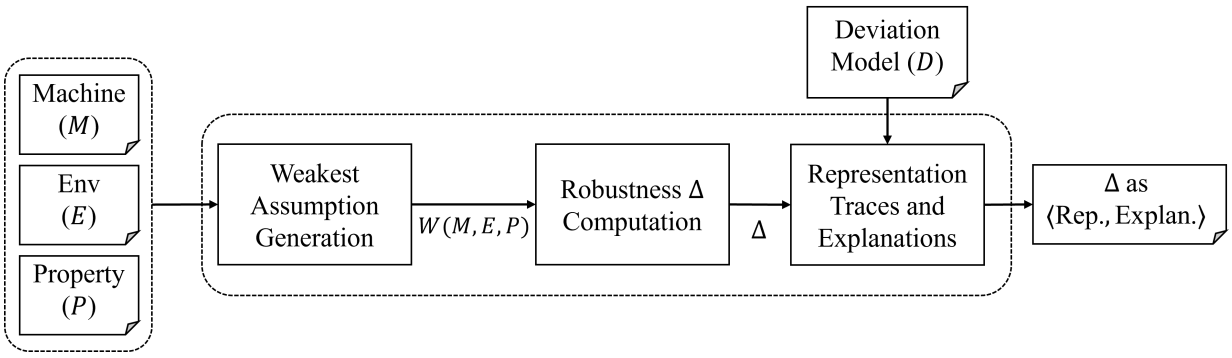


Figure 2.5: Overview of the robustness computation process.

way. The final output of the process is a set of pairs of a representative trace and its explanation.

2.5.2 Weakest Assumption

In assume-guarantee style of reasoning [54, 55], a machine is considered capable of establishing a property under some *assumption* about the behavior of the environment. The *weakest assumption* is the largest set of possible environmental behaviors under which the machine satisfies the property. More formally:

Definition 2.3. *The weakest assumption $W(M, E, P) \subseteq \alpha E_M{}^*$ of a machine M with respect to environment E and property P defines the largest set of environmental behaviors under which M satisfies P, i.e.,*

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

If stated otherwise, we will simply use W to mean $W(M, E, P)$. Therefore, given this definition, the weakest assumption W should include all behaviors in $beh(M_{safe})$ plus the behaviors in $\delta_2 = \alpha A^* \setminus beh(M)$, according to the illustration in Figure 2.4. Then, the robustness of a machine can be computed by its weakest assumption *minus* the behaviors of the normative environment:

$$\begin{aligned} \Delta(M, E, P) &= beh(W) \setminus beh(E{\restriction}\alpha E_M) \\ &= \{\sigma \in beh(W) \mid \sigma \notin beh(E{\restriction}\alpha E_M)\} \end{aligned} \tag{2.1}$$

We use the approach by Giannakopoulou et al. [56] to generate the weakest assumption given that P is a safety property. We briefly summarize their approach: Given the LTS of machine M, environment E, and safety property P,

1. Compose machine M with the *error* LTS of the safety property P, i.e., $M||P_{err}$.
2. Hide the internal events of M with respect to E from $M||P_{err}$, and propagate the error state over τ transitions (unobservable transitions). That is, if a state can reach the error state via one or more consecutive τ transitions, then that state should also

be an error state. The rationale is that the environment cannot prevent the machine from violating the property when the machine is in a state that can reach the error state with some internal actions.

3. Determinize the resulting LTS from step 2 by applying τ elimination and subset construction [57].
4. Add a sink state θ to make the LTS obtained from step 3 *complete*, i.e., all events should be enabled on every state. This is achieved by adding a transition to θ if an event is disabled on a state. Finally, remove the error state and all of its incoming transitions to obtain the LTS representing the weakest assumption.

From a high level, any trace that eventually reaches the error state is a trace that is accepted by the machine and violates the safety property, which is eventually removed from the model. The introduction of the sink state adds the traces that are not accepted by the machine. Therefore, this process computes the weakest assumption of machine M with respect to environment E and safety property P.

2.5.3 Representation of Robustness

In general, the set of environmental traces that represent robustness in Equation 2.1 may be infinite. Since simply enumerating this set may not be an effective way to present this information to system designers, we propose a succinct, *finite* representation of robustness. The key idea behind our approach is that many of the traces in $\Delta(M, E, P)$ capture a similar type of deviation (e.g., a human operator erroneously skipping an action) and can be grouped into the same equivalence class with a single *representative trace* that describes the deviations. Based on this idea, we describe a method for automatically converting Δ into a finite number of such equivalence classes, and thus, a finite set of representative traces.

Representative model of robustness. Recall from Equation 2.1 that Δ contains traces that are in the weakest assumption W but not in the original environment E. To construct an LTS that represents Δ, we take advantage of the method to check safety properties in LTS (described in Section 2.3). In particular, we treat the original environment E projected over $\alpha {E_M}^*$ as a safety property and compute the traces in W that lead to a violation of this property; any such trace represents a *prefix* of the traces in $\Delta(M, E, P)$.

To illustrate our approach, consider a simple example in Figure 2.6, where W is the weakest assumption generated from some machine M and E is the original environment. To compute the representation of $\Delta(M, E, P)$, we first test whether $W \models (E \uparrow \alpha E_M)$, which is equivalent to testing whether the error state is reachable in $W || (E \uparrow \alpha E_M)_{err}$, as shown in Figure 2.6(c). We say $W || (E \uparrow \alpha E_M)_{err}$ is the *representative model* of $\Delta(M, E, P)$, and let $\Pi(W, E)$ be the set of all error traces in it. Then, we have:

$$\Delta(M, E, P) = \{\sigma \in beh(W) \mid \exists \sigma' \in \Pi(W, E) : \mathsf{prefix}(\sigma', \sigma)\} \tag{2.2}$$

where $\mathsf{prefix}(\sigma_1, \sigma_2)$ means σ_1 is a prefix of σ_2. Thus, a trace in $\Pi(W, E)$ can represent a set of traces in $\Delta(M, E, P)$ that share this prefix. For this example, trace $\langle a, c \rangle$ in $\Pi(W, E)$

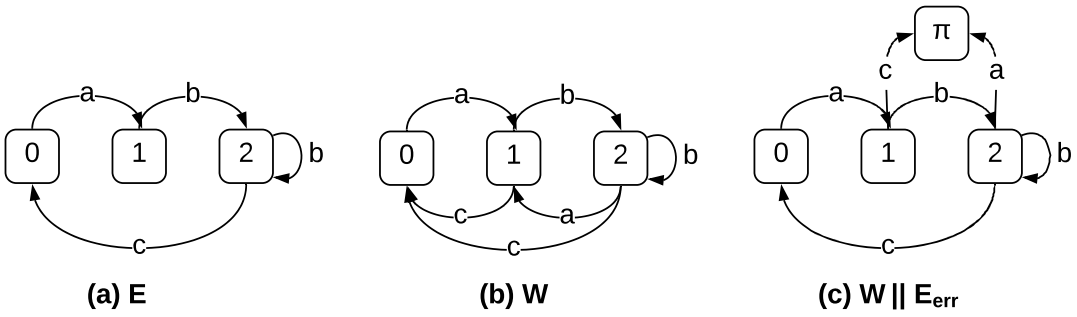


Figure 2.6: LTSs for a simple example illustrating the construction of robustness, where state 0 is the initial state.

can represent, e.g., $\langle a, c, a, b, \ldots\rangle$ and $\langle a, c, a, c, \ldots\rangle$ in $\Delta(M, E, P)$.

Representative traces of robustness. Nevertheless, $\Pi(W, E)$ may also be infinite due to possible cycles in the representative model. For instance, in Figure 2.6(c), $\langle a, b, b, \ldots, a\rangle$ would result in an infinite number of error traces. Therefore, we further divide the traces into equivalence classes:

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

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

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

$$\Delta(M, E, P) = \{\sigma \in beh(W) \mid \exists \Pi_{s,a}(W, E), \sigma' \in \Pi_{s,a}(W, E) : \mathsf{prefix}(\sigma', \sigma)\} \tag{2.3}$$

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

The rationale is that state s is the last state when following the normative behavior of the original environment, and event a is the first deviated event. Thus, $\Pi_{s,a}(W, E)$ describes a class of traces that deviate from the original environment starting from the same normative state s and by the same event a. Since $S_{W,E}$ and αE_M are finite, we have a finite number of equivalence classes. We can generate them by enumerating all the transitions leading to the error state. Then, we pick one of the traces in each equivalence class to represent $\Delta(M, E, P)$. Because we may not be interested in how the environment reaches the last normative state, we choose the shortest trace in each equivalence class. Finally, we have:

Definition 2.4. *The representation of $\Delta(M, E, P)$, denoted by $\Delta_{rep}(M, E, P)$, is a finite set of traces such that each trace in it is the shortest trace of one of the equivalence classes of $\Pi(W, E)$.*

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

2.5.4 Explanation of Robustness

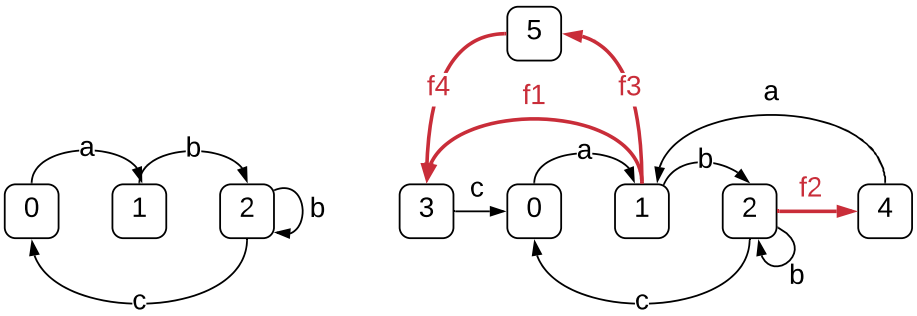
By definition, a representative trace in $\Delta_{rep}(M, E, P)$ contains only actions from αE_M. While this trace describes how the environment deviates from its expected behavior as *observed* by the machine, it does not capture how the internal behavior of the environment could have caused this deviation. To provide such an *explanation* for an environmental deviation, we propose a method for augmenting the representative traces with additional domain-specific information (called *faulty events*) about the underlying root cause of the deviation. In this approach, the normative environment model is augmented with additional transitions for these faulty events (which are internal to the environment), and an automated method is used to extract a *minimal explanation* for a particular representative trace.

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

Definition 2.5. *A deviation model D of environment E is an LTS $T = \langle S, \alpha D, R, s_0 \rangle$ where $\alpha D = \alpha E \cup \{f_1, f_2, \ldots, f_n\}$, f_i is a fault in the environment, $beh(E) \subseteq beh(D{\restriction}\alpha E)$, and $beh(D{\restriction}\alpha E_M) \cap \Delta(M, E, P) \neq \emptyset$.*

Our approach makes no assumptions on how to generate such a deviation model. It can be built manually or derived from existing fault models in other fields (e.g., an existing human error behavior model). The model may not necessarily cover all the traces in $\Delta(M, E, P)$; however, we say a deviation model is *complete* with respect to $\Delta(M, E, P)$ if and only if $\Delta(M, E, P) \subseteq beh(D{\restriction}\alpha E_M)$.

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



(a) Original Environment *E* **(b) Deviation Model *D***

Figure 2.7: Deviation model for the simple example, where state 0 is the initial state.

Definition 2.6. *For any trace $\sigma \in \Delta_{rep}(M, E, P)$ and $\sigma' \in beh(D)$, if $\sigma' \upharpoonright \alpha E_M = \sigma$, then we say σ' is an explanation of σ.*

Consider a deviation model for our simple example in Figure 2.7. Then, we have: for the representative trace $\langle a, c\rangle$, we can build explanations $\langle a, f_1, c\rangle$ and $\langle a, f_3, f_4, c\rangle$; and for the representative trace $\langle a, b, a\rangle$, we can build an explanation $\langle a, b, f_2, a\rangle$.

The minimal explanation. In general, there could be an infinite number of explanations for a representative trace. However, similar to software testing, where we are often interested in the smallest test cases against certain errors, here we are also interested in the explanation of σ that contains the minimal number of faults.

Definition 2.7. *The* minimal explanation *for $\sigma = \langle a_0, \ldots, a_{n-1}, a_n\rangle$ in $\Delta_{rep}(M, E, P)$ under deviation model D is the shortest trace $\sigma' \in beh(D)$ where $\sigma' \upharpoonright \alpha E_M = \sigma$ and faulty events only exist between a_{n-1} and a_n.*

A *minimal explanation* describes: 1) how the environment can reach the last normative state without any faults; and 2) what minimal sequence of faults has caused the environment to deviate from the normative behavior.

To compute the minimal explanation for $\sigma \in \Delta_{rep}(M, E, P)$, let $T_\sigma = \langle S, \alpha E_M, R, s_0\rangle$ be the LTS where σ and its prefixes are the only traces in it. Additionally, we make the last event in σ lead to π to denote the end state, i.e., $(s, a_n, \pi) \in R$. Then, we use a *breadth-first search* (BFS) to find the minimal explanation in $D||T_\sigma$, as shown in Algorithm 1.

Lines 1 to 3 define an empty queue to store the remaining search states and an empty set to store the visited states, and add the initial state to the queue. The algorithm loops until the queue is empty (line 4). If the current visiting state is π, then it returns the current trace as the explanation (lines 7 to 8); otherwise, it adds the next states to the queue. Specifically, if the current trace does not match the prefix of σ, i.e., $\langle a_0, \ldots, a_{n-1}\rangle$, then it only adds states with non-faulty transitions (lines 12 to 16). Since BFS returns on the first result, it is guaranteed to find the minimal explanation. For example, our algorithm returns $\langle a, f_1, c\rangle$ as the minimal explanation for $\langle a, c\rangle$ instead of $\langle a, f_3, f_4, c\rangle$ in the deviation model in Figure 2.7(b).

2.5.5 Robustness Comparison

We then show how to compare robustness between a pair of machine designs (Problem 2.2) or a machine against a pair of properties (Problem 2.3). According to Equation 2.1, to solve Problem 2.2, we have:

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

According to Definition 2.1, $M||E \models P$, thus we have $beh(E\upharpoonright\alpha E_M) \subseteq beh(W_{M_1})$. Then, we can simplify this equation to:

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

Algorithm 1: Minimal explanation search

Data: A trace $\sigma \in \Delta_{rep}(M, E, P)$ and the LTS of $D||T_\sigma$
Result: The minimal explanation $\sigma' \in beh(D)$ for σ

```
1  q := empty queue ;                          // remaining search states
2  v := empty set of states ;                  // visited states
3  enqueue(q, (s_0, ⟨⟩)) ;
4  while ¬isEmpty(q) do
5  |  s, t := dequeue(q);      // s the current state, t the current
   |   trace
6  |  if s ∉ v then
7  |  |  if s = π then
8  |  |  |  return t;
9  |  |  else
10 |  |  |  v := v ∪ {s};
11 |  |  |  for (s, a, s') ∈ R do
12 |  |  |  |  if t ↾ αE_M = subTrace(σ, 0, n − 1) then
13 |  |  |  |  |  enqueue(q, (s', t ⌢ a))
14 |  |  |  |  else if a is not a fault then
15 |  |  |  |  |  enqueue(q, (s', t ⌢ a))
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Then, we can use the same method described in Section 2.5.3 to generate its representation. By computing $W_{M_2}||(W_{M_1})_{err}$, we obtain $\Pi(W_{M_2}, W_{M_1})$, which represents all the prefixes of $\mathcal{X}$. Similarly, we divide it into equivalence classes, i.e., $\Pi_{s,a}(W_{M_2}, W_{M_1})$, where (s, a) leads to the error state. Then, we have:

$$\begin{aligned}\mathcal{X} &= \Delta(M_2, E, P) - \Delta(M_1, E, P) \\ &= \{\sigma \in beh(W_{M_2}) \mid \exists \Pi_{s,a}(W_{M_2}, W_{M_1}), \sigma' \in \Pi_{s,a}(W_{M_2}, W_{M_1}) : \mathsf{prefix}(\sigma', \sigma)\}\end{aligned} \tag{2.4}$$

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

We apply the same process to $\mathcal{X} = \Delta(M, E, P_2) - \Delta(M, E, P_1)$. With $beh(E{\restriction}\alpha E_M) \subseteq beh(W_{P_1})$ and by computing $\Pi(W_{P_2}, W_{P_1})$ and its equivalence classes, we have:

$$\begin{aligned}\mathcal{X} &= \Delta(M, E, P_2) - \Delta(M, E, P_1) \\ &= \{\sigma \in beh(W_{P_2}) \mid \exists \Pi_{s,a}(W_{P_2}, W_{P_1}), \sigma' \in \Pi_{s,a}(W_{P_2}, W_{P_1}) : \mathsf{prefix}(\sigma', \sigma)\}\end{aligned} \tag{2.5}$$

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

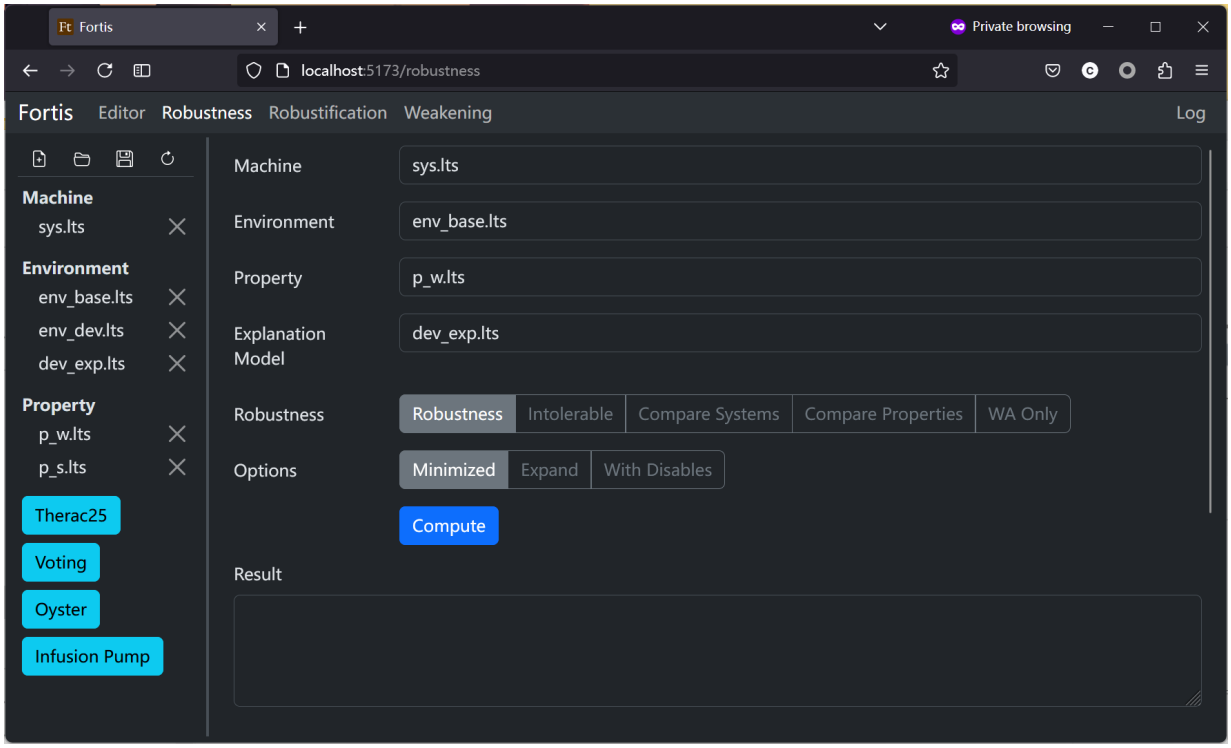


Figure 2.8: Screenshot of Fortis for robustness analysis.

2.6 Evaluation

2.6.1 Research Questions

Our evaluation focus on two research questions:

- **RQ1 (Applicability)**: Is our robustness analysis technique applicable to software systems from different domains? Particularly, do the deviations generated from our analysis correspond to real-world erroneous scenarios that have been previously investigated in other domains?
- **RQ2 (Scalability)**: How well does our robustness computation method scale?

To answer these research questions, we evaluated our approach on five case studies, including the radiation therapy machine (described in Section 2.2), an electronic voting machine, network protocols, a medical infusion pump, and a public transportation fare system. Specifically, to answer RQ1, we focused on the network protocol example, which features the robustness of distributed systems against different types of network failures, and the radiation therapy example, which features the robustness of safety-critical interfaces against human errors. To answer RQ2, we built a set of benchmark problems from the five case studies. Then, we evaluated the scalability of our robustness computation approach by measuring the time to solve all these benchmark problems.

2.6.2 Implementation

We implemented our robustness analysis techniques in a tool named Fortis [43]. It is implemented as a Kotlin program (a JVM-based language) with an optional web-based user interface developed in Vue.js[3]. It leverages Automatalib [58] and LTSA [48] for model specification and manipulation. It supports commonly used specification languages such as AUT and FSM (through Automatalib) and FSP (the language used by LTSA) to specify and output system models. The source code of the implementation is available on GitHub at `https://github.com/cmu-soda/fortis-core`.

Figure 2.8 shows the web interface of Fortis for robustness analysis. On the left, a user can manage all their model specifications. There is also an Editor tab with basic specification editing support. On the right panel, the user can input the models for the machine, environment, and safety property, respectively. The user can also provide an optional deviation model for explanations. Finally, by clicking the Compute button, the Fortis back-end is invoked to compute the robustness.

Fortis also provides a command-line interface. A user can interact with Fortis through command-line arguments or a JSON configuration file, and the tool produces results in command-line outputs. We leveraged the command-line interface to conduct our evaluation. All experiments were conducted on a Windows machine with an Intel i9-12900H processor and 32GB memory.

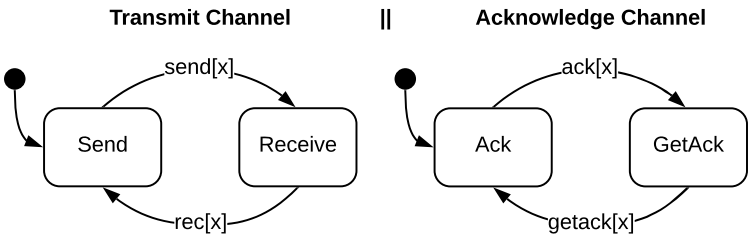
2.6.3 Network Protocol Design

This section describes a case study on rigorously evaluating the robustness of network protocol designs. In particular, we focus on two protocols: a naive protocol that assumes a perfectly reliable communication channel, and the Alternate Bit Protocol (ABP) [59], which is specifically designed to guarantee the integrity of messages over a potentially unreliable communication channel. By computing and comparing the robustness of the two, we formally show that the ABP is indeed more robust than the naive protocol.

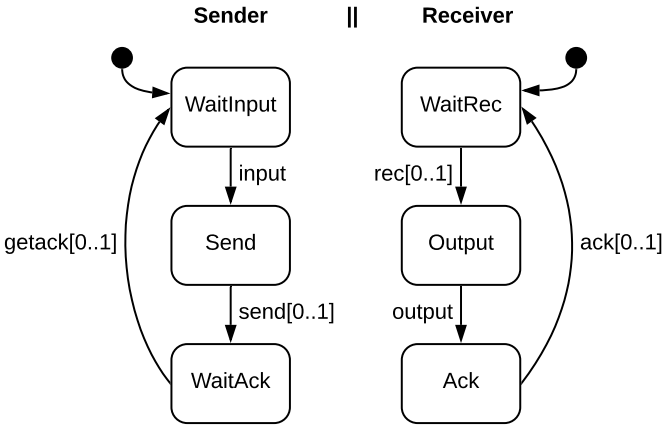
Figure 2.9a shows the LTS of the environment. Here, the environment E corresponds to a communication channel over which messages are transmitted (with $\alpha E = \{send[0..1], rec[0..1], ack[0..1], getack[0..1]\}$, where $send[0..1]$ represents a set of actions $\{send[0], send[1]\}$). Under normal circumstances, we expect that the channel reliably delivers messages to the intended receiver (i.e., it does not lose, duplicate, or corrupt messages). Figure 2.9b shows the LTS of the naive protocol M_N: The sender sends user input data with either 0 or 1, and waits on the acknowledgment; the receiver waits for messages, output the message data, and acknowledges to the sender with either 0 or 1.

Figure 2.10 shows the model of ABP, which is adopted from the model used in [1]. The sender first sends a message with 0, and it continues sending the message until it receives an acknowledgment with 0. Then, it alternates the bit to send a message with 1. The receiver first waits for a message with 0, and it continues sending acknowledgments with 0 until it receives a new message with 1. Then, it acknowledges with 1 and waits for a new message with 0.

[3] `https://vuejs.org/`



(a) A perfect network channel E.



(b) A naive communication protocol M_N.

Figure 2.9: Models of the perfect network channel and the naive protocol.

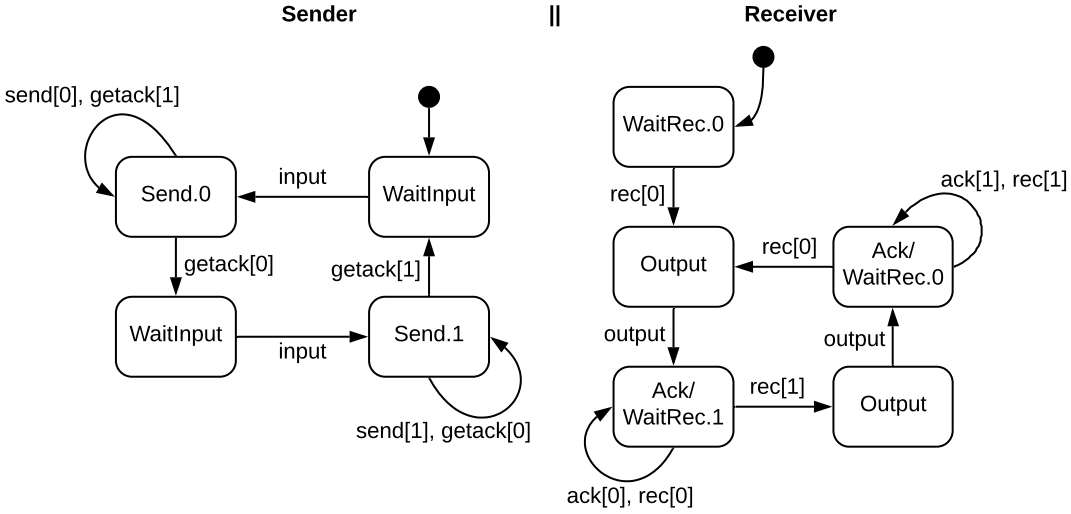


Figure 2.10: Model of the Alternate Bit Protocol (M_{ABP}) adopted from [1].

Computing robustness and explanations. We define safety property P as "the input and output should alternate". The LTS of this property can be specified in the FSP language as:

```
property P = (input -> output -> P).
```

This property ensures that the sender sends a new message only after it receives the receiver's acknowledgment that the previously sent message was successfully delivered. We used our tool to compute the robustness of the two protocols, i.e., $\Delta(M_N, E, P)$ and $\Delta(M_{ABP}, E, P)$. Then, we generated their corresponding representations, i.e., $\Delta_{rep}(M_N, E, P)$, which contains 4 representative traces from 4 equivalence classes, and $\Delta_{rep}(M_{ABP}, E, P)$, which contains 107 traces corresponding to 107 equivalence classes. Finally, we built a deviation model D, which contains message loss, duplication, and corruption of bits (only the bit parameter 0 and 1, but not the message content) to provide explanations for these representative traces. Figure 2.11 shows its specification.

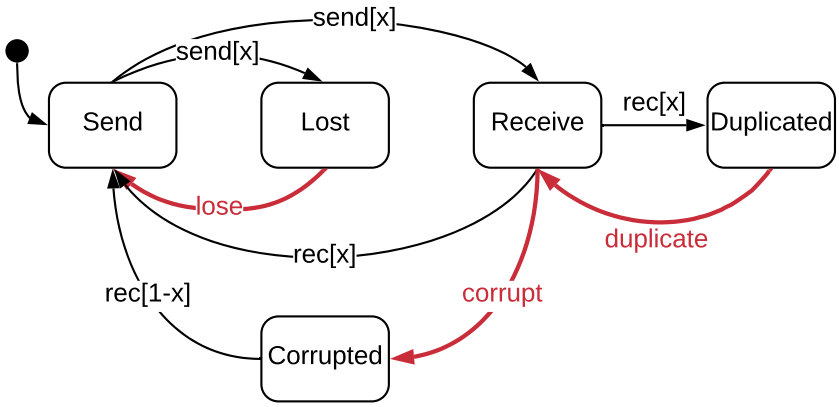


Figure 2.11: Deviation model that describes the faulty transmission channel. The faulty acknowledge channel is similarly structured and omitted here.

Analysis. All 4 traces in $\Delta_{rep}(M_N, E, P)$ correspond to bit corruption errors. For example, the explanation for $\langle send[0], rec[1]\rangle$ is $\langle input, send[0], corrupt, rec[1]\rangle$. We were surprised to find that the naive protocol is robust against such errors (our expectation was that the naive protocol would not be robust against any kind of environmental deviations at all). This is because property P is under-specified in the sense that: It requires only that the input and output actions alternate and does not say anything about the bit parameters in the sent and corresponding received messages.

For the 107 traces in $\Delta_{rep}(M_{ABP}, E, P)$, our tool finds the minimal explanations for 99 of them. For example, the explanation for $\langle send[0], send[0]\rangle$ is $\langle input, send[0], lose, send[0]\rangle$, corresponding to message loss during transmission; the explanation for $\langle send[0], rec[0], rec[0]\rangle$ is $\langle input, send[0], rec[0], output, duplicate, rec[0]\rangle$, corresponding to message duplication during transmission; and the explanation for $\langle send[0], rec[0], ack[0], getack[1]\rangle$ is $\langle input, send[0], rec[0], output, ack[0], corrupt, getack[1]\rangle$, corresponding to the bit corruption error during acknowledgment.

Table 2.1: Summary of Δ_{rep} for ABP. "trans" refers to errors during transmission, and "ack" refers to errors during acknowledgements.

Fault types	**Number of Traces**	**Fault types**	**Number of Traces**
trans.lose	23	ack.duplicate	14
trans.duplicate	18	trans.{duplicate,corrupt}	4
trans.corrupt	8	ack.{duplicate,corrupt}	2
ack.lose	22	unexplained	8
ack.corrupt	8	**Total**	**107**

We further grouped the representative traces by the type of fault in their explanations, as shown in Table 2.1. For example, $trans.\{duplicate, corrupt\}$ represents a set of deviations in which the transmitted message is duplicated and then corrupted (e.g., $\langle \ldots, rec[0], duplicate, corrupt, rec[1] \rangle$). There may be multiple representative traces of the same fault type, since the fault may occur at different points during an expected sequence of environmental actions.

Our analysis shows that the ABP is more robust than the naive protocol in handling message loss and duplication, as intended by the protocol design [59]. In addition, the 8 unexplained traces also gave us insight into a type of error that ABP was previously unknown to be robust against; namely, that the sender may receive acknowledgments even when the receiver does not send them. This type of deviation may occur, for example, when a malicious channel generates a dubious acknowledgment to deceive the sender into believing that a message has been delivered.

2.6.4 Radiation Therapy Machine

The second case study focuses on the radiation therapy machine introduced in Section 2.2. Specifically, we compare the robustness of the two designs (i.e., the original design as shown in Figure 2.1 and the redesign involving an additional check to ensure the completion of the mode switching before beam delivery, as shown in Figure 2.3). Then, we show that the redesign is indeed more robust against potential human errors. In particular, to model normative and erroneous human behavior, we adopt the Enhanced Operator Function Model (EOFM) [60], a formal notation for modeling tasks performed over human-machine interfaces. Human behavior modeling has been studied by researchers in human factors and cognitive science [45, 61], and we reuse their results in this case study to demonstrate that our approach can be combined with existing behavior models in fields other than network protocols.

EOFM and deviation model. The Enhanced Operator Function Model (EOFM) [60] is a formal description language for *human task analysis*, a well-established sub-field of human factors that focuses on the design of human operator tasks and related factors (e.g.,

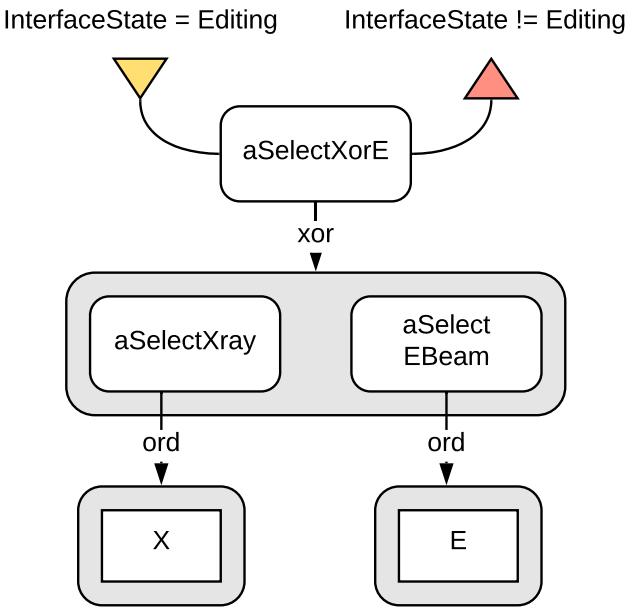


Figure 2.12: The EOFM model of the Beam Selection Task of the therapy machine. A rounded box defines an activity, a rectangular box defines an atomic action, and a rounded box in gray includes all the sub-activities/actions of a parent activity. The labels on the directed arrows are decomposition operators. The triangle in yellow defines the pre-conditions of an activity, and the triangle in red defines the completion conditions.

training, working conditions, and error prevention) [62]. An EOFM describes the task to be performed by an operator over a machine interface as a hierarchical set of *activities*. Each activity includes a set of *conditions* that describe (1) when the activity can be undertaken (*pre-conditions*) and (2) when it is considered complete (*completion conditions*). Each activity is decomposed into lower-level sub-activities and, finally, into atomic interface actions. Decomposition operators are used to specify the temporal relationships between the sub-activities or actions. The EOFM language is based on XML, and it also supports a tree-like visual notation.

Figure 2.12 shows a fragment of the EOFM model of the operator's tasks for the radiation therapy machine (adopted from [63]). It defines the Beam Selection Task, which can be performed only if the interface is in the Editing state; the operator can select *either* the X-ray or Electron beam by pressing X or E, respectively; and the activity is completed only if the interface leaves the editing state.

To generate explanations for robustness that involve human errors, we adopted a method for automatically augmenting a model of a normative operator task (specified in EOFM) with additional behaviors that correspond to human errors [64]. In particular, this approach leverages a catalog of human errors called *genotypes* [45]. For example, one type of genotype errors, named *commission*, describes errors where the operator accidentally performs an activity under the wrong condition. Other genotype errors include omission (skipping an activity) and repetition.

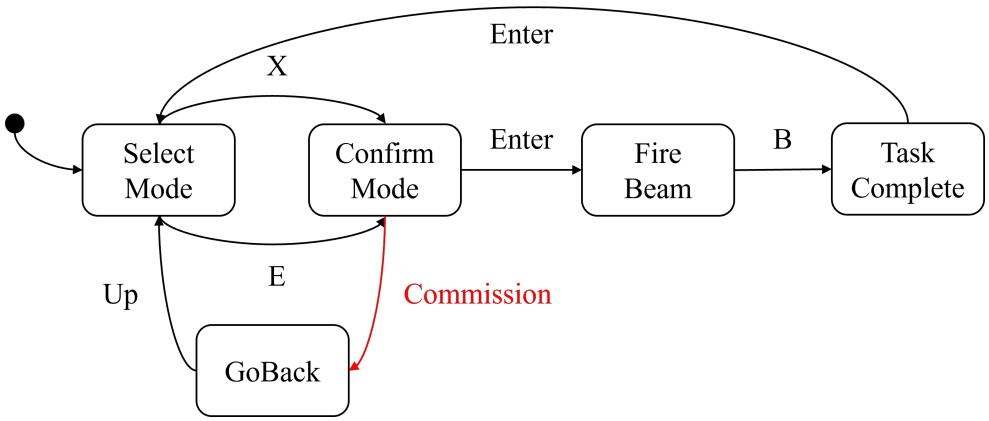


Figure 2.13: A partial deviation model of the operator task for the therapy machine generated from an EOFM.

Figure 2.13 shows a simplified version of the deviation model that was generated from the EOFM model of the therapist's task. This model captures the operator making a potential commission error; i.e., deviating from the expected task by pressing Up. For simplicity, we only show one faulty transition here; the complete deviation model is considerably more complex, as commission, omission, or repetition errors can occur at any state in the normative operator model.

Comparing robustness of M and M_R. We compared the robustness of the two designs by computing $\mathcal{X} = \Delta(M_R, E, P) - \Delta(M, E, P)$ using Equation 2.4, where $M_R = M_I'||M_B||M_S$ is the redesign of M as illustrated in Figure 2.3, and generated representative traces that illustrate differences in their robustness. Specifically, $\mathcal{X}$ contains one equivalence class with a representative trace $\langle$X, Up, E, Enter, B$\rangle$. This shows that the redesign is indeed robust against the operator error involving the mode switch from X-ray to Electron beam. Moreover, we used the deviation model described above to generate the following minimal explanation for this deviation: $\langle$X, Commission, Up, E, Enter, B$\rangle$, corresponding to the operator making a *commission* error by unexpectedly pressing Up during the task.

In addition, computing $\Delta(M, E, P) - \Delta(M_R, E, P)$ yielded an empty set, demonstrating that the redesign of the machine is strictly more robust than the original design.

Comparing robustness under two properties. Recall that the safety property P for this therapy machine example states that the machine should not fire X-ray when the spreader is out of place. It may also be desirable to ensure that the machine does not fire the Electron beam when the spreader is in place (for instance, resulting in under-dose, which, while not as life-threatening as overdose, is still considered a critical error). Let P' be a property stating that the system must prevent both overdose as well as underdose by ensuring the right mode of beam depending on the configuration of the spreader. Intuitively, P' is a stronger property than P.

To compare the robustness of the system against these two properties, we computed $\mathcal{X} = \Delta(M, E, P) - \Delta(M, E, P')$ using Equation 2.5. Our tool returned one representative

trace $\langle \mathsf{E}, \mathsf{Up}, \mathsf{X}, \mathsf{Enter}, \mathsf{B} \rangle$. Since this class of deviations is allowed in $\Delta(M, E, P)$ but not in $\Delta(M, E, P')$, we can conclude from this analysis that the machine (as expected) is less robust in establishing the stronger property P' under potential human errors.

2.6.5 Other Case Studies

We have demonstrated the applicability of our approach (RQ1) to compute and compare robustness through two case studies: one from the distributed systems domain and the other from the human-machine interface domain. We then introduce three additional case studies to further demonstrate the applicability of our approach and to build a set of benchmark problems for scalability evaluation.

Voting machine. We consider a simplified design of an electronic voting machine (called ES&S iVotronic, described in more detail in [65]), which was used in several state-wide elections in the U.S. and was involved in an election fraud [66]. In a normal scenario, a voter is expected to first enter a password to verify their identity, select and vote for the candidate they want, and finally confirm their choice. During the election, the machine was exploited by malicious actors who were able to commit voter fraud by modifying the vote selection made by other voters. To identify potential voter fraud in the design, we define the following integrity requirement: *For each voter, the voting machine must record the vote that was selected by that voter.* More details can be found in Appendix A.1.3.

Oyster transportation fare system. We consider the *Oyster* card fare collection protocol used in public transportation in London, UK (described in [67]). In this system, the user taps their Oyster transportation 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. We specify the safety property of interest to be avoiding *card collision*, i.e., *two different methods of payment are used in the same journey.* More details can be found in Appendix A.1.4.

Infusion pump. The goal of this case study is to evaluate our approach on a system that is considerably more complex than the other examples. We consider an infusion pump machine for dispensing medication to patients through tube lines [68]. The machine also includes a built-in battery that activates when the power cable is unplugged and an alarm that turns on when the battery is low. Normally, the operator plugs in the device, configures the medication dose, and starts the dispensation. However, if the cable is accidentally unplugged and the battery runs out during dispensation, this may cause serious medical accidents, such as overdose. Thus, the safety property is to guarantee that *the machine should immediately stop any ongoing dispensation if the machine loses power.* More details can be found in Appendix A.1.5.

Table 2.2: Evaluation results of robustness computation.

Problem	$\|M\|\|P\|$	**Number of Transitions**	**Time (s)**
Therapy	35	66	0.098
Naive Protocol	41	134	0.102
ABP	23	80	0.177
Voting-1-1	69	283	0.150
Voting-2-2	373	2,301	0.178
Voting-3-3	1,109	9,087	0.230
Voting-4-4	2,469	25,201	0.296
Oyster-1	289	1,800	0.203
Oyster-3	961	8,040	0.31
Oyster-5	2,017	21,000	0.522
Pump-1	163	730	0.141
Pump-2	1,679	9,946	0.263
Pump-3	19,435	144,652	1.097

2.6.6 Experimental Results

To evaluate the scalability of our approach (RQ2), we built a set of benchmark problems based on the five case studies. Specifically, we scaled up: the voting machine problem by increasing the number of voters and election officials, denoted by Voting-N-M where N is the number of voters and M is the number of officials; the Oyster problem by increasing the upper bound of the balance, denoted by Oyster-N where N is the balance bound; and the infusion pump problem by increasing the number of tube lines, denoted by Pump-N where N is the number of lines. Then, Table 2.2 shows the evaluation results.

The complexity of the robustness computation process is exponential to the size of the composition of the machine M and property P, i.e., $O(2^{|M||P|})$, due to the subset construction process in the weakest assumption generation. However, the table shows that our implementation can efficiently compute robustness even for a very large model like Pump-3 with 19,435 states and 144,652 transitions in 1.097 seconds.

2.7 Summary

Strengths of our robustness analysis. In our envisioned robust-by-design development process, a developer begins by constructing a candidate design that satisfies a desired property under a normative environment (i.e., one without any erroneous behaviors). Then,

the robustness computed over this initial design reveals the deviations that the machine is robust against. The process can also be adjusted to find deviations that the machine cannot tolerate. Based on this information, the machine can be redesigned with an improvement process and analyzed again to compute its new robustness. This chapter presents a method to systematically and rigorously compute robustness.

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

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

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

Our analysis approach could also support the development of *mixed-criticality* systems [69], where some of the system requirements are considered more critical than others (e.g., in certain distributed systems, preventing network message corruption may be more important than ensuring timely delivery). Such a system should be designed to satisfy its critical properties even under a faulty environment, while it might be considered acceptable for other, non-critical properties to be violated under the same situation. By applying the property comparison analysis as stated in Problem 2.3, the developer can rigorously check whether a given design achieves appropriate levels of robustness for properties with different levels of criticality.

Chapter 3

Robustification of Designs by Control

3.1 Introduction

In Chapter 2, we present our robustness analysis approach that computes robustness as a set of deviation traces under which a machine continues to satisfy a desired safety property. In contrast, *design robustification* addresses the opposite case, i.e., given deviations under which the machine is unsafe, how to find a new design such that it can satisfy the desired property under those deviations. To distinguish these deviations from the deviations that the machine is robust against, we call them *intolerable deviations*, denoted by $\bar{\delta}$.

In this chapter, we present our technique called *robustification-by-control* as an approach to systematically improving the robustness of a software machine at the design stage. In particular, given models of a machine M and its environment E specified in labeled transition systems, along with a set of intolerable deviations $\bar{\delta}$, our approach *robustifies* M into a new design, M', such that M' is capable of satisfying the desired safety property P even under those deviations. For instance, given the model of the radiation therapy machine M, the expected operator behavior E, and a set of intolerable human errors $\bar{\delta}$, the robustification-by-control method constructs a redesign M' that prevents a safety failure (e.g., patient overdose) even when the operator commits one of those errors.

There are a number of technical challenges to overcome in developing an effective robustification method. First, the space of possible candidate redesigns M' can be enormous, so an effective method must be able to efficiently search this space. Second, not all of these redesigns may be desirable; there are trade-offs between different redesigns.

Specifically, we consider four trade-off dimensions of robustification: *safety*, *functionality*, *controllability and observability*, and *cost*. The goal of robustification is to find a new design satisfying a desired safety property under a set of intolerable deviations. Safety can be achieved by disabling actions of the machine, which, however, may hurt the functionality of the machine. In an extreme case, a machine entering a termination state and doing nothing is deemed to be robust based on our robustness definition (Definition 2.2). Therefore, we would expect our redesign to maintain as much functionality as possible.

When the desired safety property and certain functionality cannot be satisfied simultaneously, one reason may be the lack of controllability and observability of the machine.

For example, if the interface of the therapy machine cannot observe and synchronize on the mode switching events, then it may not be able to prevent the user from firing the wrong beam. However, enhanced controllability and observability often come at a higher cost. Therefore, the robustification method would also need to balance the cost.

To capture these trade-offs of a candidate new design, we introduce two types of quality metrics: (1) the amount of *common behavior* with respect to the original design M to capture the retained functionality, and (2) the *cost of change*, which is reflected by the amount of controllability and observability of the new design. The robustification process then becomes a multi-objective optimization problem [70], where the goal is to find a redesign M' that preserves as much of the existing behavior as possible while minimizing the cost of changes incurred. We present a novel robustification method that leverages techniques from supervisory control theory [39] to automatically generate a set of optimal candidate redesigns.

The rest of this chapter is organized as follows:

- Section 3.4 formally defines a *robustification-by-control* problem and a formulation of it as a multi-objective optimization problem over two quality metrics for robustified designs;
- Section 3.5 presents a novel approach to the robustification-by-control problem that leverages supervisory control theory;
- Section 3.6 presents a set of heuristics for efficiently generating optimal redesigns;
- Section 3.7 presents the implementation of our approach and the evaluation on a set of case studies.

3.2 Motivating Example

We continue using the radiation therapy machine to illustrate the idea of robustification-by-control. Consider a *deviated* environment model E' in Figure 3.1. The model contains a deviation trace $\langle$X, Commission, Up, E, Enter, B$\rangle$ where Commission represents a type of human error of committing unexpected actions. We can check that the therapy machine is not robust against it through techniques like model checking. Specifically, this intolerable deviation 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 the transition from X-ray to Electron beam while the spreader is out of place, which causes a safety violation.

Therefore, the robustification-by-control problem defines the process of finding a new therapy machine design M' such that M' is robust against the deviated environment by "controlling" (modifying) the old design M. More precisely, the *robustification-by-control* task is defined as follows:

> *Given machine design M, environment E, intolerable deviations $\bar{\delta}$, and safety property P such that $M||E' \not\models P$ for deviated environment E' where $beh(E') = beh(E) \cup \bar{\delta}$, construct a new design M' such that $M'||E' \models P$.*

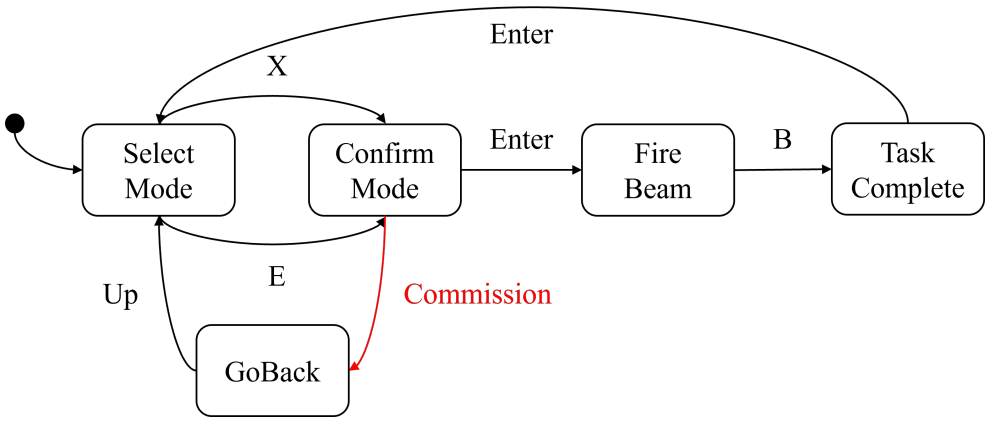


Figure 3.1: A deviated environment model E' with *commission errors* for the radiation therapy machine, under which the original design M is not robust.

Not every solution to this problem, however, may be desirable to the developer. For example, one possible way is to remove all of the Up transitions from the interface; this way, the therapy machine would be prevented from changing the beam mode, thus ensuring that the new design satisfies the safety requirement. However, this design is also less desirable, in that it also removes the ability for the user to change between different modes before firing.

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

> *Given machine design M, environment E, intolerable deviation $\bar{\delta}$, and safety property P, for E' where $beh(E') = beh(E) \cup \bar{\delta}$, construct M' such that $M'||E' \models P$, and M' maximizes common behavior with M and minimizes the cost of change.*

In the following sections, we formally define the robustification-by-control problem and a notion of optimal redesigns in terms of the above quality metrics. We then present our method to generate optimal redesigns, which leverages supervisory control theory [39].

3.3 Preliminaries

Our proposed robustification approach leverages techniques from an area of control theory called *supervisory control* [39]. 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 from occurring.

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

A typical controller C 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 3.1. *A supervisory controller is a function*

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

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

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

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

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 [39].

3.4 Robustification Problems

3.4.1 Basic Robustification Problem

Robustification deals with intolerable deviations under which the machine may violate the property. However, the intolerable deviations $\bar{\delta}$ could represent a potentially infinite set of traces. Therefore, to practically compute the deviated environment E' with respect to deviations $\bar{\delta}$, we introduce the concepts of a *deviation augmentation model* and the

augmentation operator $\oplus$. A deviation augmentation model describes how the environment may deviate from its original behavior, in terms of additional transitions, states, or events:

Definition 3.3. *Given an LTS* $T = \langle S, \alpha T, R, s_0 \rangle$ *and a deviation augmentation model* $\mathfrak{d} = \langle S_\mathfrak{d}, \alpha\mathfrak{d}, R_\mathfrak{d} \rangle$*, where* $S \subseteq S_\mathfrak{d}$*,* $\alpha T \subseteq \alpha\mathfrak{d}$*, and* $R_\mathfrak{d} \subseteq S_\mathfrak{d} \times \alpha\mathfrak{d} \times S_\mathfrak{d}$*, the* augmentation *operator* $\oplus$ *augments* T *by adding states and transitions to it, i.e.,* $T \oplus \mathfrak{d} = \langle S_\mathfrak{d}, \alpha\mathfrak{d}, R \cup R_\mathfrak{d}, s_0 \rangle$*, and* $beh(T) \subseteq beh(T \oplus \mathfrak{d})$*.*

For example, in Figure 3.1, to model the deviation from the expected operator behavior, the original environment model is augmented with an additional state GoBack and additional transitions (ConfirmMode, Commission, GoBack) and (GoBack, Up, SelectMode).

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

Definition 3.4. *Given machine* M*, environment* E*, deviation augmentation model* $\mathfrak{d}$*, and property* P *such that* $M||E \models P$*, the goal of robustification-by-control,* $\mathfrak{R}(M, E, \mathfrak{d}, P)$*, is to find an LTS* M' *such that for* $E' = E \oplus \mathfrak{d}$*,* $M'||E' \models P$*.*

3.4.2 Constraints on Robustified Designs

In this work, we specifically focus on robustification against safety properties. A safety property defines the unsafe behavior that should be avoided. However, it is possible to have an overly restrictive M' that satisfies the safety property but does nothing "meaningful". For instance, in the radiation therapy machine, one could disable all B events, but this solution would also prevent users from being able to fire the beam. Therefore, to avoid such "useless" solutions, we introduce another class of properties named *progress properties*.

For an LTS T, a *progress* property $Pg \subseteq \alpha T$ defines a set of events such that the system T must eventually be able to execute $a \in Pg$ along all paths.[1] It is a restricted subset of liveness properties [9]. Thus, in the therapy machine example, we can specify a progress property requiring that event B can eventually occur along all traces. Through this way, we can constrain a robustification-by-control problem to find solutions that at least satisfy certain functionalities.

3.4.3 Quality Metrics for Robustified Designs

In general, there may be a large number of possible solutions to a robustification problem, but some of them may be considered more desirable than others, even if they may all satisfy certain progress constraints. Therefore, we consider two desirable qualities of a robustified design: (1) the redesign should retain as much of the important functionality from the original design as possible, and (2) the cost of modifying M to M' should be small.

Common behavior. To capture the amount of common behavior or, in other words, the retained functionality, we introduce the notion of *preferred behavior*. A preferred behavior b is an execution trace and represents an operational scenario that the developer wishes

[1] Note that our definition of progress is slightly different from the one used in LTSA [48], where they require an event to occur infinitely often in an infinite trace. However, we only require the event to occur at least once in a finite trace.

for an LTS T to contain, i.e., $b \in beh(T\uparrow \alpha b)$, where $\alpha b \subseteq \alpha T$ refers to the events of trace b. We denote it as $T \models b$ when a preferred behavior b is satisfied by the LTS T. Then, maximizing the common behavior between the original design M and the new design M' can be formulated as maximizing the number of b's such that $M||E' \models b$ and $M'||E' \models b$. Formally, we define:

Definition 3.5. *Given a set of preferred behaviors $B = \{b_1, b_2, \ldots, b_n\}$, we state $T \models B$ for some LTS T if and only if $\bigwedge_{b_i \in B} T \models b_i$.*

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

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

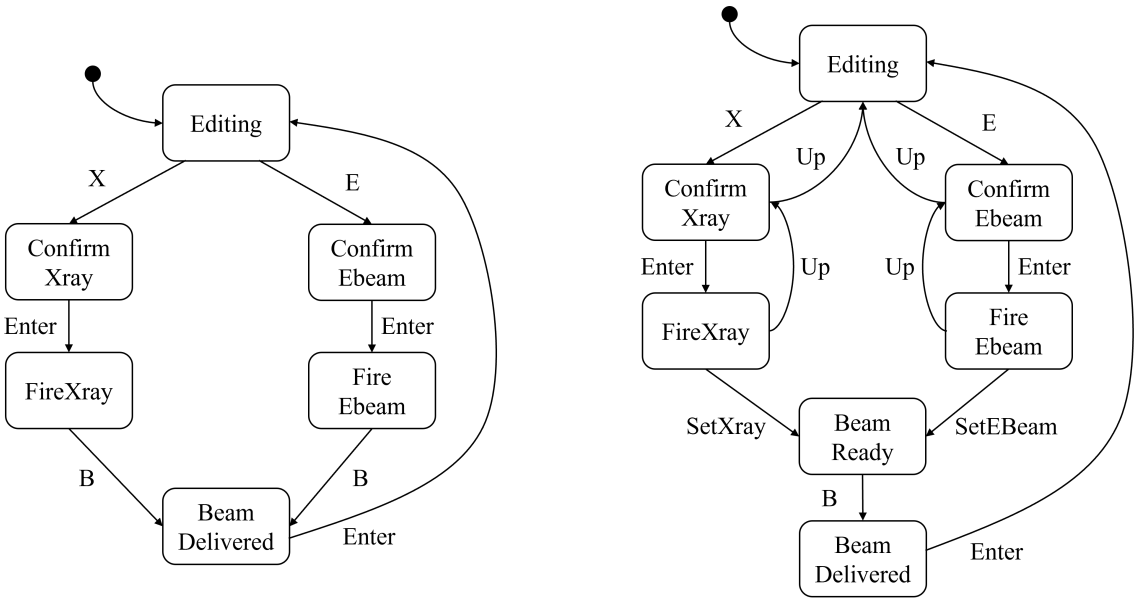
Instead of syntactic changes to an LTS, our intuition is that the cost of redesign can be better approximated by reflecting the set of *environment and machine events that are observed or controlled* by the machine for the purpose of robustification. Intuitively, to make the machine more robust, one may need to place an additional sensor or detector to observe a part of the environment or some internal event of the machine (e.g., add a synchronization sensor for the mode switching in the therapy machine example) or modify an existing actuator to disable a particular event under certain situations (e.g., block the firing button B when the mode switching has not been completed). These types of changes in the sensing and actuating capability of the machine may better reflect the cost of implementation than just counting the syntactic changes of the model.

More precisely, the developer can designate 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 can be associated with a cost measure to reflect the effort of implementing an actuator or sensor to control or observe (respectively) that event in the real world. This, in turn, allows us to measure the total cost of changes as the sum of the individual costs of the events in A that are used to robustify the machine.

Example. In the radiation therapy machine example, one can define preferred behaviors:

$$b_1 : \langle \mathsf{X},\ \mathsf{Up},\ \mathsf{E},\ \mathsf{Enter},\ \mathsf{B} \rangle \text{ and } b_2 : \langle \mathsf{E},\ \mathsf{Up},\ \mathsf{X},\ \mathsf{Enter},\ \mathsf{B} \rangle$$

to specify the desire that the user should be able to switch between the X-ray mode and the Electron beam mode using the Up button. Then, one can assign a moderate cost to events SetXray and SetEBeam for observability to reflect the cost to implement the sensing capability in the interface to synchronize mode switching. They can also assign a cheap cost to events Up and B for controllability to reflect the cost of (1) disabling Up to stop



(a) Redesign by disabling *Up*. **(b) Redesign by synchronizing on *SetXray* and *SetEBeam*.**

Figure 3.2: Alternative ways to robustify the radiation therapy machine.

the user from mode switching and (2) controlling B accordingly to avoid accidentally firing the wrong beam.

Then, Figure 3.2 shows two alternative ways to robustify the radiation therapy machine. In solution (a), we remove all the Up transitions from the interface model; alternatively, in (b), we let the interface synchronize on the SetXray and SetEBeam events from the beam setter and control the B button to prevent firing a beam before the mode has been properly set.

Consider the progress property $Pg = \{\mathsf{B}\}$ that requires B to eventually occur. Both solutions satisfy this constraint; but in terms of preserving behaviors of the original design, it is easy to see that solution (a) does not satisfy the preferred behaviors (b_1 and b_2) while (b) does. However, in terms of the cost of changes, solution (a) involves removing all Up actions (e.g., removing the corresponding button) from the interface, which is considered to have a cheap cost based on our assumed setting. On the other hand, solution (b) requires extending the interface with additional detectors and controlling the firing button (B) to determine when the beam mode has been successfully set and is safe to fire, which has a higher cost than disabling only Up in our setting.

3.4.4 Optimal Robustification Problem

Given a robustification problem $\mathfrak{R}(M, E, \mathfrak{d}, P)$, a progress property Pg, preferred behaviors B, and modifiable events A, let $\vec{R} = \langle M', B', A' \rangle$ be a solution such that it satisfies the progress property Pg and a subset of preferred behaviors $B' \subseteq B$ using a subset of events

$A' = (A'_c, A'_o)$ where $A'_c \subseteq A_c$ and $A'_o \subseteq A_o$. We define the following objective function:

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

where

- $U_B(\vec{R}) = \sum_{b_i \in B'} u_b(b_i)$ is the amount of utility gained from fulfilling the preferred behaviors, and
- $U_A(\vec{R}) = \sum_{a_c \in A'_c} u_c(a_c) + \sum_{a_o \in A'_o} u_o(a_o)$ is the total cost of events used to redesign M.

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

Intuitively, using a larger set of events to modify M (i.e., increasing controllability and observability) allows for a more fine-grained control over the behavior of the machine, which can help maximize the preferred behaviors, i.e., a larger $U_B(\vec{R})$. However, modifying more events also leads to a higher cost, i.e., a larger negative value of $U_A(\vec{R})$. Thus, the problem becomes a *multi-objective optimization* problem that attempts to generate a solution that maximizes these two conflicting objectives [70]. Formally, this (constrained) optimization problem, denoted $\mathfrak{O}(Rb, Pg, B, A, \vec{U})$, is defined as follows:

Definition 3.6. *Given a robustification-by-control problem $Rb = \mathfrak{R}(M, E, \mathfrak{d}, P)$, a progress property Pg, a set of preferred behaviors B (where $M||E' \models B$), and a set of available events for robustification $A = (A_c, A_o)$, the goal of optimal robustification-by-control $\mathfrak{O}(Rb, Pg, B, A, \vec{U})$ is to find one or more solutions $\vec{R} = \langle M', B', A' \rangle$ such that M' is a solution to problem Rb, M' satisfies Pg, $M'||E' \models B'$, and $\vec{R}$ maximizes the objective function $\vec{U}$.*

As illustrated in Figure 3.2, there are trade-offs between the amount of preferred behavior retained and the cost of change. Solution (b) retains more behavior than solution (a) does but also incurs a higher implementation cost in our assumed problem setting. In general, the developer may need to examine and consider multiple such design alternatives before selecting the final robustified design. Additionally, the importance value of preserving a particular preferred behavior and the cost associated with a certain modification event highly depend on the development and business context. For example, removing the Up button from the interface could also be costly if its physical part was outsourced to another company.

To compare different robustified designs, for an optimal robustification-by-control problem, we consider the *Pareto order* of its solution space [71]. Specifically, a solution $\vec{R}$ to the problem is *Pareto optimal* if and only if there does not exist another solution $\vec{R}'$ that *dominates* it, i.e.,

$$U_B(\vec{R}') \geq U_B(\vec{R}) \ \wedge \ U_A(\vec{R}') \geq U_A(\vec{R}) \quad \text{and}$$

$$U_B(\vec{R}') > U_B(\vec{R}) \ \vee \ U_A(\vec{R}') > U_A(\vec{R}).$$

Next, we describe an algorithm that leverages supervisory control synthesis to generate a set of alternative Pareto-optimal redesigns [72].

3.5 Optimal Robustification by Control

3.5.1 Basic Robustification as Supervisory Control

The task of robustifying a machine by control can be reduced to a supervisory controller synthesis problem as follows:

Theorem 3.1. *Given a basic robustification-by-control problem $\mathfrak{R}(M, E, \mathfrak{d}, P)$, let C be a solution to the controller synthesis problem $\mathfrak{C}(G, P, \alpha G_c, \alpha G_o)$, where $G = M||E'$, $E' = E \oplus \mathfrak{d}$, and $\alpha G_c \subseteq \alpha G_o \subseteq \alpha G$. Then, $M' = C||M$ is a solution to the robustification-by-control problem where $\alpha M' = \alpha M \cup \alpha G_c \cup \alpha G_o$.*

Proof. Given a robustification-by-control problem, we have $M||E' \not\models P$. Therefore, the composition $G = M||E'$ can be treated as a plant that behaves undesirably (i.e., violates P) and thus needs to be controlled. The resulting controller C describes how the interactions between M and E' should be restricted to ensure P. Thus, composing M and C amounts to augmenting M with the additional control logic in C to ensure P under the deviations. Formally, since $C||G \models P$, then we have $C||(M||E') \models P$. Hence, $M'||E' \models P$ when $M' = C||M$. □

Then, given the characteristics of supervisory control synthesis, by default, it generates the minimally restrictive controller (Definition 3.2), which aligns with our goal of retaining as much behavior from the original design as possible. Additionally, the use of controllable and observable events to synthesize a controller aligns with our goal of using controllability and observability to measure the cost. Therefore, we have the following theorem:

Theorem 3.2. *Given an optimal robustification-by-control problem $\mathfrak{O}(Rb, Pg, B, A, \vec{U})$ and a corresponding controller synthesis problem $\mathfrak{C}(G, P', \alpha G_c, \alpha G_o)$ where P' is the property that combines the safety property P and the progress property Pg, supervisory controller synthesis generates a controller C such that $M' = C||M$ satisfies the progress property Pg and the maximal possible $B' \subseteq B$ for $A' = (\alpha G_c, \alpha G_o)$.*

Proof. The correctness of the theorem is guaranteed by the definition of supervisory controller synthesis (Definition 3.2). Specifically, given progress property Pg, we convert safety property P into a non-prefix-closed property P' and leverage the supervisory controller synthesis process for non-prefix-closed language [39]. Details about this conversion are described below. □

Therefore, we can formulate an algorithm to solve the optimal robustification-by-control problem by using supervisory controller synthesis as a searching primitive.

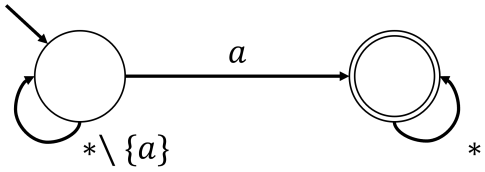


Figure 3.3: The DFA conversion of progress property $Pg = \{a\}$.

Progress property conversion. Given a robustification problem $\mathfrak{O}(Rb, Pg, B, A, \vec{U})$, where $Rb = \mathfrak{R}(M, E, \mathfrak{d}, P)$, we combine the safety property P with the progress property Pg to generate a new non-prefix-closed safety property P', which allows us to solve the problem by leveraging supervisory controller synthesis for non-prefix-closed properties.

Specifically, supervisory controller synthesis considers properties defined in *deterministic finite automata* (DFA). A DFA is a tuple $T = \langle S, \alpha T, R, s_0, F \rangle$ where S is a finite set of states, αT is the set of events of T, $R \subseteq S \times \alpha T \times S$ is the transition function, s_0 is the initial state, and $F \subseteq S$ is the set of accepting states. A DFA only accepts a trace that ends in a state in F. For a safety property P defined in an LTS, we can convert it to a DFA by letting all states in S be acceptable, i.e., $S = F$.

Then, consider a progress property $Pg = \{a\}$. We can combine it into a safety property P by converting it to a DFA, as shown in Figure 3.3. In the figure, a double-solid circle indicates an accepting state of the DFA, and $*$ represents all the events of the safety property P. Thus, this DFA only accepts traces where event a occurs at least once. We can then generate the non-prefixed-closed safety property P' by computing the parallel composition of P with the corresponding DFA of the progress property.

3.5.2 Priority-Based Utility Function

Before diving deep into our algorithm, we introduce a concrete implementation of the utility function u for the objective function $\vec{U}$ of an optimal robustification-by-control problem. We present one definition that assigns utility values based on *priorities* among preferred behaviors and modifiable events: Given the optimization problem $\mathfrak{O}\big(Rb, Pg, B, A = (A_c, A_o)\big)$, the developer assigns priorities to the elements of B, A_c, and A_o. We provide a default set of priority categories as shown in Table 3.1; in general, the priorities can be configured with other user-defined categories.

In Table 3.1, (1) a preferred behavior with a higher priority indicates that it is more

Table 3.1: The priority categories for preferred behaviors and events. Priority 0 is used for events with no cost and does not apply to preferred behaviors.

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

critical to the machine (i.e., has greater utility), and (2) a controllable or observable event with a higher priority means that it is more costly to implement (i.e., has a greater cost). Formally, let h_x represent the priority value (from 0, 1, 2, and 3) of a given preferred behavior, controllable event, or observable event; and let H_k (where $k \in \{0, 1, 2, 3\}$) represent the set of all preferred behaviors and events with priority k. Then, the overall utility function $u = (u_b, u_c, u_o)$ is defined as follows:

$$u_b(b) = \mathcal{W}(h_b), \quad u_c(a) = -\mathcal{W}(h_a), \quad u_o(a) = -\mathcal{W}(h_a), \text{ and}$$
$$\mathcal{W}(i) = 1 + \sum_{k=0}^{i-1} \mathcal{W}(k) \cdot |H_k|$$

where $|H_k|$ is the number of preferred behaviors and events with priority k, and $\mathcal{W} = 0$. Also note that an event a being controllable and observable is considered as two events when counting $|H_k|$.

This approach to defining utility is called the *lexicographic method* [70]. With these rules, the cost of making some event controllable or observable is assigned the negative utility value of fulfilling a preferred behavior in the same priority bracket. Additionally, these rules prioritize saving a cost or fulfilling a preferred behavior in a particular priority bracket over incurring any costs or gaining any utilities with a lower priority. This enables our algorithm to search in the order of higher-to-lower priorities, as discussed later.

Example. In the radiation therapy machine, we can define the scenarios ⟨X, Up, E, Enter, B⟩ and ⟨E, Up, X, Enter, B⟩ as *Essential* preferred behaviors, as it is crucial to ensure the user can switch between beam modes. Satisfying one of these will earn a utility value greater than the total utility of all preferred behaviors with a lower priority (e.g., scenario ⟨X, Enter, Up, Enter, B⟩ could be *Minor*). Similarly, when we assign events SetXray and SetEBeam with a *Moderate* cost, using any of these in robustification will incur a cost higher than the total cost of all *Cheap* events (e.g., events Up and B).

3.5.3 Algorithm for Multi-Objective Optimization

Figure 3.4 illustrates the overall process of our approach. At a high level, a design optimizer generates the next searching target $\langle B', A' \rangle$. Subsequently, we solve a supervisory control synthesis problem with the given A' and then verify if the preferred behaviors in B' are satisfied by the candidate solution. If so, the design optimizer stores this candidate solution M' and iteratively generates the next search target.

Algorithm 2 describes a naive design for finding Pareto-optimal solutions, called NAIVE-PARETO. It employs a top-down, enumerative search approach, where it (1) searches for a solution that fulfills a subset of preferred behaviors $B' \subseteq B$ at the lowest cost possible for B', and (2) iteratively reduces B' to find other Pareto-optimal solutions.

Specifically, on lines 1-2, NAIVEPARETO starts by generating the deviated environment E' and the new property P' combining P and Pg, and synthesizing a controller (C_{max}) that has access to all of the user-specified controllable and observable events (A_c and A_o).

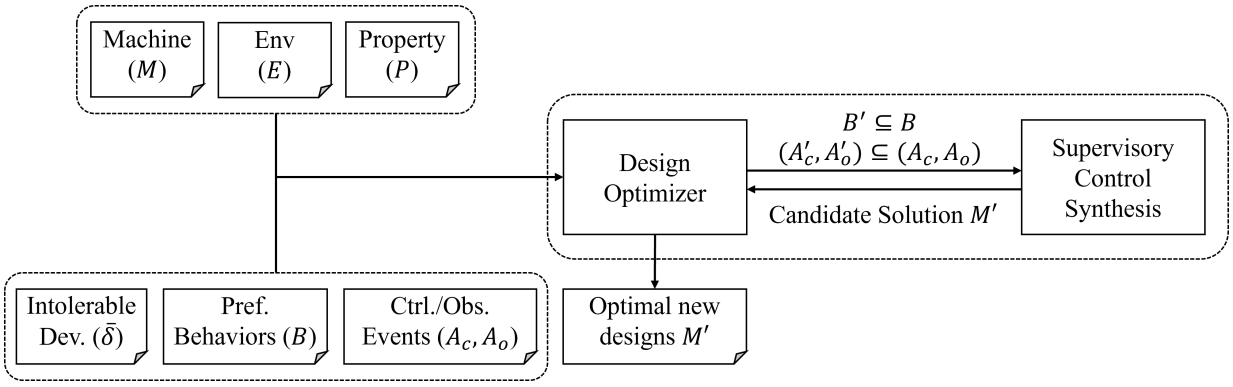


Figure 3.4: The overall process for solving optimal robustification-by-control problems.

Algorithm 2: NAIVEPARETO — A naive algorithm for optimal robustification-by-control.

Input : $M, E, \mathfrak{d}, P, Pg, B, A = (A_c, A_o), \vec{U}$
Output: *solutions*, a set of solutions to robustification-by-control

$E' \leftarrow E \oplus \mathfrak{d},\ P' \leftarrow$ `constructProp(`P, Pg`)` ;
$C_{max} \leftarrow \mathfrak{C}(M||E', P', A_c, A_o)$; `// Solve a controller synthesis problem`
$B_{max} \leftarrow$ `checkPreferred(`$C_{max}||M||E', B$`)` ;
$solutions \leftarrow \emptyset, i \leftarrow 0$;
while *true* **do**
  $\mathbb{B}_i \leftarrow$ `nextToRemove(`B_{max}, i`)` ; `//` $\mathbb{B}_i \subseteq 2^{B_{max}}$
  if $\mathbb{B}_i = \emptyset$ **then**
    return *solutions*;
  end
  for $B_{rm} \in \mathbb{B}_i$ **do**
    $B' \leftarrow B_{max} \setminus B_{rm}$;
    $candidates \leftarrow$ `minimizeCost(`$M||E', P, B', A_c, A_o, \vec{U}$`)` ;
    for $\vec{R} = \langle M', B', A' \rangle \in candidates$ **do**
      $v \leftarrow \vec{U}(\vec{R})$;
      `// Maintain the set of Pareto-optimal solutions.`
      $solutions \leftarrow$ `updateSols(`$solutions, \vec{R}, v$`)`;
    end
  end
  $i \leftarrow i + 1$;
end

Then, on line 3, it checks whether $C_{max}||M||E'$ satisfies each preferred behavior $b \in B$. Since this is the most "powerful" controller, based on Theorem 3.2, it fulfills the maximal subset of the user-specified preferred behaviors (B_{max}), while also being the most costly solution.

In the iteration from lines 5 to 19, NaivePareto incrementally removes elements from B_{max} in the order of utility values to find solutions with a lower cost. On line 6, it generates multiple sets of preferred behaviors to remove for an iteration i, which have the same total utility value. For example, consider $B_{max} = \{b_1, b_2, b_3\}$ where $u_b(b_1) = u_b(b_2)$ and $u_b(b_1) + u_b(b_2) < u_b(b_3)$:

1. At iteration $i = 0$, it generates $\mathbb{B}_0 = \{ \emptyset \}$ to remove from B_{max};
2. At iteration $i = 1$, $\mathbb{B}_1 = \big\{ \{b_1\}, \{b_2\} \big\}$, where $\{b_1\}$ and $\{b_2\}$ have the same total utility value;
3. At iteration $i = 2$, $\mathbb{B}_2 = \big\{ \{b_1, b_2\} \big\}$, etc.

Thus, in each iteration, the preferred behavior sets to remove all have the same total utility value; and the value increases with the increasing of iterations. This process continues until $\mathbb{B}_i$ is empty, i.e., we have explored all subsets of B_{max} to remove (lines 7 to 9).

Then, from lines 10 to 17, at iteration i with, e.g., $\mathbb{B}_i = \{B_1, B_2\}$ where $B_1, B_2 \subset B_{max}$, we remove B_1 and B_2 from B_{max} (line 11), respectively, and try to find solutions with a lower cost given the new preferred behavior sets. In particular, on line 12, given a new preferred behavior set B', NaivePareto enumerates all combinations of controllable and observable events except those where $A_c \not\subseteq A_o$ (which violates our assumption in Theorem 3.1) and attempts to synthesize a controller for each combination. Algorithm 3 describes this process in detail. The goal is to find a controller (if one exists) that fulfills B' at the lowest possible cost. If such a solution exists and is not dominated by any existing solutions, it is stored as one of the Pareto-optimal solutions to be returned as the final output (lines 13 to 16).

Complexity. The complexity of NaivePareto comes from two tasks: (1) searching all possible combinations of preferred behaviors and events, and (2) controller synthesis. For (1), the complexity is $O(2^{|B|+|A^{>0}|})$, where $A^{>0}$ is the subset of A with a non-zero cost. For each combination, the algorithm solves a controller synthesis problem, which is in general a hard problem (NP-hard) [73, 74]. Thus, the worst-case complexity can be approximated as $O(2^{|B|+|A^{>0}|+N})$, where N is the number of states of the plant, $M||E'$.

3.6 Heuristic for Multi-Objective Search

3.6.1 SmartPareto: Searching with Pruning Strategies

Given the inherent complexity and the brute-force nature of the algorithm, NaivePareto is unlikely to scale to larger models. This section introduces SmartPareto, which employs three heuristics to improve the efficiency of finding Pareto-optimal solutions.

Algorithm 3: `minimizeCost` for NAIVEPARETO

Input : $M||E', P', B', A_c, A_o, \vec{U}$
Output: *candidates*, a set of candidate solutions satisfying B' with the lowest cost.

