# AdaStress: Adaptive Stress Testing and Interpretable Categorization for Safety-Critical Systems

*Submitted in partial fulfillment of the requirements for*

*the degree of*

*Doctor of Philosophy*

*in*

*Department of Electrical and Computer Engineering*

Ritchie Lee

B.S., Electrical and Computer Engineering, University of Waterloo
M.S., Aeronautics and Astronautics Engineering, Stanford University

Carnegie Mellon University
Pittsburgh, PA

May 2019

© Ritchie Lee, 2019
All rights reserved

## Acknowledgements

First of all, I must thank my advisors Ole Mengshoel and Mykel Kochenderfer for their tremendous support and encouragement while I undertook this work. I have benefitted immensely from our interactions over the past several years and I have learned a great deal from both of you both academically and otherwise. I would like to thank my two other thesis committee members Guillaume Brat and Corina Pasareanu who have been extremely supportive and have provided invaluable feedback throughout the years.

I am grateful to my co-authors and collaborators on the ACAS X team: Neal Suchy, Josh Silbermann, Ryan Gardner, Daniel Genin, Anshu Saksena, Michael Owen, Wes Olson, Cindy McLain, Jeff Brush, and others who have helped me with my experiments, helped me understand the ins and outs of ACAS X, and provided valuable reviews for my papers.

I am grateful to my co-authors and collaborators on the ATTRACTOR team: Natalia Alexandrov, Danette Allen, Javier Puig-Navarro, and others who have informed the design of experiments, provided software, and provided valuable feedback for my papers.

I would like to thank my friends, colleagues, and collaborators at NASA Ames: Adrian Agogino, Dimitra Giannakopoulou, Corey Ippolito, Ewen Denney, Ganesh Pai, Edward Balaban, Johann Schumann, and others for their support during my Ph.D. and especially to those who helped review my papers.

I would like to thank my friends at Stanford SISL, especially Tim Wheeler, Youngjun Kim, Zach Sunberg, Rachel Tompa, Mark Koren, Saud Alsaif, and Katie Driggs-Campbell, for being exceptionally welcoming and open, and selflessly sharing code and providing help.

I would like to thank my friends, colleagues, and mentors at CMU-SV: Abe Ishihara, Bob Iannucci, Aniruddha Basak, Ming Zeng, Ervin Teng, Abhinav Jauhri, David Cohen, Tong Yu, and others, for their generous support and company. They have greatly improved my experience at CMU-SV.

Finally, I wish to thank my wife, Heidi Lee, for her unwavering support and encouragement throughout the years even as she juggled work and being a mother to our son Hugo and daughter Alice, both born during the Ph.D. I am also very grateful to my parents and mother-in-law for flying in to help at home every time I have needed it.

This work was supported by the Safe and Autonomous Systems Operations (SASO) Project and the ATTRACTOR Project under NASA Aeronautics Research Mission Directorate (ARMD) through the ISRDS-1 (NNA08CG83C) and ISRDS-2 (NNA14AA60C) contracts.

## Abstract

This thesis considers tools and techniques for the design-time validation of *cyber-physical* systems, where a software system interacts with or controls a physical system over time. We focus on safety-critical systems that may be fully or partially autonomous. The goal of the design-time validation is to identify and diagnose potential failures during system development so that issues can be addressed before they can manifest in operation. However, finding and analyzing failure scenarios in cyber-physical systems can be very challenging due to the size and complexity of the system, interactions with large environments, operation over time, black box and hidden states, rarity of failures, heterogeneous variable types, and difficulty in diagnosing failures.

This thesis presents AdaStress, a set of design-time validation tools for finding and analyzing the most likely failure scenarios of a safety-critical system. We present adaptive stress testing (AST), a framework for simulation-based stress testing to find the most likely path to a failure event. The key innovation in AST is to frame the search for the most likely failure scenarios as a sequential decision-making problem and then use reinforcement learning algorithms to adaptively search the scenarios. To handle systems with hidden state, we present an algorithm for AST, based on Monte Carlo tree search and pseudorandom seeds, that can be applied to test systems where the state is not fully observable. Furthermore, we present differential adaptive stress testing (DAST), an extension to AST. DAST compares the failure behavior of two systems. Specifically, DAST finds the most likely scenarios where a failure occurs with the system under test but not with a baseline system. This type of differential analysis is useful, for example, when choosing between two candidate systems or in regression testing. Lastly, grammar-based decision tree (GBDT) learning is an algorithm for automatically categorizing failure events based on their most relevant patterns. The algorithm combines a context-free grammar, temporal logic, and decision tree to produce categorizations with human-interpretable explanations.