```
1  candidates ← ∅;
2  for A' = (A'_c, A'_o) ∈ 2^{A_c} × 2^{A_o} do
3      if A'_c ⊆ A'_o then
4          C ← 𝔊(M||E', P', A'_c, A'_o);
5          B_sat ← checkPreferred(C||M||E', B');
6          if B_sat = B' then
7              R⃗ ← ⟨C||M||E', B', A'⟩;
8              v ← U⃗(R⃗);
               // Maintain the set of solutions with the lowest
                  cost.
9              candidates ← updateBest(candidates, R⃗, v);
10         end
11     end
12 end
13 return candidates;
```

(1) Removing unnecessary events. By analyzing C_{max}, we can extract αG_u and αG_{un}, which indicates the set of controlled or observed events and unused events, respectively. For any event $a \in \alpha G_{un}$, we can remove it from future searches if its cost is greater than the total cost of events in αG_u. Since we know that αG_u can form a valid solution, any solution with event a would always have a higher cost and, thus, there is no need to search for such solutions. Therefore, after generating C_{max} (line 2 in Algorithm 2) and before entering the main iteration loop (lines 5 to 18), we can remove the unnecessary events from A and use only the remaining events to minimize cost (line 11). Similar reduction of a controller can be found in [75], but without the consideration of cost.

(2) Minimizing cost in the order of event priority. Since we employ a priority-based utility function for $\vec{U}$, which follows a strict ordering property of the lexicographic method, we can always remove high-priority events before low-priority ones. For a combination of controllable and observable events, if removing a high-priority event generates a valid solution, then removing a lower-priority event from it cannot generate a solution with a lower cost. Thus, removing high-priority events first can potentially avoid searching certain combinations, which prunes the search space.

(3) Pruning invalid combinations. When a combination of controllable and observable events produces no controller (i.e., cannot find a controller satisfying the property) or violates some given preferred behavior set B', we can stop minimizing from this combi-

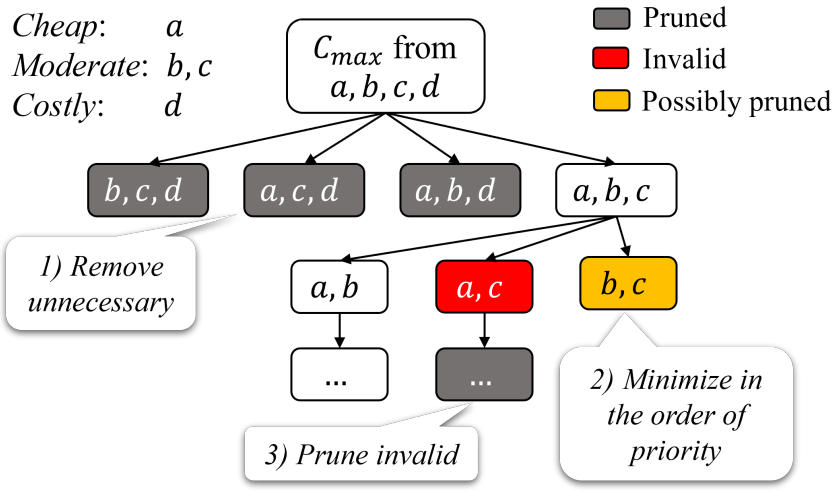


Figure 3.5: Illustration of the heuristics in SmartPareto.

nation. This is because removing events from such a combination would further limit the behavior of the controller, which would certainly result in an invalid solution.

Example. For instance, in Figure 3.5, SmartPareto first generates C_{max} with all events $\{a, b, c, d\}$. By analyzing C_{max}, we know that $\{a, b, c\}$ forms a valid solution and $\{d\}$ is not used. Since d's cost is higher than the total cost of $\{a, b, c\}$, we don't need to search for any combinations containing d (Heuristic 1). Then, to minimize $\{a, b, c\}$, it first removes event b and c, respectively, before a, because if $\{a, b\}$ or $\{a, c\}$ generate a valid solution, then we don't need to search from $\{b, c\}$ as we cannot find a lower cost solution from it (Heuristic 2). Finally, if $\{a, c\}$ is an invalid solution, its followed set $\{a\}$ and $\{c\}$ are also invalid; thus, they do not need to be searched (Heuristic 3).

In our evaluation, we demonstrate that these heuristics improve the performance of the search by significantly reducing the number of controller synthesis calls while still guaranteeing Pareto-optimality.

3.6.2 LocalSearch: Finding Locally Optimal Solutions

As a further improvement to SmartPareto, we present another algorithm called LocalSearch that trades off Pareto-optimality for improved performance.

LocalSearch is similar to NaivePareto but replaces the minimizing process, i.e., the minimizeCost function on line 12 of Algorithm 2. Algorithm 4 shows the process of the new minimizeCost function for LocalSearch. At iteration i with preferred behavior set B', instead of enumerating every possible combination of controllable and observable events (or using heuristics), it incrementally removes one event at a time (high-priority events before low-priority events, controllable events before observable events) from the given event set if removing that event would still generate a controller and retain B'.

Specifically, lines 1 to 2 generate the ordered sequences of controllable and observable

Algorithm 4: `minimizeCost` for LocalSearch

Input : $M||E', P, B', A_c, A_o, \vec{U}$
Output: *candidate*, a local-optimal solution satisfying B'.