We demonstrate AdaStress on two cyber-physical systems within aerospace. The first application analyzes prototypes of the next-generation Airborne Collision Avoidance System (ACAS X) in simulated aircraft encounters. We find, categorize, and analyze the most likely scenarios of near mid-air collisions (NMACs). We also perform differential studies comparing ACAS X to the existing Traffic Alert and Collision Avoidance System (TCAS). Our results give confidence that ACAS X offers a safety benefit over TCAS. The second application analyzes a prototype trajectory planning system for a small unmanned aircraft navigating through a three-dimensional maze. We find and analyze the most likely collision scenarios and planning failures. Our analysis identifies a variety of potential safety issues that include algorithmic robustness issues, emergent behaviors from interacting systems, and implementation bugs.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Emerging applications in artificial intelligence (AI), such as driverless cars, autonomous aircraft, and smart infrastructure, promise to revolutionize industries by being more convenient, more efficient, cheaper to operate, and always available. However, safety remains a major challenge due the safety-critical nature of the applications and the reliance of difficult-to-analyze components in critical roles of the system such as perception and decision-making. Indeed, failures of these *safety-critical* systems can have serious consequences including loss of life and property. As a result, it is imperative to have good analytic tools to ensure their safe operation. Ideally, safety issues should be uncovered and addressed as early in the life cycle as possible.

Design-time system verification and validation (V&V) aims to discover and diagnose issues while the system is still in development. Catching issues during the design phase allows them to be addressed early before the issues have had any opportunity to manifest in operation. Addressing issues early is also much more desirable in terms of cost, since the cost of resolving issues increases exponentially the later it is discovered in the life cycle. *Validation* evaluates whether the system adequately addresses operational needs, and is distinguished from *verification*, which evaluates whether a system was built matching its design specifications.

A primary concern of design-time V&V is analyzing failures to understand how the system can fail. While simple systems may have few and obvious failure modes, large complex systems can exhibit surprisingly complex failure modes resulting from emergent and subtle system behaviors. Good tools for discovering and analyzing failures are needed to assist safety engineers in understanding areas where the system may be vulnerable and in identifying potential issues. A good understanding of failures is important to the design, evaluation, and certification of safety-critical systems; and to informing decisions that can reduce the probability and impact of the failures.

However, finding and analyzing failure scenarios in safety-critical cyber-physical systems can be very challenging due to the size and complexity of cyber-physical systems, interactions with large environments, operation over time, black box and hidden states, rarity of failures, heterogeneous variable types, and the difficulty in diagnosing failures. Existing formal and simulation-based approaches do not adequately address these challenges.

One of the main contributions of this thesis is a novel stress testing approach, called adaptive stress testing (AST). The key innovation in AST is to frame the search for the most likely failure scenarios as a sequential decision-making problem, which can then be efficiently optimized using reinforcement learning algorithms. We show that this approach is much more effective at finding failure scenarios than traditional methods. Failure scenarios can then be assessed by domain experts to determine their operational significance and the experts can take appropriate action to mitigate the risk if necessary.

In this thesis, we primarily consider *cyber-physical systems*, where a software system interacts with or controls a physical system over time. We focus on safety-critical systems that may be fully or partially autonomous. We also focus on approaches that can be applied to systems containing AI components, which is where we see the strongest need. Nevertheless, the core ideas and algorithms proposed in this thesis are sufficiently general that they can also be applied to other types of systems, such as pure software systems and traditional control systems.

## 1.1 Applications

This thesis uses two cyber-physical system applications as case studies for motivating and demonstrating the algorithms proposed. The first is an aircraft collision avoidance application, where we analyze prototypes of the next-generation Airborne Collision Avoidance System (ACAS X). The second is an unmanned aerial vehicle (UAV) navigation application where we analyze a trajectory planning system for goal seeking and obstacle avoidance. These systems are both cyber-physical systems operating in a large environment over time and performing safety-critical tasks.

### 1.1.1 Aircraft Collision Avoidance

ACAS X is an aircraft collision avoidance system that operates onboard commercial aircraft. The system monitors the airspace around the aircraft and warns pilots of potential collisions with other aircraft. The system issues advisories that contain recommended maneuvers to the pilot. The pilot then responds to the advisories by maneuvering the aircraft to resolve the conflict. ACAS X was developed and tested by the

Federal Aviation Administration (FAA) and has recently been accepted by the RTCA[1] on September 20$^{th}$, 2018. The system replaces the existing Traffic Alert and Collision Avoidance System (TCAS) as the new international standard for airborne collision avoidance. This thesis work was performed on prototypes of ACAS X while the system was still being developed and tested. This work has contributed to the larger verification and validation efforts by the FAA, which ultimately led to the acceptance of ACAS X by the RTCA. This thesis presents our work in applying AdaStress to analyze prototypes of the ACAS X system in simulated aircraft encounters to discover and analyze scenarios of near mid-air collision (NMAC), where two aircraft come dangerously close to one another.

### 1.1.2 Trajectory Planning

The trajectory planner is a prototype system currently being developed as part of the Autonomy Teaming & TRAjectories for Complex Trusted Operational Reliability (ATTRACTOR) project at National Aeronautics and Space Administration (NASA) Langley. A future version of the system will be used onboard a UAV for navigating the vehicle toward a destination in the presence of obstacles. The UAV will be used as part of a search and rescue demonstration mission in a forest environment. Preliminary testing of the system is being performed in an indoor testbed, called *WireMaze*. The testbed is a three-dimensional cubic structure with wires strung across it at various positions and angles providing a three-dimensional obstacle course for the UAV. The UAV's task is to safely navigate from one side of the cube to the other side without colliding with the wires. This thesis uses the WireMaze scenario as the basis for analysis, applying AdaStress to discover potential issues that may arise during operation. We uncover a wide range of issues, including collisions, replanning issues, planning failures, and implementation bugs.

## 1.2 Challenges

Failure analysis in cyber-physical systems faces a number of challenges. This section describes some of these challenges.

**Size and Complexity of Cyber-Physical Systems.** Cyber-physical systems can be very large, complex, and consist of many parts. AI components are particularly problematic in their size and complexity. For example, a deep neural network [1] can have billions of network parameters and an autonomous vehicle can have many of such networks for various purposes, such as image processing, lidar processing, dynamics modeling, and vehicle control. The large size and complexity of the system make the search space very large and unsmooth, which makes the search for failures very challenging.

[1]RTCA, Inc. was formerly known as the Radio Technical Commission for Aeronautics, but is now known simply as the RTCA.

**Interactions with Large Environments.** System-level failures, such as an autonomous car colliding with a pedestrian, cannot be discovered by analyzing the behavior of the system alone as the failures arise from the joint behavior and interactions of the system under test with its environment. As a result, analysis must be performed over the combined system consisting of both the system under test and its environment. Large stochastic operating environments can thus dramatically increase the size of the state space to be searched.

**Operation over Time.** Cyber-physical systems are systems that interact with physical processes and thus generally must operate over time. The view that cyber-physical systems are processes that unfold over time means that the search for failure scenarios must occur over state trajectories rather than single states. Indeed, the state history leading up to how the system entered a failure state gives much more information to safety engineers than just the failure state itself. Searching over trajectories leads to an exponential explosion of the search space and exhaustive consideration of all possible trajectories is generally intractable. In this thesis, we consider discrete time processes and thus the search for failure scenarios occurs over sequences of states.

**Black Box and Hidden State Components.** Cyber-physical systems can contain black box components such as software binaries, where the internal details are not available to the validation process. The presence of these components make the system difficult to analyze because they cannot be accurately characterized and the uncertainty can affect guarantees on the system. Furthermore, systems can maintain hidden state across interactions. Systems with hidden state require testing over sequences of input rather than a single input, which makes the system much harder to test.

**Failures are Rare.** Safety-critical systems are designed to be very safe and robust against failures. As a result, it may be very difficult or impossible to reach failures states.

**Heterogeneous Variable Types.** Cyber-physical systems often have system states that contain a mix of variable types. For example in an aircraft system, the airspeed and bank angle are real values, the autopilot mode is categorical, and whether the landing gear is deployed is Boolean. Analyzing systems with heterogeneous variable types can be very challenging. Heterogeneous state spaces lack a natural distance metric and many machine learning algorithms are not capable of handling such spaces.

**Diagnosing Failures.** Diagnosing failures can be very difficult because subtle issues can manifest in complex and completely unobvious ways. Failures can also arise from interactions between multiple underlying issues. Moreover, the high-dimensional state space and complex behaviors of cyber-physical systems can also obscure underlying causes making them difficult to observe in failure examples.