```
   // A sorted sequence of controllable events by priority.
1  A_c^ord ← sortByPriority(A_c);
   // A sorted sequence of observable events by priority.
2  A_o^ord ← sortByPriority(A_o);
3  A'_c ← A_c, A'_o ← A_o;
4  for a ∈ A_c^ord ⌢ A_o^ord do
5  |  if a ∈ A_c ∧ |A'_c| > 1 then // Need at least one controllable
   |   events.
6  |  |  A'_c ← A'_c \ {a};
7  |  else if a ∈ A_o ∧ a ∉ A'_c then// Need to maintain A'_c ⊆ A'_o.
8  |  |  A'_o ← A'_o \ {a};
9  |  else
10 |  |  continue;
11 |  end
12 |  C ← 𝔊(M||E', P, A'_c, A'_o);
13 |  B_sat ← checkPreferred(C||M||E', B');
14 |  if B_sat = B' then
   |  |  /* Update the best solution as removing occurs and B'
   |  |     is satisfied.                                    */
15 |  |  R⃗ ← ⟨C||M||E', B', A' = (A'_c, A'_o)⟩;
16 |  end
17 end
18 return R⃗;
```

events based on their priorities. Then, in the loop from lines 4 to 17, it first attempts to remove a controllable event from the combination and then attempts to remove an observable events (lines 5 to 11). It also guarantees there is at least one controllable event and maintains the assumption $A'_c \subseteq A'_o$. On lines 12 to 16, it synthesizes a new controller (if it exists), and if the candidate solution continues to satisfy the preferred behavior set B', it updates the best solution so far. The result is a *local-optimal* solution w.r.t. B' such that removing any event from it would produce no controller or violate B'. However, it does not guarantee the cost to be the minimal and thus is not necessarily Pareto-optimal.

For example, consider events $\{a, b, c, d\}$ where $u(a) = u(b) = u(c) < u(d)$. LocalSearch first removes event d and checks whether a valid solution exists. Then, it arbitrarily selects one of a, b, or c to be removed since they have the same cost. Suppose it removes c and finds that removing either a or b would result in an invalid solution; then, LocalSearch returns $\{a, b\}$ as the optimal solution. This is locally optimal but not necessarily Pareto-optimal, since $\{c\}$ might also allow a valid solution and has a lower cost than $\{a, b\}$.

Complexity. The complexity of LocalSearch is $O(|A^{>0}| \cdot 2^{|B|+N})$. Compared to NaivePareto and SmartPareto, it requires much fewer synthesis instances and thus is more efficient. Although it finds only local-optimal solutions, our evaluation suggests that these solutions are often good enough compared to Pareto-optimal solutions.

3.7 Evaluation

3.7.1 Research Questions

The evaluation of our robustification approach focuses on two research questions:

- **RQ1 (Scalability)**: How well do our robustification algorithms scale? Do the heuristics in SmartPareto improve the performance of NaivePareto? How does LocalSearch compare against the two?
- **RQ2 (Quality of robustification solutions)**: How does our robustification approach compare to other existing methods in terms of the quality of the generated solutions?

To answer these research questions, we evaluated our approach on the five case studies defined in Section 2.6. Specifically, to answer RQ1, we utilized the benchmark problems derived from the case studies and compared the performance of NaivePareto, SmartPareto, and LocalSearch in terms of their solving time. Then, to answer RQ2, we compared our approach against two baseline methods: *Vanilla supervisory control* and *OASIS* (a technique aiming to revise a machine to fulfill a security requirement). In particular, we used: (1) the ability to find a solution that retains behaviors from the old design, (2) the number of controllable and observable events used for robustification, and (3) the solving time to compare the quality of their solutions.

3.7.2 Implementation

We implemented the robustification-by-control approach in our tool Fortis. It uses Supremica [76], a state-of-the-art supervisory controller synthesis tool, to perform controller synthesis as part of the robustification-by-control algorithm. The source code of the implementation is available on GitHub at `https://github.com/cmu-soda/fortis-core`.

Figure 3.6 shows the web interface of Fortis for robustification-by-control. Similar to robustness analysis, a user specifies all parameters of a robustification-by-control problem (such as model specifications, preferred behaviors, controllable and observable events) on the right panel. Then, by clicking Compute, the Fortis back-end will be invoked to compute solutions for a robustification-by-control problem.

Fortis also supports a command-line interface for robustification. A user can provide a JSON configuration file with all problem parameters to Fortis, and the tool produces the solutions on disk in a user-specified specification format such as FSP. We leveraged the command-line interface to conduct our evaluation. All experiments were done on a Windows machine with an Intel i9-12900H processor and 32GB memory.

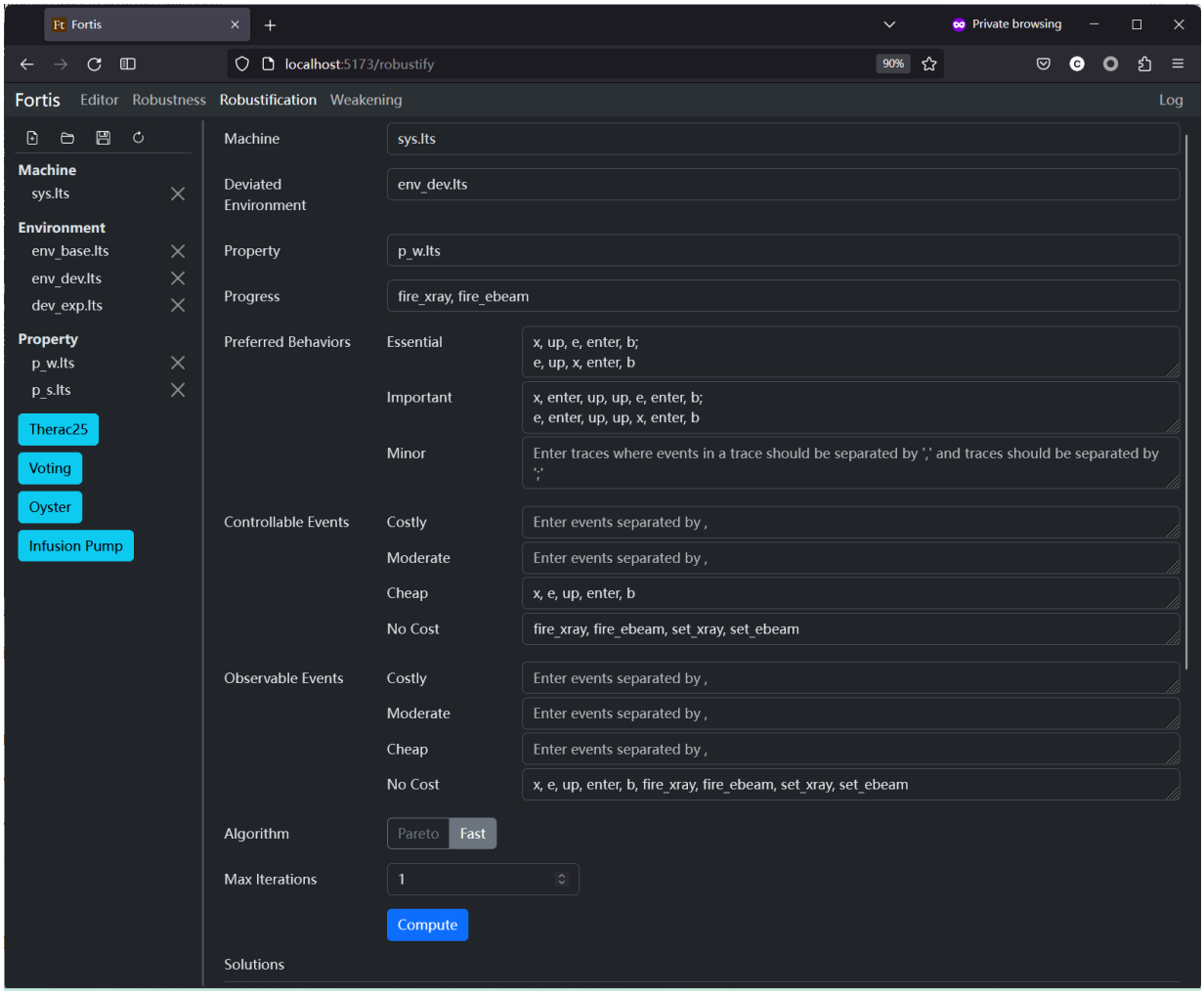


Figure 3.6: Screenshot of Fortis for robustification-by-control.

3.7.3 Case Studies

We first present the settings for all our case study problems and the results found by our approach. We did not apply robustification to the network protocol case study because both the naive protocol and ABP satisfy the safety property under the deviated environment with message loss.

Radiation therapy machine. Consider the radiation therapy machine described in Section 2.2 and the deviated environment described in Section 3.2. We have the safety property that *the spreader must be in place when the beam is fired in X-ray*. Specifically, an unsafe scenario occurs given the deviation $\langle \mathsf{X}, \mathsf{Up}, \mathsf{E}, \mathsf{Enter}, \mathsf{B} \rangle$, where the beam is fired in X-ray with the spread out of place during the transition from the X-ray mode to the Electron beam mode.

Then, we define a progress property that FireXray and FireEBeam should eventually occur to ensure that the machine will still be capable of carrying out treatments even after

robustification. We also define the following preferred behaviors:

- b_1, b_2 (*Essential*): The user can select X/E and then use Up to change the mode and fire the beam.
- b_3, b_4 (*Important*): The user can perform ⟨Up, Up⟩ after having pressed Enter to change the mode and fire.

b_1 and b_2 state that it is *Essential* to allow the user to switch the beam in case the wrong one was accidentally selected. Since it is less likely for the user to select the wrong beam and then press Enter without noticing the mistake, b_3 and b_4 are assigned a lower priority of *Important*.

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

Running Fortis with SmartPareto generated two Pareto-optimal solutions. One solution involves: (1) disabling B when the system is in the X-ray mode and the spreader is out of place, and (2) re-enabling B when the mode switching is completed by observing SetXray and SetEBeam. This solution is similar to the redesign we manually devised in Section 2.2. In addition, LocalSearch finds the other Pareto-optimal solution, which disables Enter instead of B.

Voting machine. Consider the voting machine example, where a voter enters a password to verify their identify, selects and votes for a candidate, and finally confirms their vote. The voting machine is placed inside a voting booth, and only one person can occupy the booth at a time. The safety property aims to ensure vote integrity, that *the machine must record the vote that was selected by a voter*. More details of the problem can be found in Appendix A.1.3.

A deviation of the environment may occur when a voter is not familiar with the e-voting interface and inadvertently commits errors such as omitting to confirm the vote selection before leaving the voting booth. This may lead to a counterexample scenario: After pressing vote to vote for a candidate, the voter exits the voting booth without confirming; a malicious election official then enters the booth, presses back to return to the selection screen, selects the candidate for their own interest, and completes the rest of the voting process—resulting in a possibly incorrect vote being recorded for that voter. This scenario depicts the actual voter fraud that was committed by election officials during an election in Kentucky [66].

To robustify this machine, we define a progress property that the event confirm can eventually take place, indicating that the voter should be able to confirm their vote. We also define the following preferred behavior:

- b_1 (*Essential*): The voter should be able to change their vote by performing ⟨select, back, select, vote, confirm⟩.

We assigned *NoCost* for observing all the internal events of the machine, including password, select, vote, confirm, reset, and back, but *Cheap* cost for controlling them. We also specified

that making {v, eo}.enter and {v, eo}.exit (i.e., a voter or an official enters or exits the voting booth) observable has *Moderate* cost and making them controllable is *Costly*. In practice, these costs might manifest as adding an ID scanner to determine who is entering or exiting (for observability) or a more costly security mechanism (e.g., an enclose booth with a machine-controlled door) to control entry into the booth (for controllability).

With SmartPareto, Fortis returns 6 Pareto-optimal solutions. As an example, one of them requires observing v.exit and controlling confirm. It observes the voter leaving the voting booth, and disables confirm until the election official reset the voting machine. In addition, the LocalSearch method returns one of the Pareto-optimal solutions.

Oyster transportation fare system. Consider the Oyster fare collection system, where a user either taps their Oyster transportation card or uses another payment method such as a credit card when entering the gate, and uses the same method to complete the payment when leaving the gate. The safety property states that *the user should use the same payment method in the same journey.* More details can be found in Appendix A.1.4. A deviation of the environment occurs when a user forgets which method they used on the entry gate and uses another method at the exit gate, leading to a safety violation.

To robustify the machine design, we define a progress property with events indicating that *the user should be able to eventually complete their payment and exit the gate.* Then, we define the following preferred behaviors:

- b_1, b_2 (*Essential*): The user can use their Oyster transportation card (or a credit card) to enter and exit the gate in the same journey.

b_1 and b_2 state the most *essential* functionality of the machine to allow the user to use one of the supported payment methods to enter and exit the gate. Then, we assigned *NoCost* to observe and control all the machine-initiated events such as acknowledging the use of the Oyster card or a credit card and acknowledging that the payment is completed. We assigned *NoCost* to observe the user-initiated events such as tapping the Oyster card or using a credit card when entering or exiting the gate, but *Costly* to control them. In the implementation, controlling these events may require additional mechanism to disable or reject the user input through the interface.

Running SmartPareto, Fortis returns one Pareto-optimal solution. The redesign rejects acknowledging the completion of the payment if the user uses a different payment method when exiting the gate. It waits until the user remembers the correct method they used when entering. In addition, the LocalSearch returns the same Pareto-optimal solution.

Infusion pump. Consider the infusion pump machine that is used to dispense a certain dose of medication through tube lines connected to a patient. The machine is connected to a power system with an alarm and a built-in battery that charges when the power cable is plugged in. When the cable is unplugged during operation, the power system automatically switches to battery mode; and when the battery runs low, it rings the alarm to notify the nurse. More details about the model can be found in Appendix A.1.5.

A deviation may occur in the workflow of a nurse. Normally, the nurse plugs in the cable

and starts the machine; then, the nurse sets up the medication rate, starts the dispensation, and waits for its completion. However, a deviation is that the user accidentally unplugs the cable while the pump machine is still dispensing the medication. In one possible scenario, the battery runs low and the user fails to notice the alarm; then, the machine continues dispensing even when the power fails. This might cause serious medical accidents, such as overdose. Therefore, we consider a safety property that *if the machine loses power during medicine dispensation, it should discontinue the dispensation* and a progress property that *the dispensation must be able to eventually complete*.

To robustify the machine, we define the following two preferred behaviors:

- b_1 (*Essential*): The user should be able to turn on the machine, start it, and wait for the completion of a dispensation, and then turn it off.
- b_2 (*Essential*): The user should be able to resume a dispensation after a power failure.

We define all the machine events to incur *NoCost* to observe. Environmental events like plug_in and battery_charge are *Costly* to observe, except for power_failure, which is made unobservable. All the machine events are free to control, except events like turn_on and turn_off, which are assigned *Moderate* as they might require modifying the user interface. Environmental events like plug_in are *Costly* to control, and physical events like battery_spent and power_failure are uncontrollable. More details about the problem configuration can be found in our source code repository.

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

3.7.4 Experimental Results

RQ1 (Scalability). Table 3.2 summarizes the performance of NaivePareto (with suffix -N), SmartPareto (with suffix -S), and LocalSearch (with suffix -L) over the set of benchmark problems. For scalability evaluation, we also tested them on larger variants of Voting-N-M (where N and M are the number of voters and officials), Oyster-N (where N is the bound on the card balance), and Pump-N (where N is the number of dispensation lines).

It can be seen that NaivePareto requires a large number of synthesis calls and times out on the Voting-2-2, 3-3, 4-4, and Pump-2,3 problems. In comparison, our heuristics for pruning the search space in SmartPareto are effective in reducing the number of synthesis calls, resulting in a significant performance improvement over NaivePareto. The LocalSearch method further improves on the performance by giving up on the Pareto-optimality of the generated solutions. It solves all problems, and the solving time is significantly smaller even than SmartPareto. In addition, we will later show that LocalSearch often finds a solution that is the same as or close to Pareto-optimal solutions, and thus we believe that this is an acceptable compromise between performance and qualities of the redesigns.

We also observe that controller synthesis is the key bottleneck. The time to solve one

Table 3.2: Evaluation results of Fortis generating an optimal solution. All run have a 10-minutes timeout.

Problem	$\|B\|$	$\|A^{>0}\|$	$\|M\|\|E'\|$	**Num of Trans.**	**Num of Synth.**	**Time (s)**
Therapy-N					32	1.032
Therapy-S	4	5	38	72	32	0.929
Therapy-L					6	0.369
Voting-1-1-N					5,168	15.717
Voting-1-1-S	3	14	12	30	140	1.248
Voting-1-1-L					10	0.416
Voting-2-2-N					-	T/O
Voting-2-2-S	3	24	33	76	466	15.414
Voting-2-2-L					17	1.609
Voting-3-3-N					-	T/O
Voting-3-3-S	3	33	47	114	-	T/O
Voting-3-3-L					22	8.820
Voting-4-4-N					-	T/O
Voting-4-4-S	3	42	61	152	-	T/O
Voting-4-4-L					27	95.488
Oyster-1-N					16	0.732
Oyster-1-S	2	4	144	426	1	0.317
Oyster-1-L					1	0.32
Oyster-3-N					16	1.028
Oyster-3-S	2	4	488	2,360	1	0.494
Oyster-3-L					1	0.492
Oyster-5-N					16	1.483
Oyster-5-S	2	4	1,032	6,998	1	0.699
Oyster-5-L					1	0.691
Pump-1-N					2,304	47.786
Pump-1-S	2	12	104	484	99	3.798
Pump-1-L					13	0.937
Pump-2-N					-	T/O
Pump-2-S	4	16	760	4,794	1,059	517.525
Pump-2-L					17	8.706
Pump-3-N					-	T/O
Pump-3-S	6	20	6,248	49,854	-	T/O
Pump-3-L					21	346.211

synthesis instance and the size of the solution space grow quickly with the increasing size of the plant ($M||E'$). Moreover, the synthesis problem becomes harder to solve when fewer controllable and observable events are provided (when minimizing the cost). Thus, our tool timed out on some Voting problems, which have a relatively small plant size but a large number of events to minimize. In contrast, it solved the Oyster problems more efficiently, which have a larger plant size but a much smaller set of events.

RQ2 (Quality of robustification solutions). We compared the quality of redesigns generated by Fortis to those by other approaches for robustifying behavioral models. Specifically, we considered two baseline methods:

- **OASIS**: As far as we know, our definitions of robustification-by-control problems and related qualities are new, and there is no existing tool that is directly comparable. However, one existing work that is close to ours is OASIS by Tun et al. [65]. Although they do not explicitly mention robustness, their goal is similar, in that it aims to revise a machine to fulfill a security requirement in an environment where some of the users might deviate from their expected behavior.

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

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

- **Vanilla supervisory control**: We also compare Fortis to a vanilla approach that utilizes supervisory controller synthesis to generate robustified designs without considering the two quality goals (i.e., it solves the basic robustification-by-control problem).

Since no tool for OASIS is publicly available, we implemented their algorithm with Supremica [76] as the underlying controller synthesis engine. For Vanilla supervisory control and OASIS, the controller synthesis procedure was given access to all the machine events as controllable and observable. Table 3.3 shows the evaluation results of comparing these different methods.

From the table, it can be seen that Fortis is able to generate solutions that satisfy all the preferred behaviors (except for some timed-out cases). On the other hand, Vanilla cannot solve the Voting and Pump problems; OASIS solves the Therapy, Voting-1-1, Oyster, and Pump problems but does not satisfy all the preferred behaviors in Voting-1-1 and Pump. The Vanilla and OASIS approaches assume all machine events are available for generating new designs. By comparison, Fortis is capable of finding solutions that make use of fewer events (and thus, at a lower cost) in Therapy and Oyster. In addition, it finds solutions with fewer controllable events but more observable events in the Voting and Pump problems, while the other two approaches either find no solutions or fail to retain the preferred

Table 3.3: Evaluation results of comparing quality of robustification solutions. All run have a 10-minutes timeout.

Problem	**Vanilla**			**OASIS**			SmartPareto			LocalSearch		
	Sol.*	$\lvert A_u\rvert$†	Time (s)	Sol.	$\lvert A_u\rvert$	Time (s)	Sol.	$\lvert A_u\rvert$	Time (s)	Sol.	$\lvert A_u\rvert$	Time (s)
Therapy	✓	(9, 9)	0.253	✓	(9, 9)	0.245	✓	(5, 9)	0.929	✓	(5, 9)	0.369
Voting-1-1	×	(6, 6)	0.231	⊘	(6, 6)	0.322	✓	(1, 7)	1.248	✓	(1, 7)	0.416
Voting-2-2	×	(8, 8)	0.378	×	(8, 8)	7.846	✓	(1, 10)	15.414	✓	(1, 10)	1.609
Voting-3-3	×	(9, 9)	0.973	×	(9, 9)	152.846	×	-	T/O	✓	(1, 12)	8.820
Voting-4-4	×	(10, 10)	2.864	×	(10, 10)	T/O	×	-	T/O	✓	(1, 14)	95.488
Oyster-1	✓	(10, 11)	0.31	✓	(10, 11)	0.305	✓	(6, 11)	0.317	✓	(6, 11)	0.32
Oyster-3	✓	(14, 15)	0.417	✓	(14, 15)	0.472	✓	(10, 15)	0.494	✓	(10, 15)	0.492
Oyster-5	✓	(18, 19)	0.592	✓	(18, 19)	0.668	✓	(14, 19)	0.699	✓	(14, 19)	0.691
Pump-1	×	(13, 13)	0.269	⊘	(13, 13)	0.765	✓	(7, 14)	3.798	✓	(7, 14)	0.937
Pump-2	×	(24, 24)	0.689	⊘	(24, 24)	5.083	✓	(14, 25)	517.525	✓	(14, 25)	8.706
Pump-3	×	(35, 35)	4.329	⊘	(35, 35)	222.528	×	-	T/O	✓	(21, 36)	346.211

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

† $|A_u| = (|A'_c|, |A'_o|)$ is the number of controllable and observable events used in the solution.

behaviors. Also, in our evaluation, LocalSearch can always find the Pareto-optimal solutions that SmartPareto does.

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

3.7.5 Discussion

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

Vanilla can only restrict, but not extend, the machine behavior; thus, its ability to generate an optimal robustification is limited. Fortis can extend the behavior by increasing the controllability and observability of environmental events (e.g., observing officials entering and exiting in the voting machine example). OASIS does so by abstracting and changing the sequence of events. However, such reordering may prevent it from preserving the behavior of the original design (e.g., Voting-1-1 and Pump) or sometimes result in an unusual design (e.g., in Pump, "starts dispensing" after the system "turns off"). On the other hand, by abstracting the machine and changing its event sequencing, OASIS

can produce alternative designs that are not in the solution space of Fortis. It would be an interesting direction to combine the event-based searching method of Fortis with the abstraction-based method of OASIS, which may enable a more powerful robustification process.

3.8 Summary

This chapter presents an approach, named robustification-by-control, to improve the robustness of a machine design against certain environmental deviations through supervisory control. It enables automatic robustification in our envisioned robust-by-design development process. A developer can first use our robustness analysis technique to identify both the deviations that a machine is robust against and the deviations that the machine cannot tolerate. Then, the developer can use our robustification-by-control approach to robustify the design against those intolerable deviations. Our approach also supports design decisions based on trade-offs between the developer's preferences (i.e., what behavior the new design should retain and whether it is cost-effective).

Our approach assumes deviations that augment additional behaviors to the environment. One might also consider deviations that involve *removing* behaviors from the environment (e.g., by removing transitions or states from the environment model). However, we focus on adding behaviors only, as we believe that this already captures a large and interesting class of deviations where the environment exhibits behaviors beyond those captured in its original model (e.g., security attacks, human errors, etc.).

We leverage supervisory control theory to robustify a design by controlling (disabling) events in the machine and environment given a set of controllable and observable events. According to our robustness definition in Chapter 2, it improves robustness by increasing the disabled behaviors of a machine. However, our approach also allows for extending the machine's behavior by expanding its controllability (controllable events) and observability (observable events) (e.g., observing an event in the environment). Nevertheless, this extending ability is limited by the behavior of the controlled plant, i.e., $M||E'$. It cannot introduce additional behaviors that are not present in the plant (e.g., adding retries to a network protocol).

Chapter 4

Robustification of Designs by Specification Weakening

4.1 Introduction

In Chapter 3, we present how to robustify a machine against certain intolerable deviations by synthesizing a controller. Specifically, the robustification process considers balancing the trade-offs among functionality, controllability and observability, and cost; and it assumes the desired safety property to be unchanged. However, in the cases with a strong safety property, we may not be able to find such a redesign; the redesign M' may become overly restrictive, losing certain critical system functionality; or to maintain as much functionality as possible, the redesign may demand additional controllability and observability, leading to potentially unacceptable implementation costs.

For instance, in the radiation therapy machine example, the safety requirement that "*the spreader should always be in place when the machine is in the X-ray mode*" is stronger than the requirement that "*the spreader should be in place when the machine delivers the beam in X-ray*". Given the design of the therapy machine, shown in Figure 2.1 in Chapter 2, to ensure the stronger safety under deviations (e.g., $\langle \mathsf{X}, \mathsf{Up}, \mathsf{E}, \mathsf{Enter}, \mathsf{B} \rangle$), we must give up the mode-switching capability. However, this new design may not be desirable to the developers. In other words, the goal of ensuring the strong safety property and being robust is conflicting with the goal of being functional. Such conflicts are often the result of the acquisition, specification, and evolution of requirements from multiple stakeholders, and they may also indicate the need for further elicitation of the requirements [38].

To address these challenges caused by the requirement conflicts, this chapter introduces another robustification method called *specification weakening*, which is used alongside the robustification-by-control method to allow generating more feasible redesigns. Specifically, given a machine M, its environment E, a property P, and intolerable deviations $\bar{\delta}$, assume that the robustification-by-control method cannot find a desirable redesign M'. Then, the goal of specification weakening is to identify a *weaker* property P' where $P \Rightarrow P'$ and we can find another redesign M'' through robustification-by-control such that $M''||E' \models P'$ and M'' is more desirable.

In particular, we focus on the conflicts between safety requirements and system functionality. When a safety property is overly restrictive, causing certain functionality to be unfulfilled during robustification, we present a weakening method that can find a weaker safety property under which the robustification-by-control process can find another redesign that satisfies the weaker property while still retaining that functionality. The method considers safety properties specified in Fluent Linear Temporal Logic (FLTL) [77] and leverages Linear Temporal Logic (LTL) learning techniques [78, 41] to synthesize weakened formulas. Specifically, we focus on a particular type of safety property in the form of $\mathbf{G}(\phi \Rightarrow \psi)$, where ϕ and ψ are propositional formulas. Then, our approach finds weakened properties that are *close* to the original safety property by formulating the problem as an instance of LTL learning.

The rest of this chapter is organized as follows:

- A formal definition of the *specification weakening* problem for robustification based on FLTL (Section 4.4);
- A novel approach to specification weakening that leverages LTL learning (Section 4.5);
- An implementation of the approach and evaluation against a set of case studies (Section 4.6).

4.2 Motivating Example

Let's revisit the radiation therapy machine example. Consider a *strong* safety property P of the machine, that the spreader should always be in place when the machine is in the X-ray mode. This can be represented in LTL as:

$$\mathbf{G}(Xray \Rightarrow InPlace)$$

where $Xray$ is a proposition indicating that the machine is in the X-ray mode, and $InPlace$ is a proposition indicating that the spreader is in place. We can use techniques like model checking to assert that the machine, as presented in Figure 2.1 in Chapter 2, satisfies this property when given the normative operator model in Figure 2.2.

Then, consider the deviated environment model E', as shown in Figure 3.1 in Chapter 3, which contains a deviation trace $\langle$X, Commission, Up, E, Enter, B$\rangle$. The original machine design is not robust against this deviation, and we can use the (optimal) robustification-by-control method to robustify the design. For simplicity, let all events be controllable and observable, and trace $\langle$X, Up, E, Enter, B$\rangle$ be a preferred behavior. However, solving this robustification-by-control problem fails to generate a redesign that can preserve this preferred behavior even with all available controllable and observable events. The reason is that the beam setter (M_B) will enter the SwitchToEBeam state during the mode switching from X-ray to Electron beam, where the beam mode is still in X-ray but the spreader is out of place, leading to a safety property violation. Thus, the redesign needs to disable mode switching from X-ray to Electron beam to ensure the safety property.

While the robustification-by-control process can find a redesign, it sacrifices the critical system functionality for mode switching as defined by the preferred behavior $\langle$X, Up,

E, Enter, B⟩. This result is arguably not desirable, despite the fact that all events have been set to be controllable and observable, and the optimal robustification-by-control process has attempted to retain as much functionality as possible. On the other hand, the safety property may be considered too restrictive, as ensuring it under the deviation would conflict with the goal of fulfilling certain important functionality. Therefore, we could then *weaken* the safety property so that the new property still guarantees the most critical safety requirement (e.g., avoiding overdoses) while allowing the generation of a robust redesign that can maintain certain functionality. Thus, in this case, the goal of specification weakening in robustification can be stated as:

> *Given a robustification-by-control problem for machine M, deviated environment E', and safety property P, construct a weakened property P' of P such that, under the new robustification-by-control problem with respect to P', it can find a redesign that retains more functionality that could not be retained before.*

In our example, a solution to such a weakening problem is that the spreader should be in place when the beam is *delivered* in the X-ray mode. Formally, it can be represented in LTL as:

$$\mathbf{G}(Xray \wedge Fired \Rightarrow InPlace)$$

where $Fired$ is a proposition indicating that the beam is fired. This property is weaker because it allows a temporary mismatch between the X-ray mode and the spreader's position, while still guaranteeing the most critical safety requirement of avoiding overdoses. In fact, this is the same safety property that we used in Chapter 2 and 3, where we demonstrated that the robustification-by-control process can find a redesign with respect to this weakened property that tolerates the deviation ⟨X, Up, E, Enter, B⟩. In the following sections, we formally define the specification weakening problem and present our method for synthesizing weakened safety properties by leveraging LTL learning techniques.

4.3 Preliminaries

4.3.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [79] is an extension of propositional logic with temporal operators. Its syntax is as follows:

$$\begin{aligned}\phi \;:=\; & p \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \Rightarrow \psi \mid \\ & \mathbf{G}\phi \mid \mathbf{F}\phi \mid \mathbf{X}\phi \mid \phi \mathbf{U} \psi\end{aligned}$$

where $p \in AP$ is an atomic proposition of a finite set of propositions AP. An LTL formula is interpreted over an *infinite* trace $\sigma \in (2^{AP})^{\omega}$. Specifically, the temporal operators are interpreted as:

- $\sigma, i \models \mathbf{G}\phi$ if and only if $\forall j : i \leq j \Rightarrow \sigma, j \models \phi$.
- $\sigma, i \models \mathbf{F}\phi$ if and only if $\exists j : i \leq j \wedge \sigma, j \models \phi$.

- $\sigma, i \models \mathbf{X}\phi$ if and only if $\sigma, i+1 \models \phi$.
- $\sigma, i \models \phi \mathbf{U} \psi$ if and only if $\exists j : i \leq j \wedge \sigma, j \models \psi$ and $\forall k : i \leq k < j \Rightarrow \sigma, k \models \phi$.

In addition, we use $L^{\omega}(P) \subseteq (2^{AP})^{\omega}$ to denote the language of an LTL property P, which is a set of infinite traces of propositions.

4.3.2 Linear Temporal Logic over Finite Traces

However, in this work, we target finite traces (based on the semantics of LTS in Section 2.3). Thus, we consider the variant of LTL over finite traces, namely LTL_f [80]. LTL_f uses the same syntax as LTL but is interpreted over a *finite* trace $\sigma \in (2^{AP})^*$. Specifically, the semantics of the temporal operators for LTL_f are defined as follows:

- $\sigma, i \models \mathbf{G}\phi$ if and only if $\forall j : i \leq j < |\sigma| \Rightarrow \sigma, j \models \phi$.
- $\sigma, i \models \mathbf{F}\phi$ if and only if $\exists j : i \leq j < |\sigma| \wedge \sigma, j \models \phi$.
- $\sigma, i \models \mathbf{X}\phi$ if and only if $i + 1 < |\sigma|$ and $\sigma, i+1 \models \phi$.
- $\sigma, i \models \phi \mathbf{U} \psi$ if and only if $\exists j : i \leq j < |\sigma| \wedge \sigma, j \models \psi$ and $\forall k : i \leq k < j \Rightarrow \sigma, k \models \phi$.

Thus, the major difference between LTL and LTL_f is that the interpretation of LTL_f depends on the length of a finite trace, denoted by $|\sigma|$. Similarly, we use $L(P) \subseteq (2^{AP})^*$ to denote the language of an LTL_f property P, which is a set of finite traces of propositions.

4.3.3 Fluent Linear Temporal Logic

In LTL, atomic propositions are predicates over state variables of a system. However, for event-based models such as labeled transition systems (LTSs), the behavior of a system is represented by events (actions), and states are not characterized by state variables. Thus, *Fluent Linear Temporal Logic* (FLTL) [77] is an extension to LTL that bridges the gap between state-based models and event-based models, where formulas are built from atomic propositions that are predicates on the occurrence of events, named fluents. A *fluent* represents a proposition that holds after an event occurs and becomes false when terminated by another event.

Given an LTS $T = \langle S, \alpha T, R, s_0 \rangle$, a fluent of it is a tuple $F = \langle \mathcal{I}_F, \mathcal{T}_F, Init_F \rangle$ where $\mathcal{I}_F, \mathcal{T}_F \subset \alpha T$ are the *initiating* actions and *terminating* actions, respectively, $\mathcal{I}_F \cap \mathcal{T}_F = \emptyset$, and $Init_F$ is a Boolean value indicating the fluent F *may* initially be true or false at time zero. For simplicity, when $Init_F$ is not specified, it is assigned to be false by default.

A fluent F corresponds to an atomic proposition p_F when being evaluated against an event trace. For an infinite event trace $\sigma = \langle a_0, a_1, a_2, \ldots \rangle$, we have its corresponding sequence of proposition valuations $\sigma_F = \langle s_0, s_1, s_2, \ldots \rangle$, where the fluent proposition p_F is true at s_i if and only if either of the following conditions holds:

- $Init_F \wedge \forall k \in \mathbb{N} : 0 \leq k \leq i \Rightarrow a_k \notin \mathcal{T}_F$
- $\exists j \in \mathbb{N} : j \leq i \wedge a_j \in \mathcal{I}_F \wedge \forall k \in \mathbb{N} : j < k \leq i \Rightarrow a_k \notin \mathcal{T}_F$

In other words, a fluent F holds at a time i *instantly* if and only if it holds initially or an initiating action $a \in \mathcal{I}_F$ has occurred; and in both conditions, no terminating actions in $\mathcal{T}_F$

have yet occurred. Also, an action has an *immediate* effect on the state values, i.e., each state value s_i corresponds to the immediate result of action a_i at time i.

FLTL introduces a way for converting a trace of events to a trace of state propositions. Then, given a set of fluents $\mathcal{F}$, we can use the same syntax as LTL to define properties over $\mathcal{F}$, and $\mathcal{F}$ corresponds to the set of atomic propositions AP.

FLTL over finite traces. Since we focus on finite traces in this work, we can naturally define FLTL over finite traces, namely FLTL_f. Formally speaking, given a set of fluents $\mathcal{F}$, we have a translation function $\gamma : \alpha\mathcal{F}^* \to (2^{\mathcal{F}})^*$ that maps an event trace to a trace of propositions, where $\alpha\mathcal{F} = \bigcup_{F\in\mathcal{F}}(\mathcal{I}_F \cup \mathcal{T}_F)$. Then, for an *event* trace $\sigma \in \alpha\mathcal{F}^*$, $\sigma, i \models P$ if and only if $\gamma(\sigma), i \models P'$ where P is an FLTL_f property and P' is its LTL_f counterpart.

An FLTL_f property P defines a set of event traces, $beh(P) \subseteq \alpha\mathcal{F}^*$. Then, we can also define its relation to the language of its LTL_f counterpart as follows:

Definition 4.1. *Given a set of fluents $\mathcal{F}$ and an $FLTL_f$ property P, let $\gamma : \alpha\mathcal{F}^* \to (2^{\mathcal{F}})^*$ be its translation function, then we have $beh(P) = \bigcup \gamma^{-1}(\sigma_{\mathcal{F}})$ for all $\sigma_{\mathcal{F}} \in L(P')$, where P' is the LTL_f counterpart of P and $\gamma^{-1} : (2^{\mathcal{F}})^* \to 2^{\alpha\mathcal{F}^*}$ is the inverse function of γ.*

4.3.4 LTL Learning from Examples

In a typical setting, *LTL learning* defines the problem of inferring an LTL formula from positive and negative example traces [78]: Given a set of atomic propositions AP, let $\mathcal{P}, \mathcal{N} \subset (2^{AP})^\omega$ be two (potentially empty) disjoint sets of infinite traces, where $\mathcal{P}$ are *positive examples* and $\mathcal{N}$ are *negative examples*. We call $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ a *sample*. Then, the task of LTL learning is to find a formula ϕ such that $\forall\sigma \in \mathcal{P}$: the trace σ *satisfies* formula ϕ, $\sigma \models \phi$, and $\forall\bar{\sigma} \in \mathcal{N}$: the trace $\bar{\sigma}$ *does not satisfy* formula ϕ, $\bar{\sigma} \not\models \phi$. There exists a trivial solution to separate $\mathcal{P}$ and $\mathcal{N}$ in the form $\bigvee_{a\in\mathcal{P}} \bigwedge_{b\in\mathcal{N}} \varphi_{a,b}$, where $\varphi_{a,b}$ separates each example pair (a, b). However, this solution is obviously overfitting and less helpful in practice. Thus, we are often interested in finding an LTL formula of *minimal* size.