## 1.3 Approach

This thesis presents a collection of novel algorithms, called AdaStress, for analyzing failure events in cyber-physical systems. AdaStress aims to address the challenges mentioned in the previous section and is suitable for a broad class of systems. The algorithms in AdaStress can be separated into two functional categories: algorithms that find examples of failures and algorithms that help human experts diagnose failures.

Scalability is perhaps the biggest challenge to the validation and testing of cyber-physical systems. The size of the systems and their operating environments compounded by the search over temporal sequences means that the search space can be exponentially large, which makes exhaustive coverage intractable. Furthermore, because failures are rare and may be hard to reach, naive sampling methods can be impractically inefficient. AdaStress takes the approach of scaling the search using adaptive sampling. That is, we optimize where to sample next by learning from past samples and focus on the most promising areas of the search space. A sample can be valuable in two ways. It can (1) immediately reveal a failure, which is the goal of the search; or (2) reveal new information about the search space, which can help inform future sampling. Adaptive sampling is a core idea behind the adaptive stress testing (AST) framework, where we formulate the problem of finding the most likely failure scenarios as a sequential decision problem and then use reinforcement learning to optimize it.

Systems with hidden state are significantly more challenging to test and validate than those without. There are very few approaches that can be applied to these systems with hidden state. AdaStress handles software systems (or software simulations) with hidden states by assuming control of the pseudorandom number generator seed. The control of the seed not only converts a stochastic process into a deterministic process, but also gives the ability to recreate a system state even if it is unknown to us.

While AST can be very effective at finding failure examples, it can still be very difficult for a domain expert to diagnose the underlying cause from the examples. The reason is that cyber-physical systems can have a very high-dimensional state space and exhibit complex behavior, which can be very unintuitive. Another reason is that deep issues can manifest itself in complex and unobvious ways. Machine learning techniques are the most appropriate candidates to help analyze the data for recurring patterns. However, a core problem is how to generate results that are both useful and interpretable to a human expert. AdaStress addresses this problem by requiring the user to provide a grammar to the search algorithm. The grammar not only defines the search space but also provides a template for generating interpretable and domain-specific expressions. This idea of leveraging a grammar for aiding in interpretability is used in the grammar-based decision tree (GBDT) framework. A key capability of GBDT is to automatically categorize data, which both organizes the data according to similar patterns and produces explanations for them. The categories and their explanations can help human experts diagnose the failures.

AdaStress can fit very well with major iterative and non-iterative development processes. In the traditional *waterfall development model* [2, 3], which consists of a sequence of stages: requirements, design, implementation, verification, and maintenance, AdaStress can be extensively applied in the verification stage. We can use AdaStress to find the most likely failure scenarios and then analyze the failures to determine whether the nature of these failures is acceptable or not. In an iterative development process, such as the *spiral development model* [4], AdaStress can play an even larger role. The spiral model iterates between determining objectives, identifying and resolving risks, developing and testing, and planning the next iteration. In this model, AdaStress can be applied at every iteration to find the failure scenarios and categorize them for a human, so that the humans can make more informed planning and development decisions for the next iteration.

## 1.4 Contributions

Contributions are highlighted in the chapters in which they are presented, and Chapter 10 summarizes the specific contributions made throughout this thesis. This section provides a brief, high-level overview of the primary contributions.

This thesis presents AdaStress, a collection of algorithms for finding and analyzing failure events in safety-critical cyber-physical systems. Implementations of and experiments with these algorithms demonstrate the effectiveness of the algorithms in analyzing failures in large cyber-physical systems.

An important contribution of this thesis is adaptive stress testing (AST), a general framework for using sequential decision-making algorithms for finding the most likely path to a failure event. This framework initiates a novel research direction for scaling and improving the effectiveness of testing and validation algorithms.

This thesis presents a novel algorithm for stress testing black box and non-Markovian systems in the AST framework. The algorithm only requires access to the seed of the pseudorandom number generator, which makes it very broadly applicable. This contribution is significant because the algorithm has some very desirable characteristics: simple integration with existing simulators, can operate on black box systems, is scalable, can be applied to multi-time step systems, and supports systems with hidden state.

Another contribution of this thesis is differential adaptive stress testing (DAST), a novel approach for stress testing a system under test relative to a baseline system. DAST discovers failures that occur in the system under test but not in the baseline system, making it very useful for regression testing or comparing two candidate systems.

In addition to finding failure scenarios, this thesis contributes a novel framework for automatically categorizing failure examples. The framework, called grammar-based decision tree (GBDT), groups the failures according to their most relevant properties and also provides precise yet human-interpretable explanations for each category. This contribution is significant because it simultaneously addresses a number of key challenges: it provides precise human-interpretable explanations, supports heterogeneous features, supports multivariate time series data, and produces a hierarchical organization of the data for human consumption.

This thesis applies AdaStress to analyze NMACs in the next-generation Airborne Collision Avoidance System (ACAS X). While the system was still under development, we discussed the results of our NMAC analyses with the ACAS X team to inform their design decisions contributing to the verification and validation efforts of the ACAS X program. ACAS X has been accepted by the RTCA to replace TCAS as the new international standard for airborne collision avoidance. The RTCA is a not-for-profit organization that works with government regulatory bodies to develop aviation standards. We also apply AdaStress to analyze the trajectory planning algorithms used in the ATTRACTOR project, which aims to develop trustworthy autonomous systems. We discuss the issues that we find with the ATTRACTOR team so that the issues can be addressed in future versions of the system.

Finally, we contribute to the open source and scientific communities by making our implementation of AdaStress available as an open source Julia package, AdaStress.jl. The software package is available at `https://github.com/rcnlee/AdaStress.jl`.

## 1.5 Overview

This thesis presents AdaStress, a collection of algorithms for discovering and analyzing failures in cyber-physical systems. This chapter has motivated the need for better V&V algorithms for cyber-physical systems, and summarized contributions. The remainder of this thesis is structured as follows.

Chapter 2 provides an overview of the various approaches taken in the field of testing and validation of safety-critical systems. The purpose of this chapter is to motivate the approach taken in AdaStress and provide a broad context in which to understand the contributions made in this thesis.

Chapter 3 provides an introduction to sequential decision processes and approaches to their solution. This background material will serve as the basis for understanding the AST framework presented in Chapters 5 and 6. Chapter 4 provides an introduction to approaches for interpretability and explainability in machine learning. The purpose of this chapter is to provide background and motivate the GBDT approach presented in Chapter 7.

Chapter 5 presents adaptive stress testing (AST), a framework that formulates finding the most likely path to a failure as a sequential decision process allowing it to be solved by reinforcement learning algorithms. The chapter further presents a novel algorithm suitable for testing black box systems with hidden state. Chapter 6 presents differential adaptive stress testing (DAST), an extension to AST for stress testing relative to a baseline system. That is, this chapter presents a method for finding the most likely failure path that occurs in the system under test, but not in the baseline system.

Chapter 7 presents grammar-based decision tree (GBDT), an interpretable machine learning model for automatically categorizing failure scenarios. Each category is accompanied by a precise yet human-interpretable expression that uniquely describes the failure category assisting domain experts in identifying the key common patterns in the failure scenarios.

Chapter 8 describes the application of AdaStress to stress test the next-generation Airborne Collision Avoidance System (ACAS X). We first introduce the aircraft encounter scenario and the basic operation of ACAS X, then we describe our studies analyzing the NMAC behavior of the system. Our studies include applying AST to find NMAC scenarios in single-threat and multi-threat encounters, differential studies using DAST to compare NMACs in ACAS X and TCAS, and using GBDT to automatic categorize NMAC scenarios.

Chapter 9 describes the application of AdaStress to stress test a trajectory planning system. We first describe the basic UAV scenario and operation of the planner, then we describe the application of AST to discover a variety of issues in the planner.

Chapter 10 concludes the thesis with a summary of the key contributions and ideas for further research.

# Chapter 2

# Verification and Validation of Safety-Critical Systems

This chapter provides a broad overview of approaches to design-time verification and validation of safety-critical systems, including formal methods and simulation-based testing. The purpose of this chapter is to provide the context and motivation for the adaptive stress testing approach discussed in Chapter 5. Section 2.1 introduces the topic of verification and validation and presents an overview of the main approaches. Section 2.2 reviews formal verification methods including model checking and automated theorem proving. Section 2.3 reviews simulation-based testing approaches to which the proposed approach in this thesis belongs. Finally, Section 2.4 summarizes the chapter.

## 2.1 Introduction

Verification and validation (V&V) play a critical role in the development and certification processes of safety-critical systems. V&V methods check that a system meets requirements and that it fulfills its intended purpose. During the development process, V&V is applied to uncover bugs and reveal potential issues. The findings of V&V analysis play a critical role in informing the development path of the system. During certification, the results of V&V are used to show to a regulating authority that the system is safe and meets certification criteria. V&V inspires trust in the new system and makes clear the residual risk in its operation.