Moreover, in this work, we leverage a more general LTL learning problem, called constrained LTL learning described in [41]. A *constrained LTL learning problem* is defined as a tuple $\langle AP, \mathcal{S}, \Phi, \Psi\rangle$, where AP is a *finite* set of atomic propositions, $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ is a sample, Φ is a first-order predicate that constrains the *syntactic structure* of the learned formula, and Ψ is an optimization objective over the syntactic structure of the formula. The goal of the problem is to find an LTL formula ϕ such that $\forall\sigma \in \mathcal{P} : \sigma \models \phi$, $\forall\bar{\sigma} \in \mathcal{N} : \bar{\sigma} \not\models \phi$, $\Phi\big(\mathsf{syntax}(\phi)\big)$ holds, and $\mathsf{syntax}(\phi)$ optimizes Ψ, where $\mathsf{syntax}(\phi)$ represents the syntactic structure of ϕ. Thus, with this technique, we can learn a formula satisfying a particular pattern (e.g., a specification weakening pattern) instead of learning any arbitrary small formula.

4.4 Robustification by Specification Weakening

4.4.1 Basic Weakening for Robustification

In general, for a safety property P, the problem of specification weakening is to find a new safety property P' such that $P \Rightarrow P'$. Thus, weakening a safety specification means making more behavior acceptable. There are various use scenarios for weakening to resolve requirement conflicts [38]. In the context of design robustification, weakening can be used to, for example, allow for retaining more functionality or reducing implementation costs. In this thesis, we specifically focus on one scenario—when the robustification-by-control process cannot find a solution that satisfies certain functionality defined in a preferred behavior, the goal of weakening is to find a weakened safety property P' such that there is a solution to the new robustification-by-control problem where that preferred behavior is satisfied. The task of this specific use case of weakening is defined as follows:

Definition 4.2. *Given a robustification-by-control problem* $Rb = \mathfrak{R}(M, E, \mathfrak{d}, P)$ *and its corresponding optimal robustification problem* $\mathfrak{O}(Rb, Pg, B, A, \vec{U})$ *such that there exists a preferred behavior* $\bar{b} \in B$ *that cannot be satisfied, the goal of the weakening problem* $\mathfrak{W}(Rb, \bar{b})$ *is to find* P' *such that* $P \Rightarrow P'$ *and there exists a solution to the new problem* $\mathfrak{O}(Rb', Pg, B, A, \vec{U})$ *that satisfies* $\bar{b}$*, where* $Rb' = \mathfrak{R}(M, E, \mathfrak{d}, P')$*.*

Specification weakening may also be used to reduce implementation costs in robustification. This can be achieved by a user manually removing certain controllable or observable events from the problem configuration, which causes some preferred behaviors to become unsatisfied, and then solving a weakening problem with respect to the unsatisfied preferred behaviors. Thus, we argue that weakening for retaining more preferred behaviors is more challenging than weakening for reducing cost and, therefore, requires more research attention.

In addition, a weakened safety property is not guaranteed to enable robustification-by-control to generate solutions that satisfy more preferred behaviors. It also depends on the available controllable and observable events. In our problem setting, we assume that the user has provided sufficient controllable and observable events such that the originally unsatisfied preferred behavior $\bar{b}$ can be satisfied by a weakened property.

4.4.2 FLTL$_f$-Based Specification Weakening

It is unclear and challenging how a safety property modeled as an LTS can be weakened in general. One potential way is to add or remove states and transitions from the model, but the resulting property would likely be hard to understand. Thus, we instead focus on safety properties specified in FLTL$_f$. We can then leverage existing LTL weakening methods to weaken a safety property. Specifically, given a set of fluents $\mathcal{F}$, we address the weakening problem for a safety property defined in the form of

$$\mathbf{G}(\phi \Rightarrow \psi)$$

where ϕ and ψ are propositional formulas without any temporal operators. This is a commonly used pattern in LTL to specify safety invariants, e.g., in GR(1) synthesis [51, 81],

and it also provides us the advantage to "minimally" weaken a safety property.

Syntax-based "minimal" weakening. There may be multiple solutions to a specification weakening problem. However, not every weakened safety property is desirable. For example, it should not be too weak, e.g., becoming $P' = \mathit{true}$, which accepts any behavior as safe. Ideally, we want to *minimally* weaken a safety property in the sense that the weakened property accepts only the "minimal" set of additional traces compared to the old one, which allows us to find a new solution satisfying those originally unsatisfied preferred behaviors.

While ensuring such semantic-based minimal weakening is challenging, the safety invariant pattern $\mathbf{G}(\phi \Rightarrow \psi)$ provides a way to gradually weaken a safety property through syntactic patterns. Intuitively, the pattern $\mathbf{G}(\phi \Rightarrow \psi)$ can be interpreted as: *anytime assumption ϕ holds, the system should guarantee ψ*. Thus, we can append conjunctions to the antecedent ϕ to weaken the assumption of this invariant, or append disjunctions to the consequent ψ to allow more guarantees to be acceptable.

In general, for a safety property in $\mathbf{G}(\phi \Rightarrow \psi)$, where ϕ and ψ are propositional formulas without any temporal operators, ϕ is in *conjunction normal form* (CNF), and ψ is in *disjunction normal form* (DNF), the property can be weakened by:

- appending conjunctions to the antecedent ϕ, i.e., $\phi \wedge \bigwedge \bigvee [\neg] p_i$, or
- appending disjunctions to the consequent ψ, i.e., $\psi \vee \bigvee \bigwedge [\neg] p_j$,

where p_i and p_j are atomic propositions. Moreover, by appending more conjunctions to the antecedent or disjunctions to the consequent, the new property P' is guaranteed to be weaker or equal to the original property P, i.e., $P \Rightarrow P'$. This can be proved by the following theorems:

Theorem 4.1. $\vdash \mathbf{G}(p \Rightarrow q) \Rightarrow \mathbf{G}(p \wedge r \Rightarrow q)$

Proof. Since p, q, r are all propositional formulas, this can be reduced to proving $\vdash (p \Rightarrow q) \Rightarrow (p \wedge r \Rightarrow q)$. By assuming $p \Rightarrow q$ is true, we need to show $p \wedge r \Rightarrow q$. Then, by assuming $p \wedge r$ is true, both p and r are true ($\wedge$-elimination). Thus, from assumption $p \Rightarrow q$, q is true ($\Rightarrow$ -elimination). Then, we have $p \wedge r \Rightarrow q$ ($\Rightarrow$ -introduction). Finally, we conclude that $(p \Rightarrow q) \Rightarrow (p \wedge r \Rightarrow q)$ ($\Rightarrow$ -introduction). □

Theorem 4.2. $\vdash \mathbf{G}(p \Rightarrow q) \Rightarrow \mathbf{G}(p \Rightarrow q \vee r)$

Proof. Since p, q, r are all propositional formulas, this can be reduced to proving $\vdash (p \Rightarrow q) \Rightarrow (p \Rightarrow q \vee r)$. By assuming $p \Rightarrow q$ is true, we need to show $p \Rightarrow q \vee r$. Then, by assuming p is true, q is true ($\Rightarrow$ -elimination). Thus, $q \vee r$ is true ($\vee$-introduction). Then, we have $p \Rightarrow q \vee r$ ($\Rightarrow$ -introduction). Finally, we conclude that $(p \Rightarrow q) \Rightarrow (p \Rightarrow q \vee r)$ ($\Rightarrow$ -introduction). □

Therefore, given these properties of the pattern $\mathbf{G}(\phi \Rightarrow \psi)$, to minimally weaken such a safety property, we aim to find a solution with a *minimal* number of additional conjunctions and disjunctions to ϕ and ψ, respectively. In addition, generally speaking, any safety invariant in $\mathbf{G}\varphi$, where φ is a propositional formula, can be transformed into $\mathbf{G}(\phi \Rightarrow \psi)$ by converting φ into DNF and letting the antecedent be true, i.e., $\top \Rightarrow \varphi$. Furthermore, we

constrain the antecedent ϕ to be in CNF and the consequent ψ to be in DNF for improved readability, where a general transformation process exists. Thus, this pattern can cover a wide range of safety properties in practice.

FLTL$_f$-based weakening problem. Therefore, we can define the specification weakening problem with respect to a safety property in FLTL$_f$ as follows:

Definition 4.3. *Given a specification weakening problem $\mathfrak{W}(Rb, \bar{b})$ where the safety property P is an FLTL$_f$ property with fluents $\mathcal{F}$ and is in the form of $\mathbf{G}(\phi \Rightarrow \psi)$ such that ϕ and ψ are propositional formulas, ϕ is in CNF, and ψ is in DNF, the goal of the FLTL$_f$-based weakening problem $\mathfrak{W}_{\mathcal{F}}(Rb, \bar{b}, \mathcal{F})$ is to find P' such that P' is a solution to $\mathfrak{W}(Rb, \bar{b})$, P' is in the form of $\mathbf{G}((\phi \wedge \bigwedge \bigvee [\neg] p_i) \Rightarrow (\psi \vee \bigvee \bigwedge [\neg] p_j))$ where $p_i, p_j \in \mathcal{F}$, and the formula size of P' is minimized.*[1]

Example. In the motivating example, consider the safety property $\mathbf{G}(Xray \Rightarrow InPlace)$ of the radiation therapy machine, and another fluent $Fired$ indicating whether the beam is fired or not. Then, we can weaken the safety property by adding another conjunction to the assumption (antecedent):

$$\mathbf{G}(Xray \wedge Fired \Rightarrow InPlace)$$

This property is weaker in the sense that the spreader should only be in place when the X-ray beam is fired, while allowing it not to be in place when switching from the X-ray mode to the Electron beam mode.

4.4.3 Safety FLTL$_f$ and Safety LTS

In our robustness analysis and robustification-by-control methods, we consider safety properties defined in LTSs. Therefore, before discussing how we can solve an FLTL$_f$-based weakening problem, we also need to show that a safety property in FLTL$_f$ shares the same semantics as a safety property in LTS, such that it is consistent with our existing techniques. Specifically, according to the definition in Section 2.3, a safety property P in LTS describes a *prefix-closed* set of traces, such that the behavior of a satisfying system should be within this set. Thus, we need to show that an FLTL$_f$ safety property also defines such a set.

We first define *safety properties* in LTL$_f$ [82]:

Definition 4.4. *Given an LTL$_f$ property P with atomic propositions AP, let $L(P) \subseteq (2^{AP})^*$ be its accepted language, P is a safety property if, for any trace $\sigma \notin L(P)$, there is a prefix u of σ such that for any $v \in (2^{AP})^*$, the trace $uv \notin L(P)$. u is a bad prefix of P.*

[1] Note that, in theory, to ensure minimal weakening, we want to add a minimal number of additional conjunctions to ϕ, and the number of atomic propositions in each conjunction should be maximized (similar for the additional disjunctions to ψ). However, in practice, this makes the solution hard to understand with redundant conjunctions and disjunctions. Thus, we choose to only minimize the total size of the solution.

A property in $\mathbf{G}\phi$, where ϕ is a propositional formula, is a safety property [80]. Given this definition, the language of an LTL_f safety property is prefix-closed.

Theorem 4.3. *For an LTL_f safety property P, its language $L(P) \subseteq (2^{AP})^*$ is prefix-closed.*

Proof. For an LTL_f safety property P, consider a trace $\sigma \in L(P)$. Let u be a prefix of σ. If $u \notin L(P)$, then by definition, for any $v \in (2^{AP})^*$, $uv \notin L(P)$. Thus, σ should not be in $L(P)$. Hence, the theorem is proved by contradiction. □

The language of an LTL_f property is defined based on a set of atomic propositions. We then show that an FLTL_f safety property is also prefix-closed with respect to a set of events.

Theorem 4.4. *For a set of fluents $\mathcal{F}$ and an FLTL_f safety property P, its language $beh(P) \subseteq \alpha\mathcal{F}^*$ is prefix-closed.*

Proof. According to Definition 4.1, let $\gamma : \alpha\mathcal{F}^* \to (2^{\mathcal{F}})^*$ be the translation function and $\gamma^{-1} : (2^{\mathcal{F}})^* \to 2^{\alpha\mathcal{F}^*}$ be its inverse function. Then, $beh(P) = \bigcup \gamma^{-1}(\sigma_{\mathcal{F}})$ for all $\sigma_{\mathcal{F}} \in L(P')$, where P' is the LTL_f counterpart of P. For a trace $\sigma \in beh(P)$, $\sigma_{\mathcal{F}} = \gamma(\sigma)$ and $\sigma_{\mathcal{F}} \in L(P')$. Consider u as a prefix of σ, and $u_{\mathcal{F}} = \gamma(u)$. According to the definition of fluents, for any state s_i of $\sigma_{\mathcal{F}}$, its valuation only depends on states s_j such that $j \leq i$. Thus, $u_{\mathcal{F}}$ is also a prefix of $\sigma_{\mathcal{F}}$. Since $\sigma_{\mathcal{F}} \in L(P')$, we have $u_{\mathcal{F}} \in L(P')$. Hence, $u \in beh(P)$, and thus $beh(P)$ is also prefix-closed. □

Finally, we define the satisfaction of an FLTL_f safety property as follows:

Definition 4.5. *For a system T in LTS, a set of fluents $\mathcal{F}$, and an FLTL_f safety property P, $T \models P$ if and only if $beh(T{\uparrow}\alpha\mathcal{F}) \subseteq beh(P)$.*

This definition is necessary because there are no well-established definitions for the satisfaction of LTL_f and FLTL_f. In [82], they propose a definition for *non-terminating transition systems* (e.g., an LTS is a non-terminating transition system that has no designated accepting states): $T \models P$ if and only if for any trace σ of T, there *exists* a finite prefix u of σ such that $u \models P$. However, this definition is problematic for safety properties such as $\mathbf{G}\phi$ in that we can always construct a system T such that all its traces have a shared non-empty prefix u where $u \models \mathbf{G}\phi$. Thus, T is deemed to satisfy $\mathbf{G}\phi$ even when there is a trace σ' that has a state violating ϕ after that prefix u.

Therefore, in this work, we define the semantics of $T \models P$ for T in LTS and P being an FLTL_f safety property as: *$T \models P$ if and only if for any trace σ of T and any prefix u of it, $u \models P$*. Thus, we have shown that the semantics of the satisfaction of an FLTL_f safety property is equivalent to the semantics of a safety property in LTS.

Converting a safety FLTL_f to an LTS. We then show how to convert an FLTL_f safety property to an LTS so it can be seamlessly utilized in our existing robustness analysis and robustification-by-control techniques. Our translation method is a modification of the process proposed by Giannakopoulou et al. [77]. Given a set of fluents $\mathcal{F}$, converting an FLTL_f property P consists of the following steps:

1. Construct a deterministic finite automaton (DFA) $D_{\neg P}$ for $\neg P$, where the alphabet of this automaton is the power set of the fluent propositions $2^{\mathcal{F}}$. This process can be adopted from an existing LTL to Büchi translation, such as [83, 84, 85].
2. Construct automata T_F for all the fluents $F \in \mathcal{F}$, where the transitions indicate the valuation changes of a fluent according to its initiating and terminating events.
3. Construct a synchronizing automaton $Sync$ to synchronize on the changes of the fluent values.
4. Compute the parallel composition of $D_{\neg P}$, all the fluent automata T_F, and the synchronizing automaton $Sync$.
5. Finally, hide the fluent proposition labels ($2^{\mathcal{F}}$) from the composition result.

Specifically, for an FLTL$_f$ formula in $\mathbf{G}\phi$, where ϕ does not contain any temporal operators, the final automaton should have only one accepting state, and no transitions can leave that state. Any trace reaching that state satisfies $\neg\mathbf{G}\phi$. Thus, we can replace that accepting state with an error state π, and the resulting automaton is the LTS representation for the safety property.

Example. Consider the following fluents for the radiation therapy machine:

- $Xray = \langle\{\mathsf{SetXray}\}, \{\mathsf{SetEBeam}, \mathsf{B}\}, false\rangle$
- $InPlace = \langle\{\mathsf{X}\}, \{\mathsf{E}\}, true\rangle$

where fluent $Xray$ indicates whether the machine is in the X-ray mode, and fluent $InPlace$ indicates whether the spreader is in-place. Then, to translate the following safety property $P = \mathbf{G}(Xray \Rightarrow InPlace)$, we first construct the DFA for $\neg P$, as shown in Figure 4.1. Label $true$ is an abbreviation for all propositions in $2^{\mathcal{F}}$.

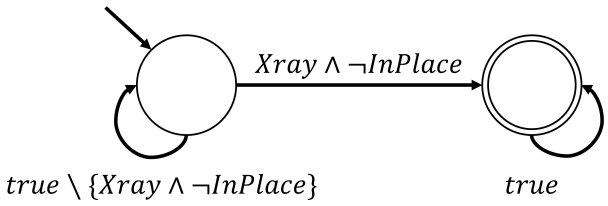


Figure 4.1: The DFA for property $\neg\mathbf{G}(Xray \Rightarrow InPlace)$. The double solid circle indicates the accepting state of a DFA.

Secondly, we construct the automata for the two fluents shown in Figure 4.2. In the automaton for fluent $Xray$, the label $Xray$ is an abbreviation for the labels $\{Xray \wedge InPlace, Xray \wedge \neg InPlace\}$, and the label $\neg Xray$ is an abbreviation for the labels $\{\neg Xray \wedge InPlace, \neg Xray \wedge \neg InPlace\}$. Similarly, in the automaton for fluent $InPlace$, the labels $InPlace$ and $\neg InPlace$ are also abbreviations.

Thirdly, we construct the synchronizing automaton shown in Figure 4.3. Specifically, it forces the final composition to alternate between the system events and the fluent proposition labels so that the DFA for $\neg P$ can update its current state according to an event that has just occurred.

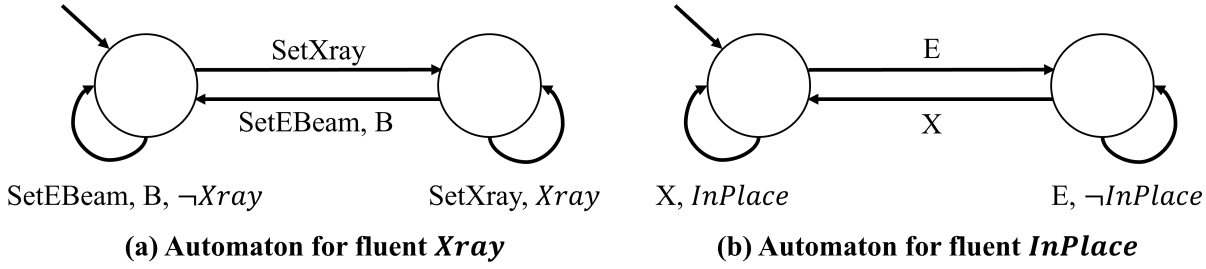


Figure 4.2: The automata for fluents $Xray$ and $InPlace$.

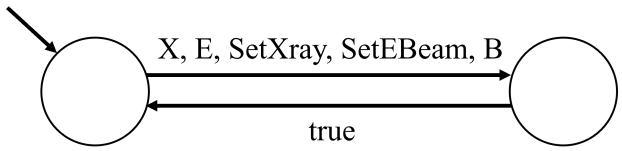


Figure 4.3: The synchronizing automaton for fluents $Xray$ and $InPlace$, which forces the DFA to synchronize on an fluent proposition label after an event.

Finally, we compute the parallel composition of the DFA, the fluent automata, and the synchronizing automaton. After hiding the fluent proposition labels and replacing the accepting state with an error state, we obtain a safety LTS as shown in Figure 4.4. Therefore, by converting an FLTL$_f$ safety property in the form of $\mathbf{G}\phi$ to a safety LTS, we can utilize our existing methods to analyze its robustness and apply robustification-by-control.

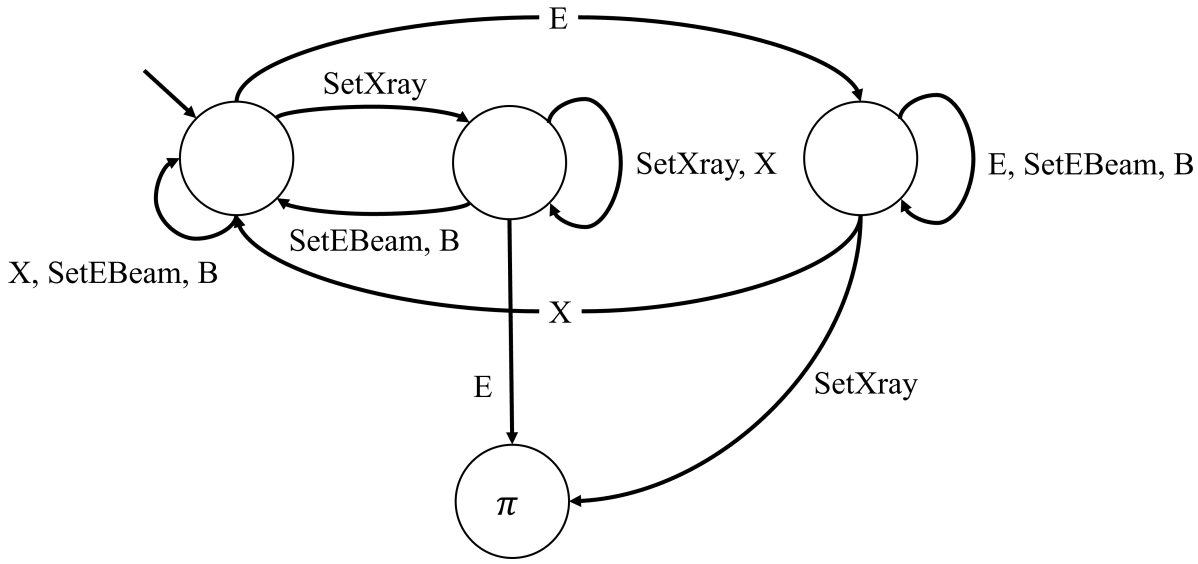


Figure 4.4: The final safety LTS for property $\mathbf{G}(Xray \Rightarrow InPlace)$ where π is the error state.

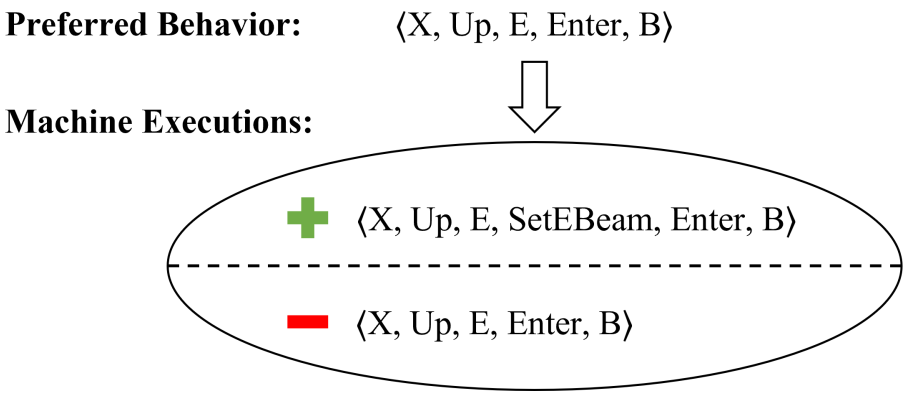


Figure 4.5: Learning examples generation for the radiation therapy machine w.r.t. preferred behavior ⟨X, Up, E, Enter, B⟩.

4.5 Specification Weakening by LTL Learning

4.5.1 Overview

Our specification weakening problem addresses the situation where a preferred behavior cannot be satisfied during the robustification of a machine with respect to a safety property. Given that a preferred behavior b is satisfied when $b \in beh(T \uparrow b)$ where $\alpha b \subseteq \alpha T$, thus preferred behavior b, in fact, represents an abstract scenario of the expected system behaviors. During the design and implementation phase, the abstract design is refined into a more concrete design with the introduction of new, detailed events, and a preferred behavior would map to a set of system executions in the concrete system design. Thus, in the context of robustification, when a preferred behavior cannot be satisfied, it indicates that all of its concrete executions are considered unsafe, even though some may be acceptable. This leads to the conflict between safety and functionality.

For example, in the radiation therapy machine, the preferred behavior ⟨X, Up, E, Enter, B⟩ cannot be satisfied given the safety property $\mathbf{G}(Xray \Rightarrow InPlace)$. This conflict arises because it describes an abstract scenario where a user wishes to switch from X-ray to Electron beam. However, it maps to a set of concrete executions, which can be classified into two categories. One category is that the beam correctly fires in the Electron beam mode, and the other is that the beam fires in the X-ray mode without the spreader. Developers may decide that the first execution is acceptable, but the latter one is not, as shown in Figure 4.5. The original safety property is too restrictive, as it does not allow the first situation. Therefore, a resolution to this conflict could be accepting a weaker safety property $\mathbf{G}(Xray \wedge Fired \Rightarrow InPlace)$. Also note that we assume the machine has enough observability to distinguish these two executions and enough controllability to prevent the machine from entering the unsafe executions.

In requirements engineering, such inconsistencies between different requirements might be desirable to allow further elicitation of the requirements [38]. As shown in Figure 4.6, requirements often come from different stakeholders, which may potentially conflict, as stakeholders view the system from different perspectives (e.g., safety vs. functionality).

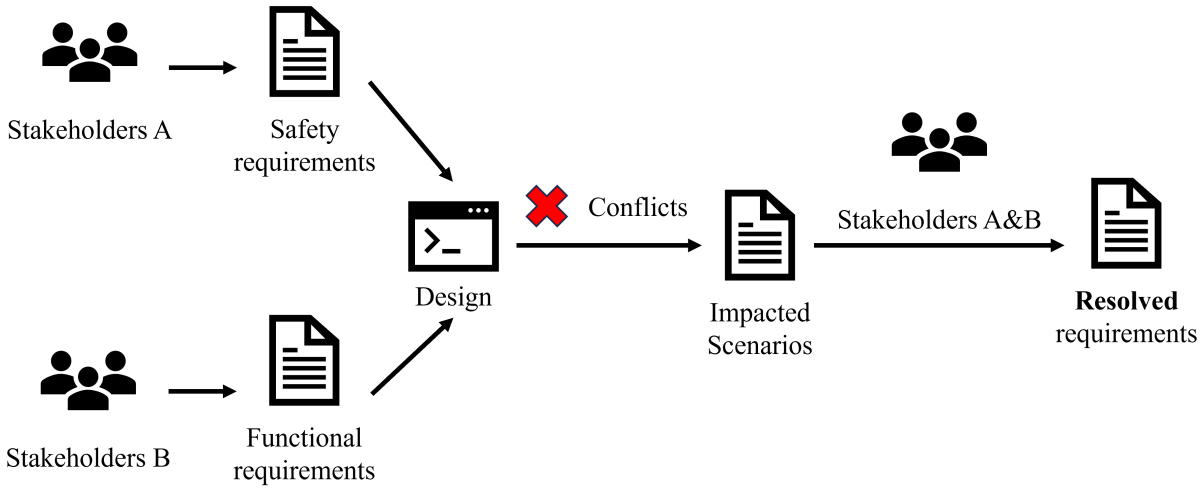


Figure 4.6: A process to resolve the conflicts in the requirements.

Some conflicts may occur during the design or implementation of the machine, due to, for example, unforeseen implementation challenges or requirements evolution. It then becomes necessary for stakeholders and developers to analyze the concrete scenarios affecting how these conflicts arise and resolve them by producing a new set of requirements. Specifically, in our use case, we consider the resolution of conflicts by weakening unnecessarily restrictive safety requirements.

For instance, the aforementioned conflict in the radiation therapy machine may occur due to the unforeseen implementation challenge that the mode switching between X-ray and Electron beam requires more time to complete than the switching of the spreader. Conflicts may also emerge due to requirements evolution. For example, in the Oyster fare collection system, the original, strict safety property requires a user to enter and exit the gate with the same payment method. However, the requirement could evolve to allow a user to enter the gate with their Oyster card but leave with a credit card if their Oyster card is out of balance. This would improve the system's usability but also conflict with the original safety property, which could then be weakened.

We propose a semi-automated workflow to weaken a safety property, as shown in Figure 4.7. Given a specification weakening problem $\mathfrak{W}_{\mathcal{F}}(Rb, \bar{b}, \mathcal{F})$, the process begins by automatically generating example traces $\mathcal{E}$ from the machine M under the deviated environment E' with respect to the unsatisfied preferred behavior $\bar{b}$. These example traces represent concrete system executions that all violate the original safety property P. Next, stakeholders and developers review the traces, marking acceptable examples as positive, while the remaining traces are deemed unacceptable (i.e., negative traces). Finally, the process leverages LTL learning to automatically synthesize a weakened safety property based on the examples and the atomic propositions defined by fluents $\mathcal{F}$.

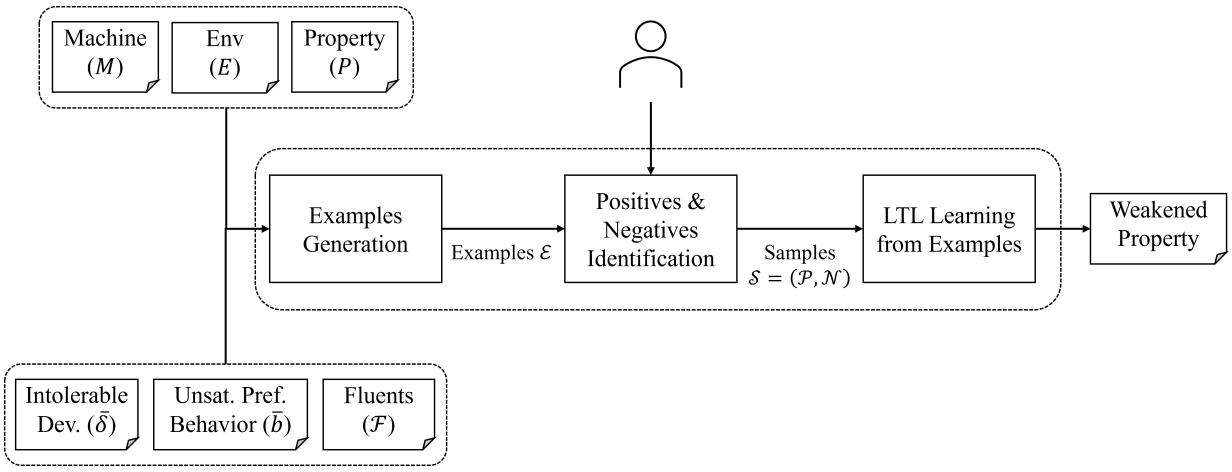


Figure 4.7: Overview of the specification weakening process.

4.5.2 Learning Examples Generation

The first step of our proposed method is to generate example system executions from the unsatisfied preferred behavior. Under the original, restrictive safety property, all concrete executions of an unsatisfied preferred behavior $\bar{b}$ are considered unsafe. In order to weaken the safety property, the user should be able to identify at least one positive trace from the generated examples. However, since the algorithm does not know what that positive trace would look like, it may need to cover all possible concrete executions of the preferred behavior. On the other hand, from a usability perspective, the user may want to start with a small set of example traces and gradually increase this set until a positive trace is found. However, this set should not be too small; otherwise, the user may miss a negative example, which can cause the new property to be too weak to identify the most crucial safety requirement. In sum, the generation algorithm should satisfy the following requirements:

- *upper-bound requirement:* it should be able to cover as many as possible concrete executions (potentially *all* of them) that can be generated from a preferred behavior;
- *lower-bound requirement:* it should cover a *small* but *enough* set of examples to avoid missing a potential unsafe scenario.

For example, in the therapy machine example, the algorithm should at least cover the two representative scenarios: $\langle$X, Up, E, SetEBeam, Enter, B$\rangle$, which is considered safe so that the property can be weakened, and $\langle$X, Up, E, Enter, B$\rangle$, which is unsafe and can prevent the new property being too weak. On the other hand, it may be optional to return the example $\langle$X, Up, E, Enter, SetEBeam, B$\rangle$, which is semantically equal to the first one with only SetEBeam and Enter switching their order. Next, Algorithm 5 describes a process that satisfies the above requirements.

Specifically, it defines a *depth-first search* (DFS) process. The inputs to the algorithm include the machine model M, the deviated environment E', unsatisfied preferred behavior $\bar{b}$, and an integer n controlling the number of additional examples to include after finding a minimal set of required examples. On line 1, it computes the parallel composition

Algorithm 5: Learning examples generation w.r.t. preferred behavior $\bar{b}$

Input : $M, E', \bar{b}$, and n optional number of additional examples
Output: A set of example traces.

1 $T = \langle S, \alpha T, R, s_0 \rangle \leftarrow M||E'||\bar{b}$; `// Model of executing preferred behavior` $\bar{b}$.
2 $covered \leftarrow \langle\rangle$, $extra \leftarrow \langle\rangle$; `// Two sets for storing example traces.`
3 $visited \leftarrow \emptyset$; `// Visited states, a partial function` $S \nrightarrow 2^{\alpha T^*}$
4 $stack \leftarrow \langle (s_0, \epsilon) \rangle$;
5 **while** *stack is not empty* **do**
6 $(s, \sigma) \leftarrow$ `stackPop(`$stack$`);` `// Current state` s `and the trace` σ `to` s.
7 add state s to *visited* set;
8 **for** *event* $a \in \alpha T$ **do**
9 **if** *there is a successor state* s' **then**
10 $\sigma' \leftarrow \sigma \frown \langle (s, a, s') \rangle$; `// The next searching trace for state` s'.
11 **if** s' *is in the current trace* σ **then**
12 add trace σ to *covered* set;
13 **else if** s' *is in the visited set* **then**
`// Add suffixes in` $visited(s')$ `to` σ' `to form new examples.`
14 $\Sigma \leftarrow \{\sigma' \frown \sigma_{suffix} \mid \sigma_{suffix} \in visited(s')\}$;
15 add a random trace in Σ to *covered* set;
16 add the rest in Σ to *extra* set;
17 **else**
18 `stackPush(`$stack, (s', \sigma')$`);`
19 **end**
20 **end**
21 **end**
22 **if** *s is a deadlock state* **then**
23 add trace σ to *covered* set;
24 **end**
25 **end**
26 **return** $covered \frown extra[0..n]$;

$T = M||E'||\bar{b}$, which represents the model of all concrete executions with respect to $\bar{b}$. On line 2, it initializes two example sets, *covered* and *extra*. The *covered* set consists of the minimal set of examples that satisfy the *lower-bound* requirement; and together with the *extra* set, they contain all possible examples given the preferred behavior $\bar{b}$. On line 3, it initializes a partial function $visited : S \nrightarrow 2^{\alpha T^*}$ that tracks all the visited states such that $visited(s)$ stores all the traces from state s that can form a valid example trace. Finally, on line 4, it initializes a stack for tracking the DFS, where a node (s, σ) contains the current

state s and trace σ to search.

In the while loop, for the current searching state s and trace σ (line 6), the algorithm enumerates all events of $M||E'||\bar{b}$ (line 8). For the state s and an event a, if there exists a successor state s' (line 9), then there are three cases:

1. Lines 11 to 12, if s' has been visited by the current trace σ, which forms a cycle, trace σ is added to the example set *covered*.
2. Lines 13 to 16, if s' does not form a cycle but has been visited by other traces before, the *visited* set should contain all valid suffixes that can form a valid example from state s'. Thus, we can generate a set Σ of example traces by appending the suffixes in $visited(s')$ to the next searching trace σ'. The algorithm first randomly picks one and add the trace to the *covered* set. Then, it adds the rest to the *extra* set.
3. Lines 17 to 19, if s' has not been visited in any manner, the algorithm pushes a new search node to the stack.

In addition, on lines 22 to 24, if state s is a deadlock state, i.e., it has no outgoing transitions, then the trace σ is also added to the *covered* set. Also note that when adding a trace to an example set, the algorithm updates the *visited* set accordingly.

The algorithm terminates when the stack is empty, indicating all states in $M||E'||\bar{b}$ have been visited. Then, it returns the examples in the *covered* set with an additional n number of examples in the *extra* set.

Theorem 4.5. *Algorithm 5 will eventually terminate. It will at least return the set of example traces that cover all states in $M||E'||\bar{b}$. When n is big enough, it will return all* finite, acyclic *traces that project to $\bar{b}$.*

Proof. The termination of the algorithm is guaranteed by the nature of the DFS. Specifically, there are two possible cases when encountering a visited state s: (1) The trace σ forms a cycle, and if it can project onto $\bar{b}$, then we add the trace to the result set *covered*. Also, we update the *visited* function such that, for any state s' in σ, we store a new suffix trace that can form a valid example from s'. (2) State s does not form a cycle but has been visited before. Then, the sub-graph whose root is s has been traversed, and $visited(s)$ contains all valid suffixes that can form a valid example from s. Thus, we don't need to search the sub-graph again. We can pick any stored suffix trace in $visited(s)$ to form a new example trace and add it to *covered*. The rest will be added to *extra*.

Thus, the algorithm terminates when all states have been visited; no states will be searched more than once. The *covered* set contains traces that just cover all states in $M||E'||\bar{b}$, and with the *extra* set, they contain all *finite, acyclic* traces that project to $\bar{b}$. □

Given Theorem 4.5, the algorithm provides us the following guarantees that satisfy our lower-bound and upper-bound requirements:

- Upper-bound: Since it will eventually find all traces that can project onto $\bar{b}$ in $M||E'||\bar{b}$, the user should be able to identify at least one positive trace so that the property can be weakened if there exists one.

- Lower-bound: Since the learning target P' is a safety invariant in $\mathbf{G}\phi$, where ϕ is a propositional formula, if there are violations of P' in $M||E'||\bar{b}$, then there should exist a state where ϕ does not hold. Since the algorithm will at least returns a set of example traces (i.e., the *covered* set) that cover all states in $M||E'||\bar{b}$, it is guaranteed to find such a violating trace if it exists.

In practice, the user could start with $n = 0$ to generate a small number of examples that just cover all states in $M||E'||\bar{b}$. The goal is to identify at least one positive trace from the returned examples. If there are no such traces, the user could gradually increase n to generate more example traces until finding one positive trace.

4.5.3 Weakening by Learning from Examples

After generating the learning examples, the user is responsible for classifying them into positive and negative sets and building the learning sample $\mathcal{S} = (\mathcal{P}, \mathcal{N})$. Then, we leverage LTL learning techniques to synthesize a weakened formula. Specifically, we utilize ATLAS [41], a general constrained LTL learning tool based on Alloy$^{\text{Max}}$ [42]. The reason is that other LTL learning tools, such as [78, 86], do not have enough express power to encode our weakening constraints, whereas ATLAS allows us to define syntactic constraints in first-order logic (FOL) and specify custom optimization goals for the learned formulas.

In ATLAS, the constraints are defined over the syntax of the learned formula. For an LTL formula ϕ, its syntax is modeled as a tree and is represented by a tuple $\mathsf{syntax}(\phi) = \langle \mathcal{L}, \mathcal{R}, root \rangle$ where $\mathcal{L}, \mathcal{R} \subseteq N \times N$ are the left and right child relations, respectively; N is the set of nodes in the syntax tree, and $root$ is the root node of the tree. In particular, $N = \bigcup_{op \in \mathcal{N}} N_{op}$ where $\mathcal{N} = \{\mathbf{G}, \mathbf{F}, \mathbf{U}, \mathbf{X}, \wedge, \vee, \Rightarrow, \neg, AP\}$ and N_{op} is the set of nodes of a particular operator type or atomic propositions. Then, we can define constraints in FOL over $\mathsf{syntax}(\phi)$. Moreover, other than the typical logic constructs in FOL and set theory, ATLAS provides an extended list of operators and functions for improved expressiveness. Here, we briefly introduce the operators and functions used in our weakening encoding:

- $l(n)$ returns the left child of a node n.
- $r(n)$ returns the right child of a node n.
- $\mathsf{desc}(n)$ returns all the descendent nodes of a node n.
- $\mathsf{subNodes}(n)$ returns all the descendent nodes of a node n, including n itself.
- $\mathsf{subTree}(n)$ returns the sub-tree from a node n.
- $min(s)$ that minimizes the number of elements in a set s.

Then, to solve our specification weakening problem $\mathfrak{W}_{\mathcal{F}}(Rb, \bar{b}, \mathcal{F})$, given the sample $\mathcal{S} = (\mathcal{P}, \mathcal{N})$ and the original property P, we define the following constraints in ATLAS.

Encoding safety invariant pattern $\mathbf{G}(\phi \Rightarrow \psi)$. The following constraints let ATLAS return formulas satisfying the pattern of $\mathbf{G}(\phi \Rightarrow \psi)$ where ϕ and ψ are propositional formulas, ϕ is in CNF, and ψ is in DNF.

$$n_{\mathbf{G}} \in N_{\mathbf{G}} \wedge n_{\Rightarrow} \in N_{\Rightarrow} \tag{4.1}$$
$$root = n_{\mathbf{G}} \wedge l(root) = n_{\Rightarrow} \tag{4.2}$$
$$\forall n \in \mathsf{desc}(n_{\Rightarrow}) : n \in N_{\{\wedge,\vee,\neg,AP\}} \tag{4.3}$$
$$\forall n \in \mathsf{desc}(n_{\Rightarrow}) : n \in N_{\neg} \Rightarrow l(n) \in N_{AP} \tag{4.4}$$
$$\forall n \in \mathsf{subNodes}\big(l(n_{\Rightarrow})\big) \cap N_{\vee} : \mathsf{desc}(n) \cap N_{\wedge} = \emptyset \tag{4.5}$$
$$\forall n \in \mathsf{subNodes}\big(r(n_{\Rightarrow})\big) \cap N_{\wedge} : \mathsf{desc}(n) \cap N_{\vee} = \emptyset \tag{4.6}$$

Line (4.1) instantiates a **G**-operator node and a $\Rightarrow$-operator node, and line (4.2) defines that the *root* node is the **G**-node and its child is the $\Rightarrow$-node. Line (4.3) defines that all the descendant nodes of $n_{\Rightarrow}$ are in $\wedge, \vee, \neg$, and AP as formulas ϕ and ψ in our invariant pattern should be propositional formulas. Line (4.4) defines that ϕ and ψ are in *negation normal form* (NNF). Finally, line (4.5) defines ϕ is in CNF, i.e., for any $\vee$-node in ϕ, no $\wedge$-nodes should be its descendants; and line (4.6) defines ψ is in DNF, i.e., for any $\wedge$-node in ψ, no $\vee$-nodes should be its descendants.

Encoding original property P. Then, we define the constraints for the original safety property $P = \mathbf{G}(\phi \Rightarrow \psi)$, which should be weakened. Let $\mathsf{syntax}(\phi) = \langle \mathcal{L}_{\phi}, \mathcal{R}_{\phi}, root_{\phi} \rangle$ be the syntax tree of formula ϕ, and $\mathsf{syntax}(\psi) = \langle \mathcal{L}_{\psi}, \mathcal{R}_{\psi}, root_{\psi} \rangle$ be the syntax tree of ψ. Then, we have:

$$l(n_{\Rightarrow}) = root_{\phi} \vee \Big(l(n_{\Rightarrow}) \in N_{\wedge} \wedge l\big(l(n_{\Rightarrow})\big) = root_{\phi} \Big) \tag{4.7}$$
$$\mathcal{L}_{\phi} \cup \mathcal{R}_{\phi} \subseteq \mathsf{subTree}\big(l(n_{\Rightarrow})\big) \tag{4.8}$$
$$r(n_{\Rightarrow}) = root_{\psi} \vee \Big(r(n_{\Rightarrow}) \in N_{\vee} \wedge l\big(r(n_{\Rightarrow})\big) = root_{\psi} \Big) \tag{4.9}$$
$$\mathcal{L}_{\psi} \cup \mathcal{R}_{\psi} \subseteq \mathsf{subTree}\big(r(n_{\Rightarrow})\big) \tag{4.10}$$

Lines (4.7) and (4.8) define that the left sub-tree of the $\Rightarrow$-node is either the original ϕ or $\phi \wedge \bigwedge \bigvee [\neg] x$. Similarly, lines (4.9) and (4.10) define that the right sub-tree of the $\Rightarrow$-node is either the original ψ or $\psi \bigvee \bigwedge [\neg] x$. Therefore, from lines (4.1) to (4.10), we have specified the constraints for finding a solution satisfying our "gradual" weakening pattern as described in Section 4.4.2.

For instance, the safety property $\mathbf{G}(Xray \Rightarrow InPlace)$ will be converted to the following constraints:

$$l(n_{\Rightarrow}) = n_{Xray} \vee \Big(l(n_{\Rightarrow}) \in N_{\wedge} \wedge l\big(l(n_{\Rightarrow})\big) = n_{Xray} \Big)$$
$$r(n_{\Rightarrow}) = n_{InPlace} \vee \Big(r(n_{\Rightarrow}) \in N_{\vee} \wedge l\big(r(n_{\Rightarrow})\big) = n_{InPlace} \Big)$$

where $n_{Xray}, n_{InPlace} \in N_{AP}$ are the nodes representing the atomic propositions in the formula.

Encoding minimization goal and additional constraints. Then, we add the optimization goal

$$min(\mathcal{L} \cup \mathcal{R}), \tag{4.11}$$

which minimizes the formula size of the final solution by minimizing the number of edges in the syntax tree. In addition, we add syntactic constraints to prevent some unhelpful expressions such as tautologies (e.g., $p \vee \neg p$ and $p \Rightarrow p$) and idempotent expressions (e.g., $p \wedge p$ and $p \vee p$).

Converting to Alloy$^{\text{Max}}$ **expressions** Finally, we convert these constraints into Alloy$^{\text{Max}}$ expressions. Consider the safety property $\mathbf{G}(Xray \Rightarrow InPlace)$ in the therapy machine example. We have:

```
1  one sig Xray, InPlace extends Literal {}
2  one sig G0 extends G {}
3  one sig Imply0 extends Imply {}
4  fact {
5      root = G0 and root.l = Imply0
6      all n: desc[Imply0] | n in (Literal + And + Or + Neg)
7      // & stands for intersection.
8      all n: desc[Imply0] & Neg | n.l in Literal
9      // no stands for empty set
10     all n: subNodes[Imply0.l] & Or | no desc[n] & And
11     all n: subNodes[Imply0.r] & And | no desc[n] & Or
12 }
13 fact {
14     Imply0.l = Xray or (Imply0.l in And and Imply0.l.l = Xray)
15     Imply0.r = InPlace or (Imply0.r in Or and Imply0.r.l = InPlace)
16 }
17 fact {
18     minsome l + r
19     all n: And + Or + Imply | n.l != n.r
20     all n: Or | (n.l in Neg implies n.l.l != n.r) and
21         (n.r in Neg implies n.r.l != n.l)
22     all imply: Imply |
23         no ((imply.l - And) + (subNodes[imply.l] & And).(l + r)) &
24             ((imply.r - Or) + (subNodes[imply.r] & Or).(l + r))
25 }
```

Lines 1 to 3 declare the nodes used in the original formula. Lines 4 to 12 correspond to the constraints from equation lines (4.2) to (4.6), which specify the invariant pattern. Lines 13 to 16 correspond to the constraints from equation lines (4.7) to (4.10) that encode the original formula. Finally, line 18 corresponds to the minimization goal, and lines 19 to 24 are the additional syntactic constraints to prevent unhelpful expressions including tautologies and idempotent expressions. With the above constraints, ATLAS can synthesize solutions to the specification weakening problem $\mathfrak{W}_{\mathcal{F}}(Rb, \bar{b}, \mathcal{F})$.

4.6 Evaluation

4.6.1 Research Questions

The evaluation of our weakening approach focuses on two research questions:

- **RQ1 (Applicability)**: Is our specification weakening approach applicable to software systems from different domains? Is it able to find weakened safety properties that allow retaining preferred behaviors that cannot be retained with only robustification-by-control?
- **RQ2 (Scalability)**: How much additional performance overhead would the weakening process introduce compared to robustification-by-control? How well does it scale, particularly the LTL learning process?

To answer RQ1, we evaluated our approach on the five case studies that we used in our robustness computation and robustification-by-control evaluation. We define a strong FLTL_f safety property in $\mathbf{G}(\phi \Rightarrow \psi)$ for each problem. Each property describes a similar safety requirement to the one we used in previous evaluations. For each problem, we considered a preferred behavior that cannot be satisfied when performing only robustification-by-control with respect to the strong safety property. We then applied our weakening approach to show that it can find a weaker safety property; and with this new property, re-running the robustification-by-control process can find a redesign that retains the preferred behavior.

To answer RQ2, we first compared the time between robustification-by-control and specification weakening in our case studies. Specifically, a robustification process with both robustification-by-control and specification weakening involves at least four steps: (1) the user first runs robustification-by-control and finds that certain preferred behavior cannot be satisfied; (2) they run the example generation and analyze the returned concrete scenarios; (3) the tool learns one or more candidate solutions from the examples; (4) the user reruns the robustification-by-control process with respect to the new property. Therefore, for each problem, we recorded the computation time of the first robustification-by-control process, the example generation process, the LTL learning process, and the second robustification-by-control process. However, we do not consider the time for the user to identify the positive examples.

Moreover, the complexity of our example generation process is linear to the size of the system model, whereas the LTL learning process requires solving a MaxSAT problem [87], which is NP-hard. Thus, we also built a set of random weakening problems to evaluate the scalability of ATLAS in solving this particular type of LTL learning problem to understand how it may affect the scalability of our weakening approach. However, a more general and comprehensive performance analysis of ATLAS can be found in [41].

4.6.2 Implementation

We implemented the weakening approach in our tool Fortis. It uses ATLAS to perform the constrained LTL learning process. The source code of the implementation is available on GitHub at `https://github.com/cmu-soda/fortis-core`.

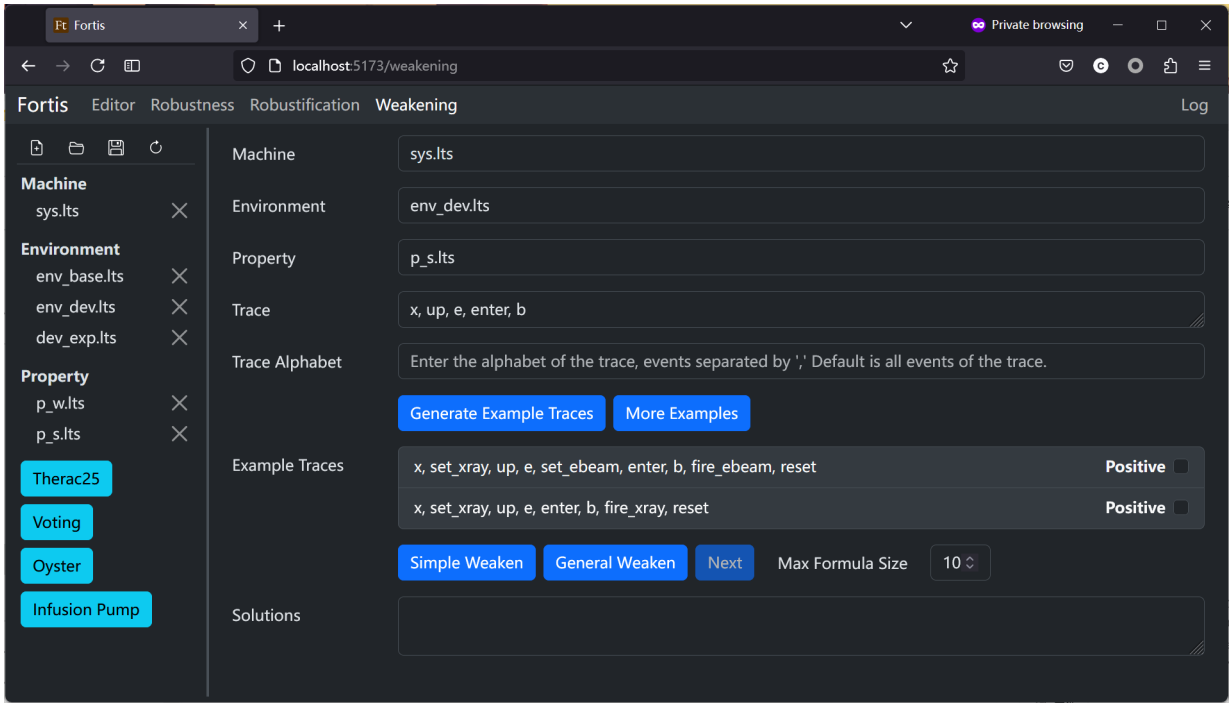


Figure 4.8: Screenshot of Fortis for robustification-by-weakening

Figure 4.8 shows the web interface of Fortis for weakening. Similar to robustness analysis and robustification-by-control, a user specifies all parameters of a weakening problem on the right panel, including the model specifications, the safety property to weaken, and the unsatisfied preferred behavior. Then, by clicking Generate Example Traces, the Fortis back-end will be invoked to generate a list of example traces. The user can click the Positive checkbox at the end of each trace to mark it as a positive example; otherwise, the trace is considered negative. Finally, the user clicks the General Weaken button to let Fortis synthesize a weaker safety property. The user can also click the Next button to enumerate the solutions.

We used the web interface to conduct the experiments for RQ1. Then, we used a program to generate random weakening problems to evaluate the scalability by directly invoking the Fortis back-end. All experiments were done on a Windows machine with an Intel i9-12900H processor and 32GB memory.

4.6.3 Case Studies

We first present the experiment settings for all our case study problems and their results. We did not apply weakening to the network protocol problem because both the naive protocol and ABP satisfy the safety property under the deviated environment with message loss. In addition, its safety property, that the input and output events should alternate, cannot be expressed in the LTL pattern of $\mathbf{G}(\phi \Rightarrow \psi)$. It is challenging to express this property in LTL whereas it is straightforward in process algebra (e.g., by using FSP [48]).

Radiation therapy machine. Consider the robustification of the radiation therapy machine. To build the FLTL$_f$ property, we defined the following set of fluents:

- $Xray = \langle\{\mathsf{SetXray}\}, \{\mathsf{SetEBeam},\ \mathsf{Reset}\}, false\rangle$
- $EBeam = \langle\{\mathsf{SetEBeam}\}, \{\mathsf{SetXray},\ \mathsf{Reset}\}, false\rangle$
- $InPlace = \langle\{\mathsf{X}\}, \{\mathsf{E}\}, true\rangle$
- $Fired = \langle\{\mathsf{FireXray},\ \mathsf{FireEBeam}\}, \{\mathsf{Reset}\}, false\rangle$

Then, we defined a strong safety property that *the spreader must be in place when the beam is in X-ray*:

$$P_s = \mathbf{G}(Xray \Rightarrow InPlace)$$

For the robustification-by-control process, we defined the progress property that the machine should eventually be able to fire the beam. Then, we assigned *NoCost* for observing all the events of the machine and *NoCost* for controlling FireXray, FireEBeam, SetXray, and SetEBeam. We assigned a *Cheap* cost to control events X, E, Enter, Up, and B. More details can be found in the problem setting described in Section 3.7.

We considered the preferred behavior $\bar{b} = \langle\mathsf{X},\ \mathsf{Up},\ \mathsf{E},\ \mathsf{Enter},\ \mathsf{B}\rangle$, which cannot be satisfied under the strong safety property P_s when performing only robustification-by-control. Therefore, we invoked our weakening process against this preferred behavior, and the example generation process returned two example traces:

- $e_1 : \langle\mathsf{X},\ \mathsf{SetXray},\ \mathsf{Up},\ \mathsf{E},\ \mathsf{SetEBeam},\ \mathsf{Enter},\ \mathsf{B},\ \mathsf{FireEBeam},\ \mathsf{Reset}\rangle$
- $e_2 : \langle\mathsf{X},\ \mathsf{SetXray},\ \mathsf{Up},\ \mathsf{E},\ \mathsf{Enter},\ \mathsf{B},\ \mathsf{FireXray},\ \mathsf{Reset}\rangle$

We marked e_1 as positive because the beam is correctly fired in Electron beam mode, while we marked e_2 as negative because the beam is fired in X-ray with the spreader out of place. The LTL learning process returned

$$\mathbf{G}(Xray \wedge Fired \Rightarrow InPlace)$$

as the first solution, which indicates a weaker safety requirement that *the spreader must be in place when the beam is fired in X-ray*. As we have shown in previous sections, re-running the robustification-by-control process found a redesign that retains the preferred behavior $\bar{b}$.

Voting machine. Consider the robustification of the voting machine example. We defined the following set of fluents:

- $InVoting = \langle\{\mathsf{password}\}, \{\mathsf{complete},\ \mathsf{reset}\}, false\rangle$ [2]
- $VoterIn = \langle\{\mathsf{v.enter}\}, \{\mathsf{v.exit}\}, false\rangle$
- $OfficialIn = \langle\{\mathsf{eo.enter}\}, \{\mathsf{eo.exit}\}, false\rangle$
- $CriticalOp = \langle\{\mathsf{select},\ \mathsf{vote},\ \mathsf{confirm}\}, *, false\rangle$

where $*$ stands for all the other events in the alphabet of the system except those in the initiating set of $CriticalOp$. Specifically, $InVoting$ indicates whether a voter has entered their password and started a voting process, and $CriticalOp$ indicates whether a critical

[2] We introduce a helper event complete to track the voting process.

operation (including select, vote, and confirm) has been performed by any user. To identify potential voter fraud and ensure vote integrity, we defined a strong safety property that *the election official should not enter the voting booth when the voting is in progress*:

$$P_s = \mathbf{G}(OfficialIn \Rightarrow \neg InVoting)$$

For the robustification-by-control process, we defined the progress property that the voter should eventually be able to confirm their vote. We assigned *NoCost* for observing all the internal events of the machine and a *Cheap* cost to control them. We assigned a *Moderate* cost for observing {v, eo}.enter and {v, eo}.exit but *Costly* for controlling them. More details are described in Section 3.7.

We considered a preferred behavior $\bar{b} = \langle$v.enter, password, select, vote, v.exit, eo.enter, reset, eo.exit$\rangle$, which describes a scenario where the voter has voted for a candidate and leaves the voting booth before confirming, while the election official enters the booth and resets the machine to close this uncompleted session. However, this preferred behavior cannot be satisfied as the strong property forbids the official from entering the booth before a voting session is complete. We then invoked the weakening process, and the example generation returned one example trace:

$\langle$v.enter, password, select, vote, v.exit, eo.enter, back, back, reset, eo.exit$\rangle$

We marked this example as positive because the election official only resets the machine and does not change the vote made by the voter. The LTL learning process returned

$$\mathbf{G}(OfficialIn \wedge VoterIn \Rightarrow \neg InVoting)$$

as the first solution. However, since $OfficialIn \wedge VoterIn$ is always false given our system design, this property always evaluates to true, which is unsatisfactory. We then had the learner enumerate the next solutions, it returned

$$\mathbf{G}(OfficialIn \wedge CriticalOp \Rightarrow \neg InVoting) \text{ and}$$

$$\mathbf{G}(OfficialIn \Rightarrow \neg InVoting \vee \neg CriticalOp)$$

These two formulas are semantically equal, but the latter one is more readable. It represents a weaker safety requirement: *the official can enter the booth while the voting is in progress as long as they do not perform those critical actions*. By re-running the robustification-by-control process, we found a redesign that can retain the preferred behavior $\bar{b}$ by disabling those critical operations when detecting an election official entering the booth and enabling them after a reset.

Oyster transportation fare system. Consider the robustification of the Oyster fare collection system. We defined the following set of fluents:

- $UseCard = \langle$\{rcv.card.gin\}, \{rcv.card.fin\}, $false\rangle$
- $UseOyster = \langle$\{rcv.oyster.gin\}, \{rcv.oyster.fin\}, $false\rangle$
- $CardOut = \langle$\{rcv.card.fin\}, \{snd.card, snd.oyster\}, $false\rangle$

- $OysterOut = \langle\{\mathsf{rcv.oyster.fin}\}, \{\mathsf{snd.card}, \mathsf{snd.oyster}\}, false\rangle$
- $NoOysterBal = \langle\{\mathsf{no\_oyster\_bal}\}, \{\mathsf{rld.oyster}\}, false\rangle$

where $UseCard$ and $UseOyster$ indicate that the user uses a credit card or an Oyster card to enter the gate, respectively, $CardOut$ and $OysterOut$ indicate that the user uses a certain method to exit the gate, and $NoOysterBal$ indicates that the user's Oyster card balance is empty. To ensure no card collision in the system, we defined a strong safety property that *the user should use the same payment method in the same journey*:

$$P_s = \mathbf{G}(CardOut \Rightarrow \neg UseOyster) \wedge \mathbf{G}(OysterOut \Rightarrow \neg UseCard)$$

For the robustification-by-control process, we defined the progress property that the user should be able to exit the gate with a payment method. We assigned *NoCost* to observe and control all the machine-initiated events (e.g., acknowledge a payment request). We also assigned *NoCost* to observe the user-initiated events (e.g., send a payment request) but *Costly* to control them. More details are described in Section 3.7.

We considered a preferred behavior $\bar{b}$ that describes a relaxation of the payment requirement—a user enters the gate with their Oyster transportation card and leaves the gate with a credit card when their Oyster card balance is empty. This may be a result of requirement evolution, as allowing a user to leave with their credit card when their Oyster card balance is empty can improve the usability of the system. However, this preferred behavior cannot be satisfied under the strong safety property P_s. We then invoked the weakening process, and the example generation returned one example trace, which is the same as the above-described scenario. Thus, we marked it as positive, and the LTL learning process returned

$$\mathbf{G}(CardOut \Rightarrow \neg UseOyster \vee NoOysterBal) \wedge \mathbf{G}(OysterOut \Rightarrow \neg UseCard)$$

as the first solution. It is a weaker version of the original safety property by adding another accepting condition (i.e., $NoOyster$) to $CardOut$ (i.e., leaving with a credit card). Finally, re-running the robustification-by-control process found a redesign satisfying the preferred behavior $\bar{b}$.

Infusion pump. Finally, consider the robustification of the infusion pump machine. We defined the following set of fluents:

- $Dispensing = \langle\{\mathsf{line.dispense}\}, *, false\rangle$
- $PowerFailed = \langle\{\mathsf{power\_failure}\}, \{\mathsf{battery\_charge}\}, false\rangle$
- $Plugged = \langle\{\mathsf{plug\_in}\}, \{\mathsf{unplug}\}, false\rangle$

where $Dispensing$ indicates whether the machine is dispensing medicine to a patient, $PowerFailed$ indicates whether the machine encounters a power failure, and $Plugged$ indicates whether the power cable is plugged in. Specifically, $*$ stands for all the other events in the alphabet of the system except those in the initiating set of $Dispensing$. To ensure safe dispensing, we defined a strong safety property that *the power cable should be plugged in when the machine is dispensing medicine*:

$$P_s = \mathbf{G}(Dispensing \Rightarrow Plugged)$$

Table 4.1: The computation time (in seconds) for robustification-by-control and specification weakening in case studies.

Problem	1st-Robustify	Example Gen.	LTL Learn.	Num of Tries	2nd-Robustify
Therapy	0.059	0.023	0.131	1	0.084
Voting	0.116	0.038	0.221	3	0.176
Oyster	0.154	0.129	0.385	1	0.277
Pump	0.408	0.179	0.993	3	0.691

Then, we used the same cost assignment of events for robustification-by-control as described in Section 3.7.

The above safety property may be considered too restrictive, as it does not allow the machine to take advantage of its battery mode to continue dispensing when the power cable is unplugged, as long as the battery does not run out. This can be demonstrated through a preferred behavior $\bar{b}$—the power cable is unplugged during a dispensation, and the dispensation continues in battery mode; then, the power cable is reconnected before a power failure occurs; finally, the machine correctly completes the dispensation. However, this preferred behavior cannot be satisfied under the original safety property. Running the weakening process, the example generation process returned one example trace, which describes the above scenario. We marked the trace as positive, and the LTL learning process returned

$$\mathbf{G}(Dispensing \wedge PowerFailed \Rightarrow Plugged)$$

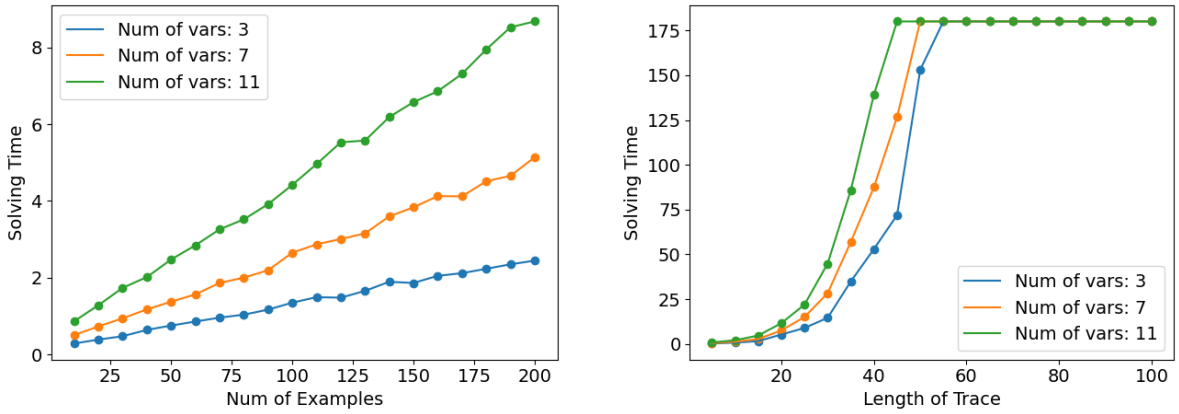
as the first solution. This formula is, in fact, a valid weakening because a power failure cannot occur when the power cable is plugged in, so the machine should ensure the antecedent ($Dispensing \wedge PowerFailed$) never happens. However, it may be difficult for a human developer to understand. We then searched for the next solutions, and the third solution returned by the tool was more preferable:

$$\mathbf{G}(Dispensing \Rightarrow Plugged \vee \neg PowerFailed)$$

Finally, re-running the robustification-by-control process found a redesign satisfying the preferred behavior $\bar{b}$.

4.6.4 Experimental Results

Table 4.1 shows the computation time for each step during the robustification of a case study problem. From the table, the weakening process, including example generation and LTL learning, does not introduce significant performance overhead compared to the robustification-by-control process. All the computations were completed in less than a second. There are situations where the learning process does not return a satisfactory weakening as the first solution (often due to a *true* antecedent or lack of formula readability) and requires enumeration of solutions. However, it does not require a large number of enumerations and does not significantly impact the total learning time either.



(a) Increasing number of examples.

(b) Increasing length of traces.

Figure 4.9: Evaluation results of Fortis synthesizing weakened safety properties using ATLAS. All problems have a 3-minutes timeout.

Different from the robustification-by-control process, whose complexity grows exponentially with respect to the model size of a system, the complexity of example generation is linear to the model size, and the complexity of LTL learning is associated with the size of fluents and examples [78, 41]. Thus, scaling up the case studies does not necessarily scale up the problem size of weakening, and we hypothesize that the LTL learning process would be the bottleneck. To further evaluate the scalability of our weakening approach and understand its limitation, we generate 234 random weakening problems following two commonly observed weakening patterns:

$$\textit{weakening } \mathbf{G}(a \Rightarrow b) \textit{ to } \mathbf{G}(a \wedge c \Rightarrow b), \textit{ and weakening } \mathbf{G}(a \Rightarrow b) \textit{ to } \mathbf{G}(a \Rightarrow b \vee c).$$

We scaled these weakening problems in three dimensions: the number of atomic propositions (i.e., the number of fluents), the number of total examples, and the length of examples. We set a 3-minute timeout for all problems and used OpenWBO [88] as the back-end MaxSAT solver. Figure 4.9 shows the experiment results.

Figure 4.9a shows the results of how the solving time grows with the increasing number of examples. Specifically, each trace has a length of 5, and the results are grouped into three lines based on the number of atomic propositions (3, 7, and 11). From the figure, the solving time grows almost linearly with the increasing number of examples, and increasing the number of atomic propositions does not significantly affect the solving time. However, the slope (i.e., the rate of increase in solving time) is steeper with more atomic propositions. The maximum solving time is around 9 seconds given our benchmark problems.

Figure 4.9b shows the results of how the solving time grows with the increasing length of traces. Specifically, each problem has 5 positive traces and 5 negative traces, and the results are also grouped by the number of atomic propositions. From the figure, although the number of propositions still does not significantly affect the performance, the solving

time grows extremely fast with the increasing number of trace length. At around a trace length of 60, all the problems in our benchmark timed out. The length of trace has a larger impact on the scalability of the LTL learning process.

These results imply that, when facing a more complex system, a functional requirement that requires a longer preferred behavior description has a greater impact on the performance overhead when applying our specification weakening technique.

4.7 Summary

This chapter presents an approach to robustify a machine design through specification weakening, alongside our robustification-by-control method. The idea is that when the given safety property is too restrictive to satisfy during robustification, leading to the loss of certain critical functionality, the approach can weaken the safety property to allow more feasible robustification solutions. Specifically, we consider a situation where a preferred behavior cannot be satisfied by any solution during a robustification-by-control process. In such cases, the developer can use our weakening approach to synthesize a weaker safety property, so that the preferred behavior can be satisfied under a weakened property during robustification.

We focus on a particular class of safety properties, which are defined in FLTL_f and follow the pattern of $\mathbf{G}(\phi \Rightarrow \psi)$, where ϕ and ψ are propositional formulas. We believe this pattern can already capture a large and interesting set of safety invariants, as we demonstrated in our case studies. However, we do observe that certain safety properties, which can be easily defined in LTS, cannot be presented in this pattern, such as the alternating inputs and outputs property for the network protocol example.

We leverage ATLAS, a tool for constrained LTL learning, to synthesize weakened safety properties. Specifically, we utilize its syntactic constraint capabilities to learn formulas that satisfy a syntactic weakening pattern ($\mathbf{G}((\phi \wedge \bigwedge \bigvee [\neg] p_i) \Rightarrow (\psi \vee \bigvee \bigwedge [\neg] p_j))$). However, this pattern only provides a syntactic approximation of minimal weakening and does not guarantee that the weakened property is semantically minimally weakened. Additionally, our evaluation shows that although ATLAS is a state-of-the-art tool that allows us to define the constraints of our weakening pattern and effectively solve the weakening problems in our case studies, it suffers from increased solving times with the growing length of example traces, as demonstrated in our scalability evaluation.

Chapter 5

Related Work

5.1 Robustness Definition and Measurement

According to the survey by Shahrokni et al. [17] and Laranjeiro et al. [18], most of the prior work on robustness for conventional software systems focuses on testing. Popular methods such as fault injection [21, 22], model-based testing [23], and fuzzing [24, 25] 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. An exemplar work is by Koopman et al. [7]. 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 [18] 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 [89, 90, 91, 92]. One common characteristic of these prior definitions is that they are all *quantitative* in nature, in that they all define some kind of function to measure the *distance* between traces and system behavior. For example, Bloem et al. [92] propose a notion of robustness that defines, for a robust system, that the ratio of the degree of incorrect machine outputs to 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. [89] 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. [90, 91] adopt the

notion of *Lipschitz 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 metric 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) [93]; similarly, Nayak et al. propose *robust computation tree logic* (rCTL) [94]. Both 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 follows: 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 [95]. This concept has been adopted in a number of engineering domains such as aviation, robotics, and manufacturing, but, as far as we know, it has not been rigorously defined in the context of software engineering. Therefore, our notion of robustness can be considered 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. [28] 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. [96] argue that a robust program should be able to stop the propagation of failures; and according to information theory, they propose that robustness might be captured as entropy loss in the code region succeeding the code region where the faults occur. The higher the entropy loss, the higher the likelihood that the propagation of the failure could be prevented.

5.2 Design Robustification

Robustification by control. 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 [92, 89, 90, 97, 98]. 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 robustification-by-control method, namely supervisory control synthesis [39], 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 [19] and run-time assurance [99, 20, 100] to dynamically enforce system requirements. For instance, in the self-adaptive systems community, the MAPE-K adaptation framework [19, 101], including tools like Rainbow [102] and MORPH [103], also employs a monitoring and actuation control loop to ensure that a system continues to satisfy certain properties against environmental uncertainties at run time. Traditionally, these run-time approaches assume fixed sensing and actuating capabilities, which then limit their adaptation capability. By comparison, our work focuses on robustifying a machine at *design time*, providing developers with more flexibility to extend the sensing and actuating abilities by introducing additional observable and controllable events.

Moreover, recent work in self-adaptive systems has also considered adaptive monitoring, which allows for dynamically changing a system's sensing ability to, for example, improve resource utilization, monitor only as needed, or adapt to changes in operational context (e.g., requirements, objectives, and architecture) [104, 105, 106, 107]. However, these approaches still assume a pre-defined, total set of monitors and compute a subset to enable based on the operational context, or they consider deploying new sensors for known types of events during adaptation. In contrast, design time robustification offers greater flexibility to address unknown events such that monitoring or controlling them may demand the development and deployment of new physical or software components, which is often infeasible at run time. In other words, run-time adaptation capabilities are typically limited by the developers' (*known*) knowledge at the time of design and deployment, whereas at design time, developers have more resources to create design solutions for previously unknown uncertainties.

Our robustification-by-control 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 where, given a system S and a property P such that $S \not\models P$, a new system S' is generated such that $S' \models P$. Buccafurri et al. [108] propose a formal definition of model repair for Computation Tree Logic (CTL) and present an approach to finding repairs using abductive reasoning. Similarly, Menezes et al. [109], Chatzieleftheriou et al. [110], and Ding et al. [111] 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 robustification-by-control approach can be considered 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 work does not distinguish between the machine and the environment of a system. Moreover, existing works do not consider the cost of a repair or consider the cost based only on the syntactic changes of the model, e.g., adding or removing states or transitions. In contrast, our approach considers multiple semantic-based quality metrics of repairs, such as 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 closest work is presented by Tun et al. [65], which also uses controller synthesis to generate new designs that satisfy a desired property. Although their work does not explicitly aim at improving the robustness of a software system, they have a similar goal of revising a machine M to fulfill a security requirement P in an environment E where users might deviate from the expected behavior, causing security violations. Our robustification-by-control 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 a machine to generate new designs that satisfy a 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 exploration method for us to search for 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 specification weakening The idea of weakening in the context of requirements engineering [38, 50] 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 [112, 51, 52].

In the work by Alrajeh et al. [50], their focus is on adapting *system requirements* to address environmental deviations during the requirement elicitation phase. The authors propose an approach that utilizes a learning technique to automatically adapt system requirements specified in a goal model [113] with Metric Temporal Logic (MTL) to changes in the environment. However, their work focuses purely on requirements and ignores the system realization (i.e., system design or implementation), whereas we consider resolving conflicts in requirements by specification weakening during the process of making a machine design more robust.

Chu et al. [52] explore the weakening idea in the context of CPS with Signal Temporal Logic (STL). They focus on the problem of feature interactions in a system at run time, where conflicting control actions may be generated as two system features both try to lead the system to satisfy their specific requirements. To resolve the conflicts, 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 satisfy. Then, at run time, when a conflict occurs, they try to minimally weaken the conflicting requirements by finding new time bounds such that they can synthesize a new control action that satisfies the weakened specifications. Compared to our weakening work, one significant difference is that they weaken the specifications by changing the time bounds of temporal operators but keeping the original propositions, whereas our approach achieves weakening by changing the propositions.

Moreover, Buckworth et al. [51] present a run-time adaptation technique involving the weakening of LTL. In their work, they consider a scenario where, at run time, a self-adaptive

system cannot continue providing its guarantees due to the changes in the environment. Then, they learn a weakened specification that aligns with the current environmental conditions and synthesize a new controller based on this adjusted specification. Our work shares similarity with this method in that we both consider weakening the specification in the presence of environmental deviations and synthesizing a controller given the weakened specification. However, the goal of weakening is different. They employ weakening at run time so that the system can continue operating under the new environment and controller but do not explicitly consider how the specification is weakened. By comparison, we distinguish between safety and functionality and focus on the weakening of safety requirements only so that the system can retain more functionality during robustification.

D'Ippolito et al. [114] also adopt the weakening concept and propose 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. However, in the context of robustification, they do not specifically consider the relationship between switchable machines (e.g., M' and M'') with respect to quality metrics (e.g., modification cost or behavioral similarity). Moreover, they focus on the theoretical foundations of this multi-tier adaptive framework, assuming that the different levels of requirements are provided, whereas we focus on how to derive a weaker specification given a requirement conflict.

Our approach to specification weakening also shares similarity with *graceful degradation*—where a system displays degraded behavior in response to the changes in the environment [115]. We both study the problem of how a system specification can be gradually and minimally weakened when it cannot be fulfilled given the deviated environment. However, the problem context and goals are slightly different. Typically, graceful degradation considers how the functionality of a system should degrade at run time, in the presence of environmental changes, to guarantee the satisfaction of critical safety or security requirements. In contrast, we consider weakening an unnecessarily restrictive safety requirement to fulfill certain functionality in the presence of environmental deviations at design time. In other words, our goal of weakening is to retain system behaviors by sacrificing safety. On the other hand, our robustification-by-control approach allows for sacrificing system functionality (i.e., preferred behaviors) to ensure a fixed safety property, which is closer to the conventional concept of graceful degradation.

Chapter 6

Discussion and Conclusion

6.1 Robustness Analysis

We present a formal notion of behavioral robustness for software based on labeled transition systems (LTS). Given a software machine, our notion defines robustness as a set of environmental traces that are not defined in the original, expected environment and that would not cause a violation of a desired safety property. We leverage the computation of the weakest assumption to compute robustness, and then robustness is the set of traces that are in the weakest assumption of a machine with respect to a safety property but are not in the expected environment. In general, robustness may contain an infinite number of traces. Then, we present a way to partition it into a finite set of equivalence classes, each of which contains traces that describe the same type of deviation, and sample representative traces from those classes. Finally, we use a deviation model to generate explanations that describe how the environment may deviate from its expected behavior in a particular way.

Representation and explanation. One limitation of our approach is that our current method of defining equivalence classes for robustness may sometimes result in a classification that is too fine-grained. For example, in the ABP case study (Section 2.6), traces like $\langle send[0], send[0]\rangle$ and $\langle send[1], send[1]\rangle$ refer to the same type of fault (i.e., message loss during sending) and could be grouped into the same class, which, however, are treated as two equivalence classes. Future work could explore different strategies for generating equivalence classes, leveraging abstraction-based methods to produce higher-level representations of deviations (e.g., $\langle send[x], send[x]\rangle$ for some event parameter x).

Another limitation is how we pick the representative traces. Currently, for each equivalence class, we choose the shortest normative prefix plus the first deviated action as the representative trace of this class. However, it may "mistakenly" represent an intolerable deviation that has the same prefix. For instance, in the radiation therapy machine, the old (unrobust) design has a class with a representative trace $\langle \mathsf{X},\ \mathsf{Up}\rangle$. Although the machine is robust against deviations such as $\langle \mathsf{X},\ \mathsf{Up},\ \mathsf{E},\ \mathsf{Enter}\rangle$, it is not robust against the deviation $\langle \mathsf{X},\ \mathsf{Up},\ \mathsf{E},\ \mathsf{Enter},\ \mathsf{B}\rangle$ even though they have the same prefix. A mitigation strategy is to generate the longest, acyclic traces to represent an equivalence class. For example, we can

use ⟨X, Up, E, Enter⟩ to represent this class; and since it is the longest, acyclic trace, it also indicates that ⟨X, Up, E, Enter, B⟩ is not a deviation in this class. However, there may be more than one such representative traces, which increases the difficulty in understanding the robustness.

Therefore, in the short term, we could improve our current way of representing robustness. However, in the long term, future work could also explore other representation methods that are unambiguous and more comprehensible to the user (e.g., a visual representation).

Controllability and observability. Another limitation aspect is the notion of controllability and observability. Currently, it is mainly used in the robustification process to decide which events can be and should be disabled to ensure the safety property and is also used to reflect the robustification cost. However, it is not used in the robustness computation process. Different from some other formalisms for discrete systems, the controllability and observability of events is not modeled in LTS as an intrinsic characteristic.

To incorporate this concept, one idea is to leverage IO automata [116] or interface automata [117]. They are extensions of LTS, which explicitly distinguish between input and output events, where a machine is able to control and observe its output events but can only observe its input events. Based on this notion, we may extend our robustness definition and computation method to also consider the controllability and observability of events [118]. In the long term, we could also extend our robustness notion to CSP (Communicating Sequential Processes) [119], which also explicitly distinguishes between input and output events. However, these formalisms assume that the controllability and observability of the events are fixed, whereas our robustification process allows changes in controllability and observability. Thus, it is also unclear how these changes would affect the underlying formalism and the way we compute robustness.

Disabled vs. undefined behavior. It is also worth noting that distinguishing between input and output events can also help us distinguish between disabled and undefined events. In LTS, there isn't a unified interpretation of the semantics for unspecified transitions in a model. An event being unspecified at a state can be interpreted either as: the machine disables that event, which implies controllability; or the transition is undefined, which may lead to an unknown system state and may be considered unsafe. In this thesis, we choose the disabling interpretation when defining robustness, and thus, in the robustification-by-control method, removing (disabling) a transition from the model can improve robustness. This interpretation also aligns with the weakest assumption computation method we used in this thesis, proposed by Giannakopoulou et al. [56]. However, by extending our robustness notion to IO automata, interface automata, or CSP, it may offer a clear distinction between disabled and undefined behavior.

Type of uncertainties. This thesis investigates a specific type of robustness, termed behavioral robustness. In our framework, deviations (a.k.a. uncertainties) are defined as sequences of environmental events that are not specified by the assumed environment.

Specifically, these deviated sequences should only differ in their order of events when compared to the behavior in the assumed environment, while the set of events (i.e., the alphabet) should remain unchanged. However, we did not address the type of uncertainties that are caused by unknown events from the environment. We argue that we handle the type of uncertainties that can be classified as *known unknowns*, i.e., the set of events that the machine and the environment can interact with is known and fixed, but the order in which these events may occur is unknown. On the other hand, uncertainties caused by unknown events would be classified *unknown unknowns*, which we currently cannot handle. Additionally, we also do not consider the probability associated with the occurrence of deviations, even though, in practice, some deviations may be extremely unlikely to occur, making it not cost-effective to be robust against them. However, extending our robustness notion to account for this, e.g., through probabilistic model checking [120], would require a long-term effort.

6.2 Robustification by Control

To robustify a machine (i.e., improve the robustness of a machine), we first present a method called robustification-by-control. The user specifies a set of intolerable deviations, which are then used to transform the normative environment into a deviated environment. Optionally, the user can also specify the preferred behaviors (i.e., execution traces) expected to be retained in the new design and the costs to control and observe the events of the machine and environment, in order to generate repairs that are optimal with respect to these two metrics. Internally, it leverages supervisory control theory to synthesize new designs, and we propose a novel algorithm and a set of heuristics to efficiently find optimal solutions for robustification.

Type of deviations. Our current method focuses on a deviated environment with additional behaviors compared to the expected environment, as we believe that it already captures a large and interesting class of deviations, where the environment exhibits behaviors beyond those captured in the assumed environment model (e.g., security attacks, human errors, etc.). Also, since our focus is on safety properties, which specify the upper bound of the expected behavior, a deviated environment with removed behaviors would be less likely to cause a safety violation. Future work could explore robustifying against a deviated environment that integrates both adding and removing behaviors. Additionally, we argue that removing behavior has a large impact on the satisfaction of liveness properties, as they often require certain good states to be reached, whereas removing behavior may cause some states to be unreachable. More discussion about liveness properties is provided later.

Synthesis of redesigns. We leverage supervisory control theory to synthesize robustified redesigns. In the future, we could also explore alternative synthesis techniques for robustification, such as GR(1) reactive synthesis [81, 74, 121]. However, reactive synthesis

considers properties defined in LTL, which interprets over infinite traces, whereas we consider safety properties in LTS, which consider a set of finite traces. Thus, our robustness definition and robustification method require theoretical extensions to handle this subtle semantic difference. It is also unclear how we can incorporate our two quality goals (i.e., retaining behavior and minimizing cost) into the synthesis process. Thus, we expect there would be a long-term effort to integrate GR(1) reactive synthesis into our current robustification process.

Additionally, active automata learning [122, 58] is also a potential method to synthesize redesigns. Active learning is originally used to learn a model whose behavior is equivalent to a *teacher* model whose behavior is a black box. We may also leverage this method to learn a new design by "teaching" it the good behavior (e.g., the behavior that needs to be retained) and the bad behavior (e.g., the behavior leading to a safety violation). One potential advantage of this approach is that it can return an intermediate candidate solution when a problem is hard to solve (i.e., requires too many iterations), even though the solution may not be optimal (e.g., retaining the most preferred behaviors). However, we still need to address the challenge of integrating our two quality goals into the learning process.

Extending behaviors. Our robustification-by-control method allows extending the machine's behavior by adding more controllable and observable events to the machine. However, this extending ability is limited. It cannot introduce new behaviors that are not present in the machine and the deviated environment (i.e., $M||E'$). An approach that combines our event-based approach with the abstraction-based strategy of OASIS [65] may enable a more powerful robustification process and find more feasible redesigns (with behaviors not in $M||E'$). However, we may also need to add more constraints to the robustification problem as it may result in unreasonable designs (e.g., in a voting machine, requiring a voter to "confirm" before "voting"). Also, we argue this limitation in extending behavior has a larger impact on the goal of retaining behaviors or, more generally, maintaining liveness properties, which are discussed later.

Multi-agent robustification. Another future direction is to consider the robustification of individual components of a machine. Our current method treats the machine as one entity. However, in practice, the machine could consist of multiple components, and we have local specifications for each of these components. The idea is that, instead of synthesizing a new design for the entire machine, we can identify the robustness vulnerabilities of individual components and synthesize new designs for these components. The challenge is that the synthesized new (local) designs should not only satisfy their local specifications but would also need to interact with other new or unchanged components to ensure the global specifications of the entire system. One idea is to leverage the decentralized control synthesis techniques from supervisory control [39]. Another idea is to leverage techniques of multi-agent controller synthesis such as [123]. Given the increased complexity of the problem, we expect that a long-term effort would be necessary to explore this direction.

6.3 Robustification by Specification Weakening.

The robustification-by-control method considers robustifying with respect to a fixed safety property. However, when the property is too strong, the robustified design may not be able to retain certain behaviors, causing the loss of some critical functionalities. To mitigate this issue, we present the robustification-by-specification method. Specifically, given a robustification problem and an unsatisfied preferred behavior, our approach can find a weakened safety property under which the preferred behavior can be satisfied by a robustified redesign. It considers a particular type of safety properties defined in FLTL (i.e., $\mathbf{G}(\phi \Rightarrow \psi)$) and leverages an LTL learning technique to synthesize a new property.

Weakening patterns. Future work can explore more patterns of safety properties. However, the challenge is how we decide when a property is weakened (i.e., $P \Rightarrow P'$) and whether there's a way to enable syntactic or semantic gradually weakening. Other than the weakening pattern we used in this thesis, Tabuada et al. [93, 124] propose a notion named Robust Linear Temporal Logic (rLTL), where they define an ordering of the satisfaction level of LTL formulas, which can potentially be used for weakening (e.g., a weaker version for $\mathbf{G}\phi$ is $\mathbf{FG}\phi$). However, this notion may convert a safety property into a liveness property, which our current robustification method does not support.

Another limitation is that, currently, we assume a fixed set of atomic propositions (i.e., the fluents) when synthesizing a weakened safety property. In future work, we could also consider extending this proposition set. For example, similar to the LTSA tool [48], we could implicitly create a fluent for each event in the system, which becomes true when the event occurs and becomes false when another event occurs. However, this may dramatically increase the search space, hurting performance, even though our evaluation shows that increasing the number of atomic propositions has a relatively small impact on performance. Therefore, more effort is needed to be done to balance between the search space and the performance of weakening.

Quality of weakening. Although we have added syntactic constraints to avoid unhelpful expressions such as tautologies and idempotent expressions, the LTL learning process may still generate unsatisfactory solutions. For example, when all the examples are identified as positive, then $P' = true$ becomes a valid solution, which can be achieved by making the antecedent $false$ (e.g., $p \wedge \neg p \Rightarrow q$). One way to mitigate this issue is to add more syntactic constraints to avoid them. On the other hand, we could also explore a feedback loop for providing negative examples to the learner to avoid weakening the property too much. However, unlike the weakening loop presented in [51] where over-weakening can be automatically witnessed by a counterexample, over-weakening in our use case cannot be automatically detected (as a weaker safety property would be easier to satisfy). Thus, it still relies on developers' domain knowledge to provide or identify the negative examples.

Semantic-based weakening. We consider only a syntactic weakening pattern and use ATLAS [41], a constrained LTL learning tool that we built, to synthesize new safety proper-

ties. In the future, we could also explore semantic-based weakening patterns. In general, it is challenging to decide whether a weakened property is semantically minimally weakened. One idea is to encode semantic-based weakening patterns as constraints in the learning tool. However, this is much more challenging because it is unclear what semantic constraints should be provided to guide the search and, as far as we are aware, there does not exist a learning tool that supports defining semantic constraints. Another idea is to consider limited patterns of formulas. For example, in this thesis, we consider only safety invariants. Since the valuation of such a property depends solely on individual system states, to guarantee semantically minimal weakening, we could maximize the number of unsafe states so that only a few executions are considered acceptable, leading to a safety property that would not be too weak.

Full-automation of weakening. Our current weakening technique is a semi-automated approach. Given an unsatisfied preferred behavior, we automatically generate a set of example executions and assume that the stakeholders are responsible for manually deciding which examples are acceptable or not. Future work could also explore automating the example identification process. One potential direction is that, given a set of examples, the algorithm could attempt to separate the set into two subsets: one with positive examples and one with negative examples. Since we consider only safety invariants, all visited states in the positive set should be safe states, and there should exist one or more unsafe states in the negative set. Additionally, the guessing heuristic could try to maximize the number of unsafe states to incorporate semantic-based gradually weakening into the process. This guessing process could be repeated multiple times until a satisfactory property is found.

6.4 Liveness Properties

Our robustness computation and improvement method mainly considers safety properties, and only two weak classes of liveness properties are considered in robustification (i.e., progress properties and preferred behaviors). We envision that a comprehensive support for robustness analysis and improvement of liveness properties would require non-trivial theoretical extensions to the existing framework.

For example, the first challenge is the underlying formalism. In this thesis, we focus on safety properties defined in LTS, which describes a set of finite traces. However, liveness properties often cannot be represented in LTS, and we may need to extend the formalism to LTL and consider infinite traces. Moreover, liveness properties make more sense with infinite traces since they often describe the desire that certain events would eventually occur in the (unbounded) future.

Then, another challenge is how we compute robustness with respect to a liveness property. We could continue leveraging the computation of the weakest assumption. For example, Farzan et al. [125] present a method to compute weakest assumption for liveness properties based on ω-regular language. Another idea is to leverage the concept of checking liveness as safety [126] or consider bounded-time liveness, such as [127, 128]. With these

techniques, we may reuse our existing robustness computation method for safety to handle liveness.

Finally, it is also challenging to improve robustness with respect to liveness properties. 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. Although our approach can extend the machine's behavior by adding new controllable and observable events, it cannot add event traces outside the machine and the deviated environment ($M||E'$). For example, it cannot add retries of message sending in a network protocol. One idea is to integrate the *abstract-based* approach presented in OASIS [65], which supports adding new event traces to the redesign, but additional constraints may be needed to avoid unusual designs. Furthermore, both of these methods consider a fixed set of events for the machine and environment, but new events may need to be introduced to robustify against a liveness property.

In sum, given the significant differences in computing and improving robustness against liveness properties, we expect at least one or two additional years of effort to extend our existing robustness techniques to support liveness.

6.5 ML and CPS Robustness

In this thesis, we classify software systems into three categories: conventional software systems, machine learning (ML) systems, and cyber-physical systems (CPS). We focus on the systematic analysis and improvement of the behavioral robustness of conventional software systems. Specifically, we consider conventional software systems as systems that can be modeled in discrete transition systems and thus use labeled transition systems (LTS) to model their behavior. Although we are also interested in the behavioral robustness of ML systems and CPS, it is outside the scope of this thesis.

ML systems typically involve statistical models trained and optimized against certain datasets. Their concrete behavior is hard to explain, and these models are often treated as black boxes, especially for complex models like Deep Neural Networks (DNN) [129]. Also, CPS interact closely with the physical world, such as aircraft or autonomous vehicles, which often involve modeling, controlling, and manipulating physical processes. These processes often need to be characterized by differential equations or hybrid logic [130].[1] Therefore, ML systems and CPS require significantly different behavior modeling approaches where LTS cannot be directly applied.

There would need non-trivial theoretical extensions to our current robustness notion to analyze and reason about the behavioral robustness of ML systems and CPS. It may need to incorporate probabilistic modeling and properties [120] or leverage statistical model checking techniques [131]. For instance, in our work [132], we present a robustness analysis framework for CPS with reinforcement learning controllers. It employs Signal Temporal Logic (STL) to specify the expected behavior of a CPS and assumes that the system

[1] In literature, CPS covers a wide range of systems including some of our case studies such as the infusion pump or the voting machine. However, we use the term CPS to specifically refer to those systems that require modeling of the physical world in continuous models.

under test and the controller are black boxes. Then, it uses an optimization-based method to find robustness violations, instead of computing the robustness. Therefore, given the significant differences in the underlying problems, robustness analysis and improvement of ML systems and CPS are out of the scope of this thesis.

Appendix A

Appendix

A.1 Models of Case Studies

A.1.1 Radiation Therapy Machine

The following code snippets show the FSP specifications of the radiation therapy machine:

Machine.

```
INTERFACE = (x -> CONFIRM | e -> CONFIRM),
CONFIRM = (up -> INTERFACE | enter -> FIRE),
FIRE = (up -> CONFIRM | b -> enter -> INTERFACE).

const NotSet = 0
const Xray = 1
const EBeam = 2
range BeamState = NotSet .. EBeam

const ToXray = 3
const ToEBeam = 4
range BeamSwitch = ToXray .. ToEBeam

BEAM = BEAM[NotSet],
BEAM[mode:BeamState] = (
    when (mode == NotSet) x -> set_xray -> BEAM[Xray]
    |
    when (mode == NotSet) e -> set_ebeam -> BEAM[EBeam]
    |
    // Xray mode
    when (mode == Xray) x -> BEAM[Xray]
    |
    when (mode == Xray) e -> BEAM_SWITCH[ToEBeam]
    |
    when (mode == Xray) b -> fire_xray -> reset -> BEAM
    |
    // EBeam mode
    when (mode == EBeam) e -> BEAM[EBeam]
    |
```

```
    when (mode == EBeam) x -> BEAM_SWITCH[ToXray]
    |
    when (mode == EBeam) b -> fire_ebeam -> reset -> BEAM
),
BEAM_SWITCH[switch:BeamSwitch] = (
    // EBeam to Xray
    when (switch == ToXray) x -> BEAM_SWITCH[ToXray]
    |
    when (switch == ToXray) e -> BEAM[EBeam]
    |
    when (switch == ToXray) b -> fire_ebeam -> reset -> BEAM
    |
    when (switch == ToXray) set_xray -> BEAM[Xray]
    |
    // Xray to EBeam
    when (switch == ToEBeam) e -> BEAM_SWITCH[ToEBeam]
    |
    when (switch == ToEBeam) x -> BEAM[Xray]
    |
    when (switch == ToEBeam) b -> fire_xray -> reset -> BEAM
    |
    when (switch == ToEBeam) set_ebeam -> BEAM[EBeam]
).

SPREADER = (e -> OUTPLACE | x -> SPREADER),
OUTPLACE = (e -> OUTPLACE | x -> SPREADER).

||SYS = (INTERFACE || BEAM || SPREADER).
```

Normative environment.

```
ENV = (x -> ENV_1 | e -> ENV_1),
ENV_1 = (enter -> ENV_2),
ENV_2 = (b -> enter -> ENV)+{up}.
```

Deviated environment.

```
ENV = (x -> ENV_1 | e -> ENV_1),
ENV_1 = (enter -> ENV_2 | up -> ENV),
ENV_2 = (b -> enter -> ENV | up -> ENV_1).
```

Safety property in LTS.

```
const True = 1
const False = 0
range Bool = False .. True

P = P[False][True],
P[isXray:Bool][isInPlace:Bool] = (
    set_xray -> P[True][isInPlace]
    |
    set_ebeam -> P[False][isInPlace] | reset -> P[False][isInPlace]
    |
    x -> P[isXray][True]
```

```
    |
    e -> P[isXray][False]
    |
    when (isXray == False || isInPlace == True) b -> P[isXray][isInPlace]
).
```

Weak safety property in FLTL.

```
fluent Xray = <set_xray, {set_ebeam, reset}>
fluent EBeam = <set_ebeam, {set_xray, reset}>
fluent InPlace = <x, e> initially 1
fluent Fired = <{fire_xray, fire_ebeam}, reset>

assert OVER_DOSE = [](Xray && Fired -> InPlace)
```

Strong safety property in FLTL.

```
fluent Xray = <set_xray, {set_ebeam, reset}>
fluent EBeam = <set_ebeam, {set_xray, reset}>
fluent InPlace = <x, e> initially 1
fluent Fired = <{fire_xray, fire_ebeam}, reset>

assert OVER_DOSE = [](Xray -> InPlace)
```

A.1.2 Network Protocols

The following code snippets show the FSP specifications of the network protocols:

Naive protocol.

```
range B= 0..1

SENDER = (input -> send[B] -> getack[B] -> SENDER).
RECEIVER = (rec[B] -> output -> ack[B] -> RECEIVER).
||SYS = (SENDER || RECEIVER).
```

ABP.

```
range B= 0..1

INPUT = (input -> SENDING[0]),
SENDING[b:B] = (send[b] -> SENDING[b]
              | getack[b] -> input -> SENDING[!b]
              | getack[!b] -> SENDING[b]).

OUTPUT = (rec[0] -> output -> ACKING[0]),
ACKING[b:B] = (ack[b] -> ACKING[b]
             | rec[b] -> ACKING[b]
             | rec[!b] -> output -> ACKING[!b]).

||SYS = (INPUT || OUTPUT).
```

Normative environment.

```
range E_B = 0 .. 1

CHANNEL = (in[b:E_B] -> out[b] -> CHANNEL).

||ENV = (trans:CHANNEL || ack:CHANNEL)
      / {send/trans.in, rec/trans.out, ack/ack.in, getack/ack.out}.
```

Deviated environment.

```
range E_B = 0 .. 1

menu ERR_ACTS = {lose, duplicate, corrupt}

CHANNEL = (in[b:E_B] -> TRANSIT[b]),
TRANSIT[b:E_B] = (out[b] -> CHANNEL | lose -> CHANNEL).

||ENV = (trans:CHANNEL || ack:CHANNEL)
      / {send/trans.in, rec/trans.out, ack/ack.in, getack/ack.out}.
```

Safety property in LTS.

```
P = (input -> output -> P).
```

A.1.3 Voting Machine

Figure A.1 shows a simplified model of the voting machine. In a typical scenario, a voter is expected to interact with the machine by carrying out the following actions: (1) Enter password to verify their identify (password); (2) Select the candidate of their choice (select); (3) Proceed to the next step by pressing vote (vote) or return to the previous step by pressing back (back); (4) Complete the vote by confirming the selection (confirm) or return to the previous step by pressing back (back). The voting machine is placed inside a voting booth, and a nearby election official is responsible for guiding voters to and away from the booth. We assume that the voting booth can be occupied by at most one person at a time. The election official cannot enter the password to vote but can perform all the other actions. In addition, the election official can reset the machine to the initial state even if the voting process is not completed (reset). Figure A.2 shows the LTS for this normative environment behavior.

The following code snippets show the FSP specifications of the voting machine:

Machine.

```
EM = (password -> P1),
P1 = (select -> P2 | reset -> EM),
P2 = (vote -> P3 | back -> P1 | reset -> EM),
P3 = (confirm -> EM | back -> P2 | reset -> EM).
```

Normative environment.

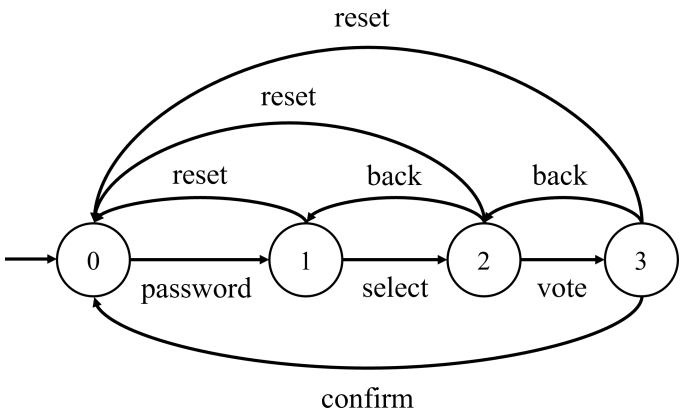


Figure A.1: Voting machine model.

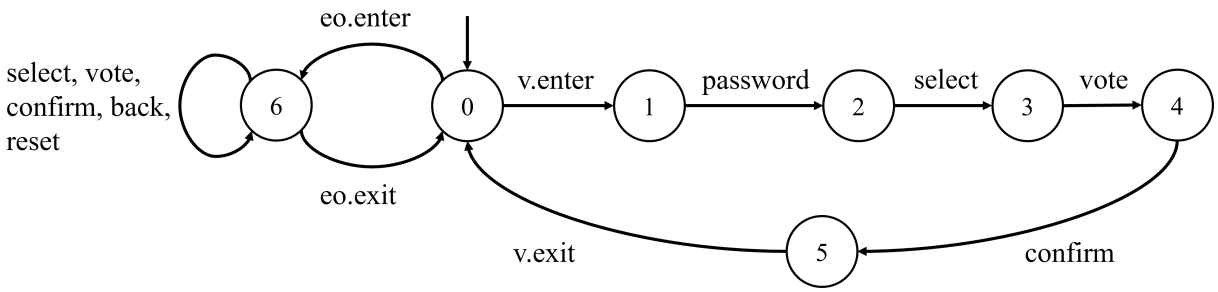


Figure A.2: Normative operator model of the voting machine.

```
ENV = (v.enter -> VOTER | eo.enter -> EO),
VOTER = (password -> VOTER1),
VOTER1 = (select -> VOTER2),
VOTER2 = (vote -> VOTER3 | back -> VOTER1),
VOTER3 = (confirm -> v.exit -> ENV | back -> VOTER2),
EO = (select -> EO | vote -> EO | confirm -> EO | back -> EO
    | reset -> EO | eo.exit -> ENV).
```

Deviated environment.

```
ENV = (v.enter -> VOTER | eo.enter -> EO),
VOTER = (password -> VOTER | select -> VOTER | vote -> VOTER
       | confirm -> VOTER | back -> VOTER | v.exit -> ENV),
EO = (select -> EO | vote -> EO | confirm -> EO | back -> EO
    | reset -> EO | eo.exit -> ENV).
```

Safety property in LTS.

```
const NoBody = 0
const Voter = 1
const EO = 2
range WHO = NoBody..EO
```

```
P = VOTE[NoBody][NoBody][NoBody],
VOTE[in:WHO][sel:WHO][v:WHO] = (
      v.enter -> VOTE[Voter][sel][v] | eo.enter -> VOTE[EO][sel][v]
    | password -> VOTE[in][sel][in]
    | select -> VOTE[in][in][v]
    | when (sel == v) confirm -> VOTE[in][NoBody][NoBody]
).
```

Weak safety property in FLTL.

```
fluent InVoting= <password, {confirm, reset}>
fluent VoterIn = <v.enter, v.exit>
fluent OfficialIn = <eo.enter, eo.exit>
fluent CriticalOp = <
  {select, vote, confirm},
  {password, back, reset, eo.enter, eo.exit, v.enter, v.exit}
>

assert VOTE_INTEGRITY = [](OfficialIn -> (!InVoting || !CriticalOp))
```

Strong safety property in FLTL.

```
fluent InVoting= <password, {confirm, reset}>
fluent VoterIn = <v.enter, v.exit>
fluent OfficialIn = <eo.enter, eo.exit>
fluent CriticalOp = <
  {select, vote, confirm},
  {password, back, reset, eo.enter, eo.exit, v.enter, v.exit}
>

assert VOTE_INTEGRITY = [](OfficialIn -> !InVoting)
```

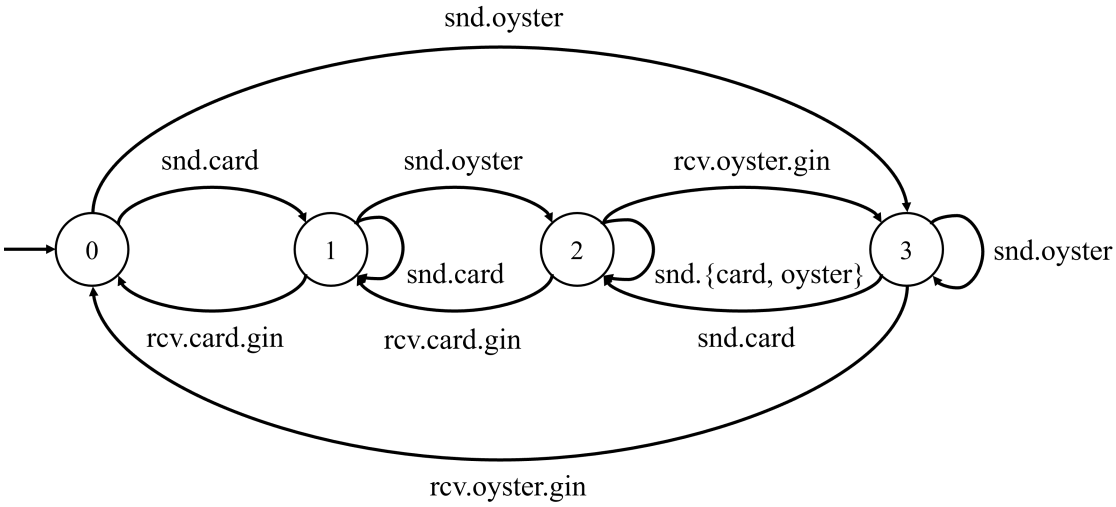
A.1.4 Oyster Transportation Fare System

Figure A.3 and A.4 show the LTS's of the Oyster system and Figure A.4c shows the LTS of the behavior of the user. Specifically, when entering the gate (M_{enter}), the user either taps their Oyster transportation card (snd.oyster) or uses another payment such as a credit card (snd.card); and the machine acknowledges with a rcv.oyster.gin or rcv.card.gin event, respectively. Then, when exiting the gate (M_{exit}), the user uses the same method to complete the payment, which sends a snd.oyster.gin or snd.card.gin event, respectively. Finally, the machine acknowledges with a rcv.oyster.fin or rcv.card.fin event indicating the completion of the payment. In addition, M_{oyster} and M_{card} model the balance of the Oyster transportation card and the credit card; and when there's no balance, the user has to reload the balance (rld.oyster and rld.card).

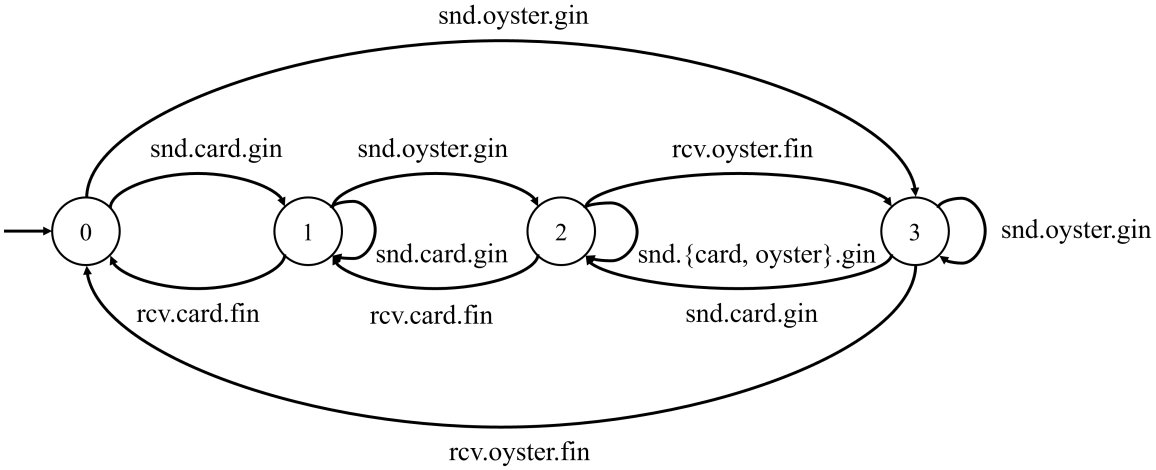
The following code snippets show the FSP specifications of the Oyster system:

Machine.

```
GATE_IN = (snd.oyster -> OYSTER | snd.card -> CARD),
OYSTER = (rcv.oyster.gin -> GATE_IN | snd.oyster -> OYSTER
        | snd.card -> ANY),
```



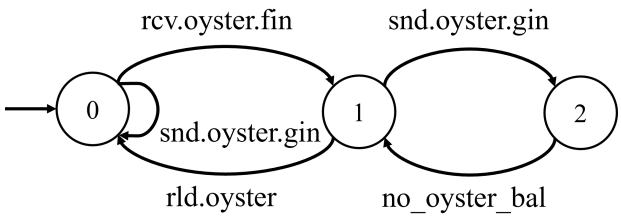
(a) Gate entering control M_{enter}.



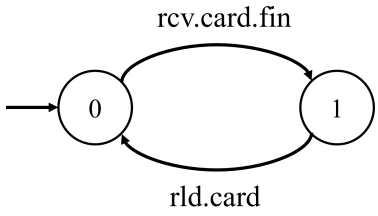
(b) Gate exiting control M_{exit}.

Figure A.3: Models of the gate control of the Oyster transportation system.

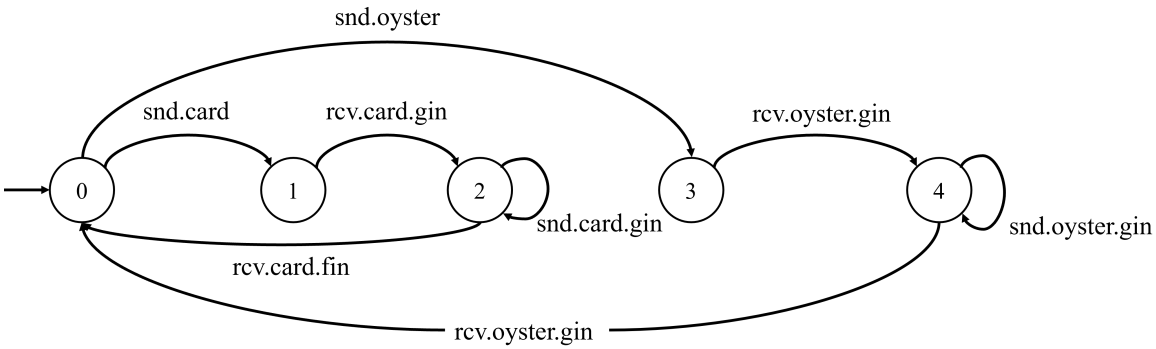
```
CARD = (rcv.card.gin -> GATE_IN | snd.card -> CARD | snd.oyster -> ANY),
ANY = (rcv.oyster.gin -> OYSTER | rcv.card.gin -> CARD
     | snd.card -> ANY | snd.oyster -> ANY).

GATE_OUT = (snd.oyster.gin -> OYSTER | snd.card.gin -> CARD),
OYSTER = (rcv.oyster.fin -> GATE_OUT
        | snd.oyster.gin -> OYSTER | snd.card.gin -> ANY),
CARD = (rcv.card.fin -> GATE_OUT
      | snd.card.gin -> CARD | snd.oyster.gin -> ANY),
ANY = (rcv.oyster.fin -> OYSTER | rcv.card.fin -> CARD
     | snd.card.gin -> ANY | snd.oyster.gin -> ANY).

const MAX_BAL = 5
```



(a) Oyster card balance M_{oyster}.



(b) Credit card balance M_{card}.



(c) Normative behavior model of the Oyster's user.

Figure A.4: Models of the fare management of the Oyster transportation system.

```
OYSTER_BAL = OYSTER_BAL[MAX_BAL],
OYSTER_BAL[b:0..MAX_BAL] = (
    when (b < MAX_BAL) rld.oyster[c:1..(MAX_BAL-b)] -> OYSTER_BAL[b+c]
    |
    when (b > 0) rcv.oyster.fin -> OYSTER_BAL[b-1]
    |
    when (b == 0) snd.oyster.gin -> no_oyster_bal -> OYSTER_BAL[b]
    |
    when (b > 0) snd.oyster.gin -> OYSTER_BAL[b]
).

CARD_BAL = CARD_BAL[MAX_BAL],
CARD_BAL[b:0..MAX_BAL] = (
    when (b < MAX_BAL) rld.card[c:1..(MAX_BAL-b)] -> CARD_BAL[b+c]
    |
    when (b > 0) rcv.card.fin -> CARD_BAL[b-1]
).
```

Normative environment.

```
E = (snd.oyster -> rcv.oyster.gin -> E1 | snd.card -> rcv.card.gin -> E2),
E1 = (snd.oyster.gin -> E1 | rcv.oyster.fin -> E),
E2 = (snd.card.gin -> E2 | rcv.card.fin -> E).
```

Deviated environment.

```
E = (snd.oyster -> rcv.oyster.gin -> E1 | snd.card -> rcv.card.gin -> E2),
E1 = (snd.oyster.gin -> E1 | rcv.oyster.fin -> E | fg -> E3),
E3 = (snd.card.gin -> E3 | rcv.card.fin -> E | rmb -> E1),
E2 = (snd.card.gin -> E2 | rcv.card.fin -> E | fg -> E4),
E4 = (snd.oyster.gin -> E4 | rcv.oyster.fin -> E | rmb -> E2).
```

Safety property in LTS.

```
P = (rcv.oyster.gin -> rcv.oyster.fin -> P
   | rcv.card.gin -> rcv.card.fin -> P).
```

Weak safety property in FLTL.

```
fluent UseCard = <rcv.card.gin, rcv.card.fin>
fluent UseOyster = <rcv.oyster.gin, rcv.oyster.fin>
fluent CardOut = <rcv.card.fin, {snd.card, snd.oyster}>
fluent OysterOut = <rcv.oyster.fin, {snd.card, snd.oyster}>
fluent NoOysterBal = <no_oyster_bal, rld.oyster[1]>

assert NO_COLLISION = [](CardOut -> !UseOyster || NoOysterBal)
                   && [](OysterOut -> !UseCard)
```

Strong safety property in FLTL.

```
fluent UseCard = <rcv.card.gin, rcv.card.fin>
fluent UseOyster = <rcv.oyster.gin, rcv.oyster.fin>
fluent CardOut = <rcv.card.fin, {snd.card, snd.oyster}>
fluent OysterOut = <rcv.oyster.fin, {snd.card, snd.oyster}>
fluent NoOysterBal = <no_oyster_bal, rld.oyster[1]>

assert NO_COLLISION = [](CardOut -> !UseOyster)
                   && [](OysterOut -> !UseCard)
```

A.1.5 Infusion Pump

The following code snippets show the FSP specifications of the infusion pump:

Machine.

```
//======================
// Constants and Ranges
//======================

//
// States of the pump alarm
//
const AlarmSilenced = 0
const AlarmSounds = 1

range AlarmState = AlarmSilenced .. AlarmSounds

//
```

```
// States of the pump settings
//
const ParamsNotSet = 2    // pump parameters not set yet
const ParamsSet    = 3    // pump parameters already set

range ParamsStateT = ParamsNotSet .. ParamsSet

//
// Locked/unlocked states of a line with respect to a pump channel
//
const LineUnlocked = 4  // line not locked into a pump channel
const LineLocked   = 5  // line locked into a pump channel

range LineLockStateT = LineUnlocked .. LineLocked

//
// Locked/unlocked states of the pump unit
//
const UnitUnlocked = 6  // the keypad of the pump is not locked
const UnitLocked   = 7  // the keypad of the pump is locked

range UnitLockStateT = UnitUnlocked .. UnitLocked

//
// Plugged/unplugged states of the pump unit
//

const Unplugged = 8 //the pump is not plugged in
const Plugged = 9 //the pump is plugged in

range PluggedState = Unplugged .. Plugged

//
// Battery states of the pump unit
//

const BatteryCharge = 12 //the battery has charge
const BatteryLow = 11
const BatteryEmpty = 10 //battery has no charge

range BatteryState = BatteryEmpty .. BatteryCharge

//
// System State
//

const SystemOff = 13
const SystemOn = 14

range SystemState = SystemOff .. SystemOn

//====================
```

```
// Alarm Definitions
//=====================

ALARM = ALARM[AlarmSilenced],
ALARM[alarm_state:AlarmState] =
(
  when (alarm_state == AlarmSounds)
    alarm_rings -> ALARM[alarm_state]
  |
  when (alarm_state == AlarmSounds)
    alarm_silence -> ALARM[AlarmSilenced]
  |
  enable_alarm -> ALARM[AlarmSounds]
  |
  power_failure -> ALARM
).

//=====================
// Process Definitions
//=====================

// Initial Pump State
PUMP_POWER = POWERED[Unplugged][BatteryEmpty],

// Pump has power but not on -- keep track of whether
// there is any battery and plug state
POWERED[plug_state:PluggedState][battery_state:BatteryState]  =
(
  when (plug_state == Unplugged)
    plug_in -> POWERED[Plugged][battery_state]
  |
  when (plug_state == Plugged)
    unplug -> POWERED[Unplugged][battery_state]
  |
  when (battery_state != BatteryEmpty)
    turn_on -> POWER_ON[plug_state][battery_state]
  |
  when (plug_state == Plugged && battery_state != BatteryCharge)
    battery_charge -> POWERED[plug_state][battery_state+1]
),

// Pump is on
POWER_ON[plug_state:PluggedState][battery_state:BatteryState] =
(
  when (plug_state == Plugged)
    unplug -> POWER_ON[Unplugged][battery_state]
  |
  when (plug_state == Unplugged)
    plug_in -> POWER_ON[Plugged][battery_state]
  |
  turn_off -> POWERED[plug_state][battery_state]
  |
```

```
  when (plug_state == Unplugged && battery_state == BatteryCharge)
    battery_spent -> POWER_ON[plug_state][BatteryLow]
  |
  when (plug_state == Unplugged && battery_state == BatteryLow)
    power_failure -> POWERED[Unplugged][BatteryEmpty]
  |
  when (plug_state == Plugged && battery_state != BatteryCharge)
    battery_charge -> POWER_ON[plug_state][battery_state+1]
  |
  when (plug_state == Unplugged && battery_state == BatteryLow)
    enable_alarm -> POWER_ON[plug_state][battery_state]
).

//
// Dispense complete
//

const Dispensing = 15
const DispenseDone = 16

range DispenseState = Dispensing .. DispenseDone

//=====================
// Process Definitions
//=====================
range NUM_LINE = 1..1

LINE = LINE[LineUnlocked],

LINE[lineLock:LineLockStateT] = (
  turn_on -> LINESETUP[ParamsNotSet][lineLock]
),

//
// Setupmode for the line

LINESETUP[params:ParamsStateT][lineLock:LineLockStateT] =
(
  turn_off -> LINE[lineLock]
  |
  power_failure -> LINE[lineLock]
  |
  when (params == ParamsNotSet && lineLock == LineUnlocked)
    set_rate -> LINESETUP[ParamsSet][lineLock]
  |
  when (params == ParamsSet && lineLock == LineUnlocked)
    clear_rate -> LINESETUP[ParamsNotSet][lineLock]
  |
  when (params == ParamsSet && lineLock == LineUnlocked)
    lock_line -> LINESETUP[params][LineLocked]
  |
  when (lineLock == LineLocked)
```

```
    erase_and_unlock_line -> LINESETUP[params][LineUnlocked]
  |
  when (params == ParamsSet && lineLock == LineLocked)
    confirm_settings -> LINEINFUSION[UnitUnlocked]
),

//
// Pump in infusion mode:
// - Always be able to turn the unit off, even if locked
// - Allow the user to lock/unlock the unit
// - Errors could occur with the pump (e.g., line became pinched or plugged)
//
LINEINFUSION[unitLock:UnitLockStateT] =
(
  turn_off -> LINE[LineLocked]
  |
  power_failure -> LINE[LineLocked]
  |
  when (unitLock == UnitUnlocked)
    change_settings -> LINESETUP[ParamsSet][LineLocked]
  |
  when (unitLock == UnitUnlocked)
    lock_unit -> LINEINFUSION[UnitLocked]
  |
  when (unitLock == UnitLocked)
    unlock_unit -> LINEINFUSION[UnitUnlocked]
  |
  when (unitLock == UnitLocked)
    start_dispense -> DISPENSE[SystemOn][Dispensing]
),

DISPENSE[system_state:SystemState][dispense:DispenseState] =
(
  dispense_main_med_flow -> DISPENSE[system_state][DispenseDone]
  |
  when (system_state == SystemOn && dispense == DispenseDone)
    flow_complete -> unlock_unit -> LINESETUP[ParamsNotSet][LineLocked]
  |
  power_failure -> DISPENSE[SystemOff][Dispensing]
  |
  when (system_state == SystemOff)
    turn_on -> LINESETUP[ParamsNotSet][LineLocked]
  |
  when (system_state == SystemOn)
    turn_off -> LINE[LineLocked]
).

||LINES = (line[NUM_LINE]:LINE)/{
  turn_on/line[NUM_LINE].turn_on,
  turn_off/line[NUM_LINE].turn_off,
  power_failure/line[NUM_LINE].power_failure}.
```

```
||SYS = (PUMP_POWER || ALARM || LINES).
```

Normative environment.

```
range LINES = 1..1

//
// Set of actions that the user of the LTSA tool can control in an
// animation of this model.
//
menu UserControlMenu = {
  alarm_silence,

  line[LINES].change_settings,
  line[LINES].clear_rate,
  line[LINES].confirm_settings,
  line[LINES].erase_and_unlock_line,
  line[LINES].lock_line,
  line[LINES].lock_unit,
  line[LINES].set_rate,
  line[LINES].unlock_unit,

  plug_in,
  turn_off,
  turn_on,
  unplug
}

ENV = (plug_in -> turn_on -> CHOOSE),
CHOOSE = (line[i:LINES].set_rate -> RUN[i] | turn_off -> unplug -> ENV),
RUN[i:LINES] = (
    line[i].lock_line -> line[i].confirm_settings -> line[i].lock_unit ->
    line[i].start_dispense -> line[i].unlock_unit ->
    line[i].erase_and_unlock_line -> CHOOSE
)+{line[LINES].clear_rate, line[LINES].change_settings}.
```

Deviated environment.

```
range LINES = 1..1

//
// Set of actions that the user of the LTSA tool can control in
// an animation of this model.
//
menu UserControlMenu = {
  alarm_silence,

  line[LINES].change_settings,
  line[LINES].clear_rate,
  line[LINES].confirm_settings,
  line[LINES].erase_and_unlock_line,
  line[LINES].lock_line,
```

```
  line[LINES].lock_unit,
  line[LINES].set_rate,
  line[LINES].unlock_unit,

  plug_in,
  turn_off,
  turn_on,
  unplug
}

ENV = (
  alarm_silence -> ENV |

  line[LINES].change_settings -> ENV |
  line[LINES].clear_rate -> ENV |
  line[LINES].confirm_settings -> ENV |
  line[LINES].erase_and_unlock_line -> ENV |
  line[LINES].lock_line -> ENV |
  line[LINES].lock_unit -> ENV |
  line[LINES].set_rate -> ENV |
  line[LINES].unlock_unit -> ENV |

  plug_in -> ENV |
  turn_off -> ENV |
  turn_on -> ENV |
  unplug -> ENV
)+{unplug}.
```

Safety property in LTS.

```
P = (line[1].set_rate -> RATE_SET | power_failure -> P),
RATE_SET = (line[1].set_rate -> RATE_SET | power_failure -> P
          | line[1].dispense_main_med_flow -> DISPENSE),
DISPENSE = (line[1].dispense_main_med_flow -> DISPENSE | power_failure -> P
          | line[1].flow_complete -> P).
```

Weak safety property in FLTL.