Figure 2.1 is an overview of the main verification and validation approaches. Existing methods can be broadly separated into two categories, formal verification and simulation-based testing. Formal verification uses mathematical or computational models to prove or exhaustively check properties, while simulation-based testing relies primarily on simulation models and sampling. The following sections briefly describe the most relevant methods in each category presented in Figure 2.1. Adaptive stress testing (AST) (to be introduced in Chapter 5) and the other methods proposed in this thesis belong to simulation-based testing.

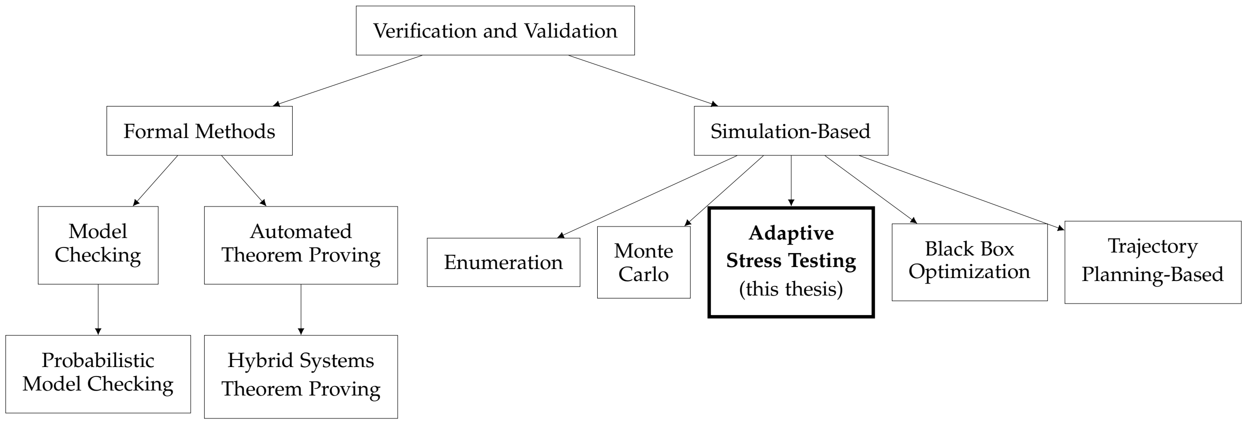

Verification and Validation
Formal Methods
Simulation-Based
Model Checking
Automated Theorem Proving
Probabilistic Model Checking
Hybrid Systems Theorem Proving
Enumeration
Monte Carlo
Adaptive Stress Testing (this thesis)
Black Box Optimization
Trajectory Planning-Based

Figure 2.1: Categories of verification and validation approaches.

## 2.2 Formal Methods for V&V

Formal verification constructs a mathematical or computational model of the system and rigorously proves or exhaustively checks whether a safety property holds [5, 6]. The properties are expressed using a formal logic, such as linear temporal logic (LTL) [7]. Formal methods can provide a counterexample when a property does not hold. More importantly, these methods provide completeness guarantees over the entire model, i.e., they can conclude the absence of violations. The major challenge is scalability. Due to their exhaustive nature, they have difficulty scaling to systems with large and complex state spaces. Consequently, applications often make simplifying assumptions, use simplified models, and test subsets of the system [8, 9, 10].

**Model Checking.** Model checking is a computer-assisted method for the analysis of dynamical systems that can be modeled by state-transition systems [8]. Model checking determines whether a temporal logic property, such as those specified in computation tree logic* (CTL*) [11] or LTL [7], holds on computational models known as Kripke structures [8]. Model checking provides exhaustive coverage of the space of the model to determine whether the property holds. Due to its exhaustive nature, model checking algorithms encounter combinatorial explosions in the number of states and paths. As a result, model checking employs techniques for simplification, including abstraction [12, 13] and composition [14]. Probabilistic model checking (PMC) extends model checking to the probabilistic setting. PMC checks probabilistic properties, such as those in probabilistic computation tree logic (PCTL), on probabilistic models, such as Markov chains [9, 15, 16], probabilistic automata [17], and probabilistic hybrid automata [18]. PMC algorithms can compute minimal or maximal probabilities for a given property. They can also provide counterexamples when a property is violated. However, scalability is a major issue with PMC methods.