```
fluent Dispensing = <
  line[1].dispense_main_med_flow,
  {alarm_rings, alarm_silence, battery_charge, battery_spent, enable_alarm,
   line[1].change_settings, line[1].clear_rate, line[1].confirm_settings,
   line[1].erase_and_unlock_line, line[1].flow_complete, line[1].lock_line,
   line[1].lock_unit, line[1].set_rate, line[1].start_dispense,
   line[1].unlock_unit, plug_in, power_failure, turn_off, turn_on, unplug}
>
fluent PowerFailed = <power_failure, battery_charge>
fluent Plugged = <plug_in, unplug>

assert SAFE_DISPENSE = [](Dispensing -> Plugged || !PowerFailed)
```

Strong safety property in FLTL.

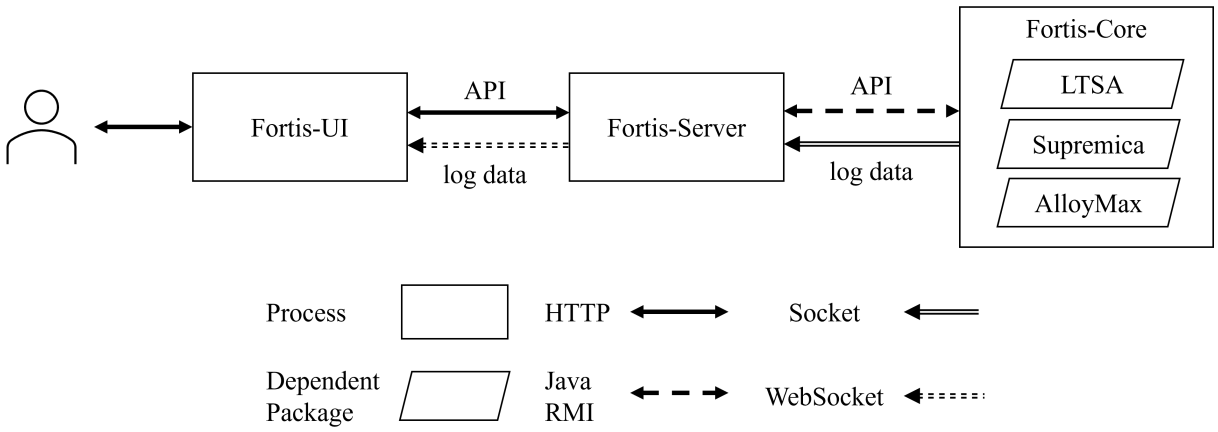


Figure A.5: Architecture of Fortis.

```
fluent Dispensing = <
  line[1].dispense_main_med_flow,
  {alarm_rings, alarm_silence, battery_charge, battery_spent, enable_alarm,
   line[1].change_settings, line[1].clear_rate, line[1].confirm_settings,
   line[1].erase_and_unlock_line, line[1].flow_complete, line[1].lock_line,
   line[1].lock_unit, line[1].set_rate, line[1].start_dispense,
   line[1].unlock_unit, plug_in, power_failure, turn_off, turn_on, unplug}
>
fluent PowerFailed = <power_failure, battery_charge>
fluent Plugged = <plug_in, unplug>

assert SAFE_DISPENSE = [](Dispensing -> Plugged)
```

A.2 Usage of Fortis

Architecture. Figure A.5 describes the architecture of Fortis. Due to technical constraints, it employs a 3-tier architecture. The user accesses Fortis through a web-based user interface. The web-interface then invokes the server through HTTP. Finally, the server invokes the actual implementation of Fortis through Java RMI APIs. In addition, we use socket to send back the logging messages from the core implementation to the server and use web-socket to send back them to the web interface.

Features. Then, the following screenshots demonstrate the usage for the main features of Fortis.

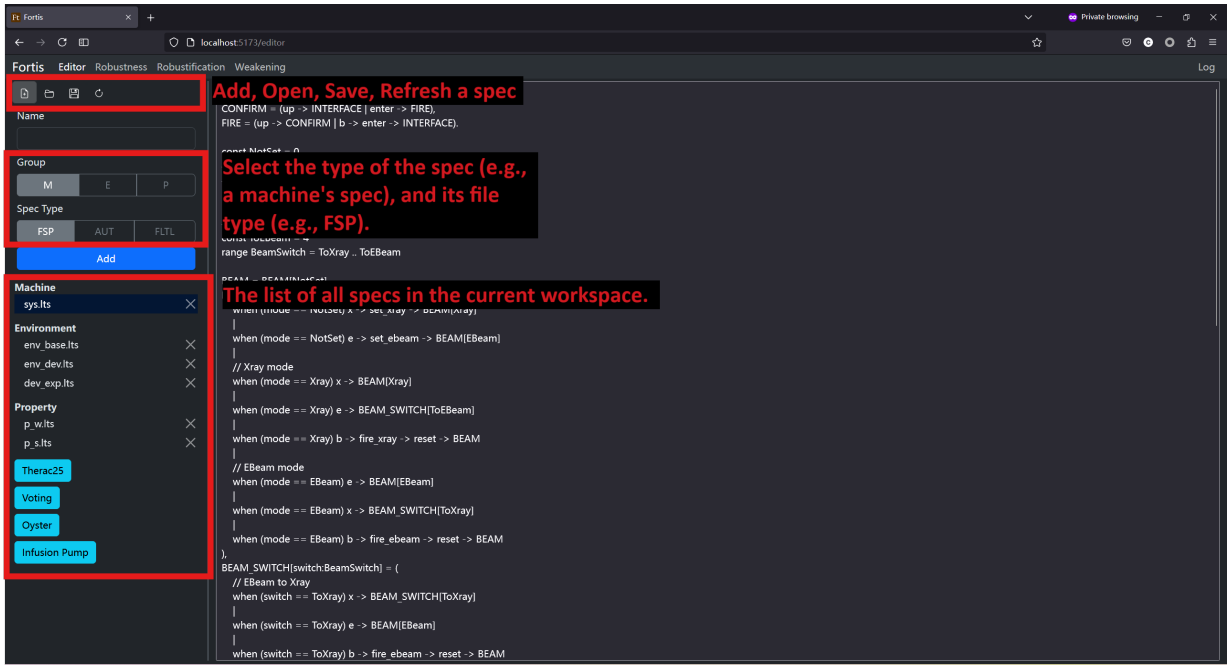


Figure A.6: The sidebar for managing specifications of a problem.

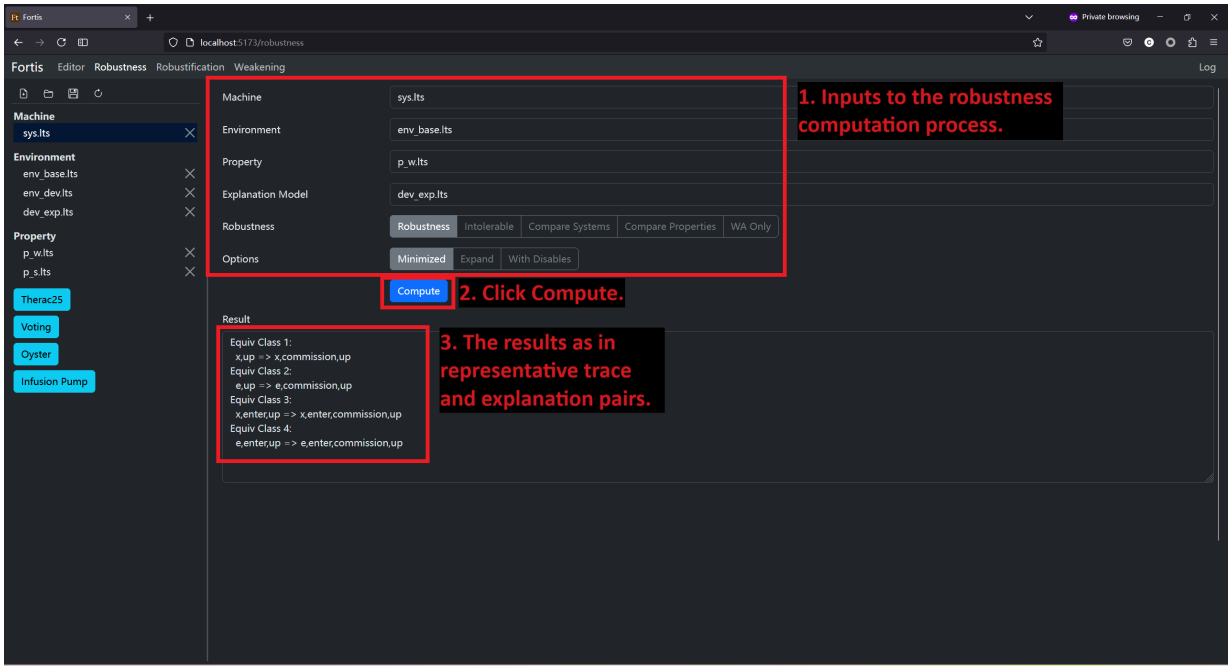


Figure A.7: The steps for computing robustness with Fortis.

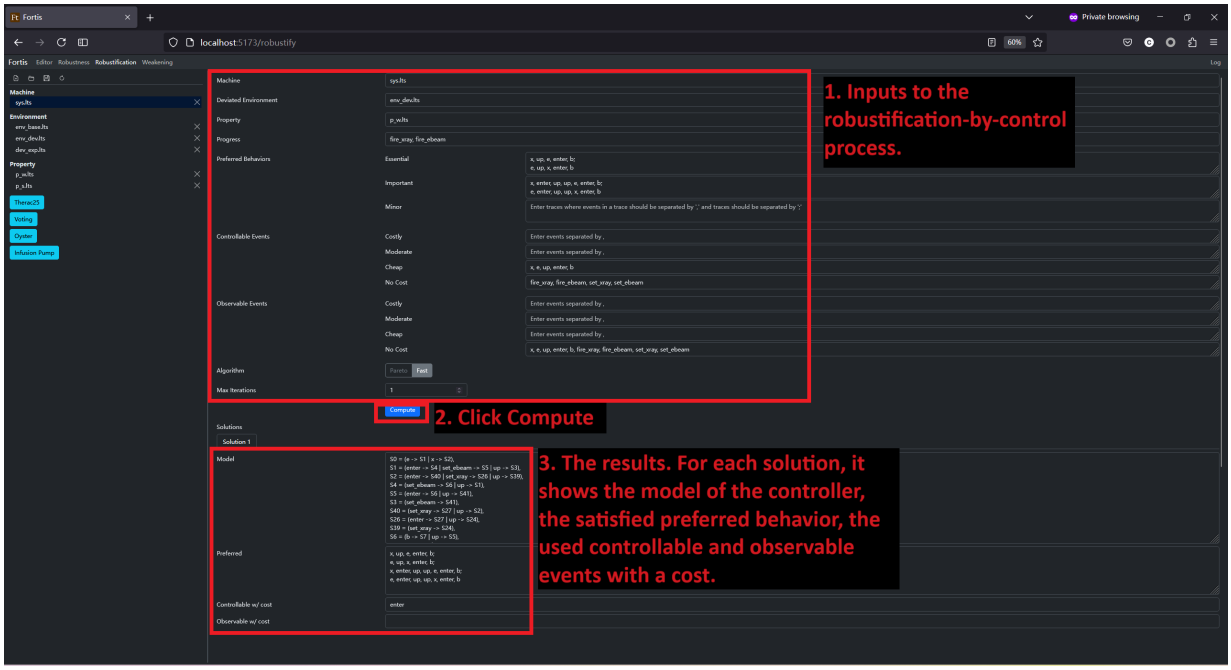


Figure A.8: The steps for conducting robustification-by-control with Fortis.

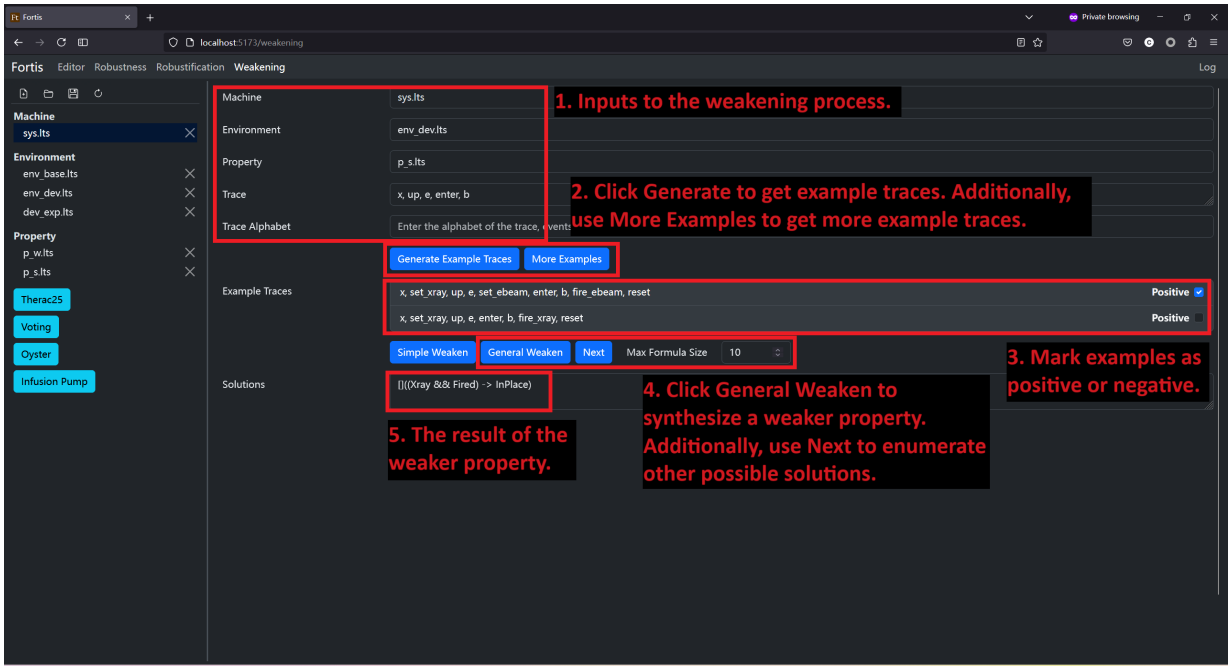


Figure A.9: The steps for conducting specification-weakening with Fortis.

Bibliography

[1] D. Giannakopoulou, J. Kramer, and J. Magee, "Practical behaviour analysis for distributed software architectures," in *UK Programmable Networks and Telecommunications Workshop*, 1998. (document), 2.6.3, 2.10

[2] E. E. Ogheneovo, "Software dysfunction: Why do software fail?" *Journal of Computer and Communications*, vol. 2014, 2014. 1.1, 1.2

[3] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990. 1.1

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. 1.1

[5] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. 1.1

[6] E. M. Clarke, T. A. Henzinger, and H. Veith, *Introduction to Model Checking*. Cham: Springer International Publishing, 2018, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_1 1.1

[7] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*, 1997, pp. 72–79. 1.1, 5.1

[8] N. Kropp, P. Koopman, and D. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, 1998, pp. 230–239. 1.1

[9] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977. 1.1, 2.3, 3.4.2

[10] B. W. Boehm, "Software engineering economics," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4–21, 1984. 1.2

[11] J. a. Brunet, G. C. Murphy, R. Terra, J. Figueiredo, and D. Serey, "Do developers discuss design?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 340–343. [Online]. Available: https://doi.org/10.1145/2597073.2597115 1.2

[12] G. Viviani, C. Janik-Jones, M. Famelis, X. Xia, and G. C. Murphy, "What design topics do developers discuss?" in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 328–331. [Online]. Available: https://doi.org/10.1145/3196321.3196357 1.2

[13] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic, "Recovering architectural design decisions," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 95–9509. 1.2

[14] G. Viviani, M. Famelis, X. Xia, C. Janik-Jones, and G. C. Murphy, "Locating latent design information in developer discussions: A study on pull requests," *IEEE Transactions on Software Engineering*, vol. 47, no. 7, pp. 1402–1413, 2021. 1.2

[15] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005. 1.2

[16] H. Petroski, *To engineer is human: The role of failure in successful design*. St Martins Press, 1985. 1.2

[17] A. Shahrokni and R. Feldt, "A systematic review of software robustness," *Information and Software Technology*, vol. 55, no. 1, pp. 1–17, 2013, special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584912001048 1.3, 1.3, 1.3, 5.1

[18] N. Laranjeiro, J. a. Agnelo, and J. Bernardino, "A systematic review on software robustness assessment," *ACM Comput. Surv.*, vol. 54, no. 4, may 2021. [Online]. Available: https://doi.org/10.1145/3448977 1.3, 1.3, 5.1

[19] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Camara, R. Calinescu, M. B. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, H. A. Müller, R. Rouvoy, C. M. F. Rubira, E. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. M. Villegas, T. Vogel, and F. Zambonelli, "Software engineering for self-adaptive systems: Research challenges in the provision of assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds. Cham: Springer International Publishing, 2017, pp. 3–30. 1.3, 1.3, 5.2

[20] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," *Formal Methods in System Design*, vol. 38, no. 3, pp. 223–262, 2011. 1.3, 1.3, 5.2

[21] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016. 1.3, 5.1

[22] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, feb 2016. [Online]. Available: https://doi.org/10.1145/2841425 1.3, 5.1

[23] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.456 1.3, 5.1

[24] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021. 1.3, 5.1

[25] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021. 1.3, 5.1

[26] J. Arcile, R. Devillers, H. Klaudel, W. Klaudel, and B. Woźna-Szcześniak, "Modeling and checking robustness of communicating autonomous vehicles," in *Distributed Computing and Artificial Intelligence, 14th International Conference*, S. Omatu, S. Rodríguez, G. Villarrubia, P. Faria, P. Sitek, and J. Prieto, Eds. Cham: Springer International Publishing, 2018, pp. 173–180. 1.3

[27] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011. 1.3

[28] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, vol. 15, pp. 281–312, 2014. 1.3, 5.1

[29] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006. 1.3

[30] M. Bishop and C. Elliott, "Robust programming by example," in *Information Assurance and Security Education and Training*, R. C. Dodge and L. Futcher, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 140–147. 1.3

[31] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2008. 1.3

[32] J. Bloch, *Effective Java: Best practices for the Java Platform*. Addison-Wesley Professional, 2017. 1.3

[33] T. Eifler, F. Campean, S. Husung, and B. Schleich, "Perspectives on robust design – an overview of challenges and research areas across industry fields," *Proceedings of the Design Society*, vol. 3, p. 2885–2894, 2023. 1.3

[34] M. Shafique, M. Naseer, T. Theocharides, C. Kyrkou, O. Mutlu, L. Orosa, and J. Choi, "Robust machine learning systems: Challenges,current trends, perspectives, and the road ahead," *IEEE Design & Test*, vol. 37, no. 2, pp. 30–57, 2020. 1.3

[35] F. Hu, Y. Lu, A. V. Vasilakos, Q. Hao, R. Ma, Y. Patil, T. Zhang, J. Lu, X. Li, and N. N. Xiong, "Robust cyber–physical systems: Concept, models, and implementation," *Future Generation Computer Systems*, vol. 56, pp. 449–475, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/

pii/S0167739X15002071 1.3

[36] S.-W. Cheng, D. Garlan, and B. Schmerl, "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. 1.3

[37] C. Zhang, D. Garlan, and E. Kang, "A behavioral notion of robustness for software systems," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, p. 1–12. 1.5.1

[38] A. van Lamsweerde, R. Darimont, and E. Letier, "Managing conflicts in goal-driven requirements engineering," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 908–926, 1998. 1.5.2, 2.4.2, 4.1, 4.4.1, 4.5.1, 5.2

[39] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 3rd ed. Springer, Cham, 2021. 1.5.2, 3.1, 3.2, 3.3, 3.3, 3.5.1, 5.2, 6.2

[40] C. Zhang, T. Saluja, R. Meira-Góes, M. Bolton, D. Garlan, and E. Kang, "Robustification of behavioral designs against environmental deviations," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023. 1.5.2

[41] C. Zhang, P. Kapoor, I. Dardik, L. Cui, R. Meira-Góes, D. Garlan, and E. Kang, "Constrained ltl specification learning from examples," 2025, under review. 1.5.2, 4.1, 4.3.4, 4.5.3, 4.6.1, 4.6.4, 6.3

[42] C. Zhang, R. Wagner, P. Orvalho, D. Garlan, V. Manquinho, R. Martins, and E. Kang, "Alloymax: bringing maximum satisfaction to relational specifications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 155–167. [Online]. Available: https://doi.org/10.1145/3468264.3468587 1.5.2, 4.5.3

[43] C. Zhang, I. Dardik, R. Meira-Góes, D. Garlan, and E. Kang, "Fortis: A tool for analysis and repair of robust software systems," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2023. 1.5.3, 2.6.2

[44] M. L. Bolton, "A task-based taxonomy of erroneous human behavior," *International Journal of Human-Computer Studies*, vol. 108, pp. 105–121, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1071581917301003 2.1

[45] J. Reason, *Human Error*. New York: Cambridge University Press, 1990. 2.1, 2.2, 2.6.4, 2.6.4

[46] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993. 2.2, 2.2

[47] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2001. 2.2

[48] J. Magee and J. Kramer, *State models and java programs.* wiley Hoboken, 1999. 2.2, 2.6.2, 1, 4.6.3, 6.3

[49] J. Bergstra, A. Ponse, and S. Smolka, Eds., *Handbook of Process Algebra.* Amsterdam: Elsevier Science, 2001. 2.3

[50] D. Alrajeh, A. Cailliau, and A. van Lamsweerde, "Adapting requirements models to varying environments," in *International Conference on Software Engineering (ICSE).* ACM, 2020, pp. 50–61. 2.4.2, 5.2

[51] T. Buckworth, D. Alrajeh, J. Kramer, and S. Uchitel, "Adapting specifications for reactive controllers," in *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2023, pp. 1–12. 2.4.2, 4.4.2, 5.2, 6.3

[52] S. Chu, E. Shedden, C. Zhang, R. Meira-Góes, G. A. Moreno, D. Garlan, and E. Kang, "Runtime resolution of feature interactions through adaptive requirement weakening," in *Proceedings of the 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '23, 2023. 2.4.2, 5.2

[53] D. Jackson and E. Kang, "Separation of concerns for dependable software design," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 173–176. [Online]. Available: https://doi.org/10.1145/1882362.1882399 2.4.2

[54] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu, *Compositional Reasoning.* Springer International Publishing, 2018, pp. 345–383. 2.5.2

[55] C. B. Jones, "Specification and design of (parallel) programs," in *9th IFIP World Computer Congress (Information Processing 83).* Newcastle University, 1983. 2.5.2

[56] D. Giannakopoulou, C. Pasareanu, and H. Barringer, "Assumption generation for software component verification," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, 2002, pp. 3–12. 2.5.2, 6.1

[57] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* USA: Addison-Wesley Longman Publishing Co., Inc., 2006. 3

[58] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 487–495. 2.6.2, 6.2

[59] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, 2000. 2.6.3, 2.6.3

[60] M. L. Bolton and E. J. Bass, "Enhanced operator function model: A generic human task behavior modeling language," in *2009 IEEE International Conference on Systems, Man and Cybernetics.* IEEE, 2009, pp. 2904–2911. 2.6.4, 2.6.4

[61] E. Hollnagel, *Cognitive Reliability and Error Analysis Method (CREAM).* Elsevier Science, 1998. 2.6.4

[62] J. Annett and N. A. Stanton, *Task Analysis.* CRC Press, 2000. 2.6.4

[63] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, "Generating phenotypical erroneous human behavior to evaluate human–automation interaction using model checking," *International Journal of Human-Computer Studies*, vol. 70, no. 11, pp. 888–906, 2012. 2.6.4

[64] M. L. Bolton and E. J. Bass, "Generating erroneous human behavior from strategic knowledge in task models and evaluating its impact on system safety with model checking," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 6, pp. 1314–1327, 2013. 2.6.4

[65] T. T. Tun, A. Bennaceur, and B. Nuseibeh, "OASIS: Weakening user obligations for security-critical systems," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 113–124. 2.6.5, 3.7.4, 5.2, 6.2, 6.4

[66] U.S. Attorney's Office Eastern District of Kentucky, "Clay county officials and residents convicted on racketeering and voter fraud charges," Mar 2010. [Online]. Available: https://archives.fbi.gov/archives/louisville/press-releases/2010/lo032510.htm 2.6.5, 3.7.3

[67] D. Sempreboni and L. Viganò, "X-men: A mutation-based approach for the formal analysis of security ceremonies," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 87–104. 2.6.5

[68] M. L. Bolton and E. J. Bass, "Evaluating human-automation interaction using task analytic behavior models, strategic knowledge-based erroneous human behavior generation, and model checking," in *2011 IEEE International Conference on Systems, Man, and Cybernetics*, 2011, pp. 1788–1794. 2.6.5

[69] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, pp. 82:1–82:37, 2018. 2.7

[70] Y. Collette and P. Siarry, *Multiobjective Optimization: Principles and Case Studies*, ser. Decision Engineering. Springer Berlin Heidelberg, 2013. 3.1, 3.4.4, 3.5.2

[71] M. T. Emmerich and A. H. Deutz, "A tutorial on multiobjective optimization: fundamentals and evolutionary methods," *Natural computing*, vol. 17, pp. 585–609, 2018. 3.4.4

[72] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012. 3.4.4

[73] J. N. Tsitsiklis, "On the control of discrete-event dynamical systems," in *26th IEEE Conference on Decision and Control*, vol. 26, 1987, pp. 419–422. 13

[74] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 179–190. [Online]. Available: https://doi.org/10.1145/75277.75293 13, 6.2

[75] R. Su and W. M. Wonham, "Supervisor reduction for discrete-event systems," *Dis-*

crete Event Dynamic Systems, vol. 14, no. 1, pp. 31–53, 2004. 3.6.1

[76] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica–an efficient tool for large-scale discrete event systems," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5794–5799, 2017, 20th IFAC World Congress. 3.7.2, 3.7.4

[77] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, p. 257–266, sep 2003. [Online]. Available: https://doi.org/10.1145/949952.940106 4.1, 4.3.3, 4.4.3

[78] D. Neider and I. Gavran, "Learning linear temporal properties," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–10. 4.1, 4.3.4, 4.5.3, 4.6.4

[79] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57. 4.3.1

[80] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI '13. AAAI Press, 2013, p. 854–860. 4.3.2, 4.4.3

[81] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *Verification, Model Checking, and Abstract Interpretation*, E. A. Emerson and K. S. Namjoshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–380. 4.4.2, 6.2

[82] S. Bansal, Y. Li, L. M. Tabajara, M. Y. Vardi, and A. Wells, "Model checking strategies from synthesis over finite traces," in *Automated Technology for Verification and Analysis*, É. André and J. Sun, Eds. Cham: Springer Nature Switzerland, 2023, pp. 227–247. 4.4.3, 4.4.3

[83] M. Daniele, F. Giunchiglia, and M. Y. Vardi, "Improved automata generation for linear temporal logic," in *Computer Aided Verification*, N. Halbwachs and D. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 249–260. 1

[84] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, *Simple On-the-fly Automatic Verification of Linear Temporal Logic*. Boston, MA: Springer US, 1996, pp. 3–18. [Online]. Available: https://doi.org/10.1007/978-0-387-34892-6_1 1

[85] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of ltl formulae to büchi automata," in *Formal Techniques for Networked and Distributed Sytems — FORTE 2002*, D. A. Peled and M. Y. Vardi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 308–326. 1

[86] S. Lutz, D. Neider, and R. Roy, "Specification sketching for linear temporal logic," in *Automated Technology for Verification and Analysis*, É. André and J. Sun, Eds. Cham: Springer Nature Switzerland, 2023, pp. 26–48. 4.5.3

[87] F. Bacchus, M. Järvisalo, and R. Martins, "Maximum Satisfiability," in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, ch. 24, pp. 929 – 991. 4.6.1

[88] R. Martins, V. Manquinho, and I. Lynce, "Open-wbo: A modular maxsat solver,,"

in *Theory and Applications of Satisfiability Testing – SAT 2014*, C. Sinz and U. Egly, Eds. Cham: Springer International Publishing, 2014, pp. 438–445. 4.6.4

[89] P. Tabuada, S. Y. Caliskan, M. Rungger, and R. Majumdar, "Towards robustness for cyber-physical systems," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3151–3163, 2014. 5.1, 5.2

[90] T. A. Henzinger, J. Otop, and R. Samanta, "Lipschitz Robustness of Finite-state Transducers," in *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), V. Raman and S. P. Suresh, Eds., vol. 29. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 431–443. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4861 5.1, 5.2

[91] ——, "Lipschitz robustness of timed i/o systems," in *Verification, Model Checking, and Abstract Interpretation*, B. Jobstmann and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 250–267. 5.1

[92] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann, "Specification-centered robustness," in *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, 2011, pp. 176–185. 5.1, 5.2

[93] P. Tabuada and D. Neider, "Robust linear temporal logic," 2015. 5.1, 6.3

[94] S. P. Nayak, D. Neider, R. Roy, and M. Zimmermann, "Robust computation tree logic," in *NASA Formal Methods*, J. V. Deshmukh, K. Havelund, and I. Perez, Eds. Cham: Springer International Publishing, 2022, pp. 538–556. 5.1

[95] J. Rasmussen, "Risk management in a dynamic society: a modelling problem," *Safety Science*, vol. 27, no. 2, pp. 183–213, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925753597000520 5.1

[96] J. Petke, D. Clark, and W. B. Langdon, "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*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1475–1478. [Online]. Available: https://doi.org/10.1145/3468264.3473133 5.1

[97] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann, "Robustness in the presence of liveness," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 410–424. 5.2

[98] T. Kobayashi, R. Salay, I. Hasuo, K. Czarnecki, F. Ishikawa, and S.-y. Katsumata, "Robustifying controller specifications of cyber-physical systems against perceptual uncertainty," in *NASA Formal Methods*, A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, Eds. Cham: Springer International Publishing, 2021, pp. 198–213. 5.2

[99] A. Easwaran, S. Kannan, and O. Sokolsky, "Steering of discrete event systems: Control theory approach," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 4, pp. 21–39, 2006, proceedings of the Fifth Workshop on Runtime Verification (RV 2005). 5.2

[100] Y. Falcone, J.-C. Fernandez, and L. Mounier, "What can you verify and enforce at runtime?" *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 349–382, 2012. 5.2

[101] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. 5.2

[102] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004. 5.2

[103] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "Morph: a reference architecture for configuration and behaviour self-adaptation," in *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, ser. CTSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 9–16. [Online]. Available: https://doi.org/10.1145/2804337.2804339 5.2

[104] M. N. A. Islam, J. Cleland-Huang, and M. Vierhauser, "Adam: Adaptive monitoring of runtime anomalies in small uncrewed aerial systems," in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 44–55. [Online]. Available: https://doi.org/10.1145/3643915.3644092 5.2

[105] T. Brand and H. Giese, "Towards generic adaptive monitoring," in *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2018, pp. 156–161. 5.2

[106] J. Ehlers and W. Hasselbring, "A self-adaptive monitoring framework for component-based software systems," in *Software Architecture*, I. Crnkovic, V. Gruhn, and M. Book, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 278–286. 5.2

[107] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas, *DYNAMICO: A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 265–293. [Online]. Available: https://doi.org/10.1007/978-3-642-35813-5_11 5.2

[108] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing model checking in verification by ai techniques," *Artificial Intelligence*, vol. 112, no. 1, pp. 57–104, 1999. 5.2

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

Berlin Heidelberg, 2010, pp. 31–40. 5.2

[110] G. Chatzieleftheriou, B. Bonakdarpour, S. A. Smolka, and P. Katsaros, "Abstract model repair," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 341–355. 5.2

[111] Y. Ding and Y. Zhang, "A logic approach for LTL system modification," in *Foundations of Intelligent Systems*, M.-S. Hacid, N. V. Murray, Z. W. Raś, and S. Tsumoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 435–444. 5.2

[112] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *2009 17th IEEE International Requirements Engineering Conference*, 2009, pp. 79–88. 5.2

[113] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*, 1st ed. Wiley Publishing, 2009. 5.2

[114] N. D'Ippolito, V. A. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel, "Hope for the best, prepare for the worst: multi-tier control for adaptive systems," in *36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 688–699. 5.2

[115] M. Herlihy and J. Wing, "Specifying security constraints with relaxation lattices," in *Proceedings of the Computer Security Foundations Workshop II,*, 1989, pp. 47–53. 5.2

[116] N. A. Lynch and M. R. Tuttle, *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988. 6.1

[117] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9. New York, NY, USA: Association for Computing Machinery, 2001, p. 109–120. [Online]. Available: https://doi.org/10.1145/503209.503226 6.1

[118] M. Emmi, D. Giannakopoulou, and C. S. Păsăreanu, "Assume-guarantee verification for interface automata," in *FM 2008: Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 116–131. 6.1

[119] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585 6.1

[120] C. Baier, L. de Alfaro, V. Forejt, and M. Kwiatkowska, *Model Checking Probabilistic Systems*. Cham: Springer International Publishing, 2018, pp. 963–999. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_28 6.1, 6.5

[121] S. Maoz and J. O. Ringert, "GR(1) synthesis for LTL specification patterns," in *Proceedings of Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 96–106. 6.2

[122] B. Steffen, F. Howar, and M. Merten, *Introduction to Active Automata Learning from a Practical Perspective*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011,

pp. 256–296. [Online]. Available: https://doi.org/10.1007/978-3-642-21455-4_8 6.2

[123] R. Alur, S. Moarref, and U. Topcu, "Compositional synthesis of reactive controllers for multi-agent systems," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 251–269. 6.2

[124] T. Anevlavis, D. Neider, M. Philippe, and P. Tabuada, "Evrostos: The rltl verifier," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 218–223. [Online]. Available: https://doi.org/10.1145/3302504.3311812 6.3

[125] A. Farzan, Y.-F. Chen, E. M. Clarke, Y.-K. Tsay, and B.-Y. Wang, "Extending automated compositional verification to the full class of omega-regular languages," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 2–17. 6.4

[126] A. Biere, C. Artho, and V. Schuppan, "Liveness checking as safety checking," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2, pp. 160–177, 2002, fMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104804109 6.4

[127] E. Letier, J. Kramer, J. Magee, and S. Uchitel, "Fluent temporal logic for discrete-time event-based models," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 70–79. [Online]. Available: https://doi.org/10.1145/1081706.1081719 6.4

[128] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 52–59. 6.4

[129] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100270, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013719302527 6.5

[130] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*, 2nd ed. The MIT Press, 2016. 6.5

[131] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3158668 6.5

[132] C. Zhang, P. Kapoor, E. Kang, R. Meira-Goes, D. Garlan, A. Ganlath, S. Mishra, and N. Ammar, "Tolerance of reinforcement learning controllers against deviations in cyber physical systems," 2024. [Online]. Available: https://arxiv.org/abs/2406.17066 6.5