**Automated Theorem Proving.** Automated theorem proving (ATP) uses computer algorithms to automatically generate mathematical proofs [19]. Systems and assumptions are modeled in formal logic, and then ATP is applied to prove whether a property holds. If a proof is generated successfully, one can conclude with certainty that the property holds over the entire model. However, if the algorithm fails to generate a proof, then it is uncertain whether the property holds. Hybrid systems theorem proving (HSTP) is a variant of ATP based on differential dynamic logic, which is a real-valued, first-order dynamic logic for hybrid systems [10, 20]. A hybrid system model can capture both continuous and discrete dynamic behavior. The continuous behavior is described by a differential equation and the discrete behavior is described by a state machine or automaton. The major challenges in ATP are scalability and that current ATP methods still require significant human assistance.

## 2.3 Simulation-Based Testing for V&V

The second major category of V&V approaches relies on simulation. Simulation models have very few modeling requirements. They only require a transition model of the system that can produce the next state or samples of the next state if the system is stochastic. Simulation-based testing methods are black box methods and do not require or leverage the internal details of the system. As a result, simulators can be very simple models or large sophisticated models. They can even directly embed software or call external programs.

Simulation-based testing methods differ primarily in how test scenarios are chosen. Because simulations are processes, test cases are fundamentally temporal, consisting of sequences of test inputs. For example, a wind disturbance can vary over time as the simulation evolves. The rest of this section describes the main approaches to simulation-based testing.

**Enumeration.** The most straightforward and brute force approach is to enumerate and test the simulation at all combinations of test inputs. Simulations with real-valued states can be discretized. This approach can work well for small simulations with short time horizons. However, it quickly becomes intractable due to a combinatorial explosion in state and time. One approach to overcoming this combinatorial explosion is to sweep over a low-dimensional parametric model instead, relying on the low-dimensional model to generate the original test inputs [21]. This approach requires assumptions and domain knowledge to define a suitable low-dimensional model. While this method can effectively test a large portion of the simulation space, rare corner case failures can be easily missed due to the abstraction.

**Monte Carlo.** An alternative approach to enumeration is to run simulations using a stochastic model of the system's operating environment [22, 23]. Sampling from stochastic models naturally focuses on the test scenarios with the highest likelihood of occurrence. The simulations produce sequences of states and then the sequences are checked for failures. Because sampling favors likelihood and does not optimize for failures, it can take a very large number of samples to encounter the correct sequence and combination of inputs to trigger a rare failure.

**Black Box Optimization.** One approach to explicitly search for failure events is to formulate it as a black box global optimization problem, where the parameters are the test inputs and the objective is to generate failures. A variety of black box optimization algorithms can been applied, including genetic algorithm (GA) [24, 25], cross-entropy method [26], simulated annealing [27], ant colony optimization [27, 28], and stochastic local search [29]. The exact formulation of the black box optimization problem can vary, such as in how the search parameters and the objective functions are defined. Some approaches try to maximize the probability of sampling a failure while others search for individual failure scenarios. These approaches improve upon direct Monte Carlo sampling by explicitly optimizing for failure events. However, performance can be limited, because they do not leverage the sequential structure of the problem.

**Trajectory Planning-Based.** An alternative approach is to formulate the problem as a trajectory planning problem and apply a trajectory optimization algorithm such as rapidly-exploring random tree (RRT) [30]. The RRT approach involves growing a tree from an initial state to a failure state by sampling a random point in the state space and growing the tree toward that target point. In the limit of infinite sample points, the tree can reach any point in the reachable search space. Crucially, RRT depends on the ability to "steer" the system transition towards a target point in the state space, which may not be possible in systems with complex transition behavior. Furthermore, the procedure requires evaluating the closeness of two points in state space to determine which node in the tree to expand. In its original context, RRT was applied to trajectory planning in physical spaces, which are Euclidean. When the state space is large and contains variables with mixed units and scales, then it is unclear what distance metric to use. Lastly, because the search requires directly operating on the state, this method cannot be used on simulators with hidden state.

## 2.4 Summary

This chapter reviewed the main approaches to verification and validation of safety-critical systems. Formal verification approaches, such as probabilistic model checking and hybrid systems theorem proving, rely on more constrained models, and are more rigorous. These factors allow them to make strong guarantees about properties such as the absence of failures. However, formal methods have difficulty scaling to large systems and their environments. Simulation-based testing requires only a simulator of the system and its environment, which allows more sophisticated and higher-fidelity models to be used. They are also able to scale to much larger systems. These methods can be used to find examples of failure scenarios if they exist, but cannot guarantee the absence of failures.

Amongst the simulation-based methods, enumeration, Monte Carlo, and black box optimization do not leverage the sequential structure of the problem. As a result, the search performance can be very poor or depend heavily on domain knowledge. Trajectory planning-based methods do leverage the sequential nature of the problem by formulating the problem as a trajectory planning problem. However, due to their reliance on RRT search algorithms, they require the ability to "steer" the simulator and compute distances, which is inappropriate for systems with large state spaces, complex transition behavior, and hidden state (non-Markovian) simulators. Thus, we are left with adaptive stress testing as the most viable approach among the ones shown in Figure 2.1.

# Chapter 3

# Decision Processes and Reinforcement Learning

This chapter introduces several fundamental topics in decision theory and reinforcement learning. We present Markov decision processes, partially observable Markov decision processes, reinforcement learning, and Monte Carlo tree search. The purpose of this chapter is provide the necessary background concepts in decision theory and reinforcement learning for understanding adaptive stress testing, a core contribution presented in Chapter 5 of this thesis. Additionally, this chapter introduces notation and terminology used in later chapters of this thesis. In Chapter 5, we will show how stress testing can be mapped to the problem of solving a Markov decision process.

## 3.1 Markov Decision Process

A *sequential decision process* is a mathematical framework for modeling situations where an agent makes a sequence of decisions in an environment to maximize utility derived from the rewards received at each time step [31]. The agent's actions interact with the environment which in general can be stochastic. If the environment is known and its state is fully observable, then the problem can be formulated as a Markov decision process (MDP). We first introduce MDP for finite state and actions, and then we extend the discussion to MDPs with continuous states and actions. Formally, an MDP is a 5-tuple $\langle S, A, P, R, \gamma \rangle$, where $S$ is a set of states; $A$ is a set of actions; $P$ is the transition probability function, where $P(s' \mid s, a)$ is the probability of choosing action $a \in A$ in state $s \in S$ and transitioning to next state $s' \in S$; and $R$ is the reward function, where $R(s, a, s')$ is the reward for taking $a$ in $s$ and transitioning to $s'$. We define the transition function $T$ for convenience, where $T(s, a)$ samples the next state $s'$ from the distribution $P(s' \mid s, a)$. The parameter $\gamma \in [0, 1]$ is the discount factor that governs how much to discount the value of future rewards. In an MDP, the agent chooses an action $a$ at time $t$ based on observing the current state $s$. The system evolves probabilistically to the next state $s'$ according to the transition function $T(s, a)$.

The agent then receives reward $r$ for the transition. The process is illustrated in Figure 3.1. In the figure, circular nodes denote random variables, diamond nodes denote reward values that are evaluated and to be maximized, and the square nodes denote the actions that the agent can choose. Subscript denotes time. The assumption that the transition function depends only on the current state and action is known as the *Markov property*.

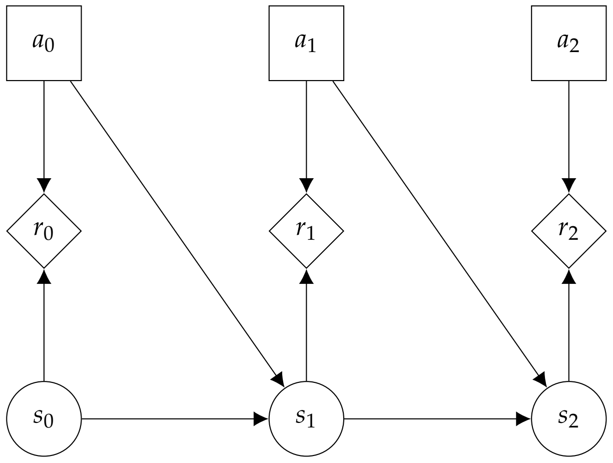


Figure 3.1: Markov decision process problem structure.

In a *finite-horizon* MDP, where the agent takes a finite number of steps, the *utility* $U$ of the agent is the sum of discounted rewards given by $\sum_{t=0}^{t_{end}} \gamma^t r_t$. Because the utility may be stochastic, the agent ultimately wishes to maximize expected utility. It is common in finite-horizon problems to set $\gamma = 1$ so that the utility is simply the sum of rewards. The two applications discussed in this thesis are both finite-horizon problems and we set $\gamma = 1$ in both cases. The agent chooses actions according to its *policy* $\pi$ which is a function of state $s$. The action is given by $a = \pi(s)$. While the policy can be stochastic in general, this thesis considers only deterministic policies.

The expected utility of executing $\pi$ from state $s$ is denoted $V^\pi(s)$. In the context of MDPs, $V^\pi$ is often referred to as the *value function*. An *optimal policy* $\pi^*$ is a policy that maximizes expected utility:

$$\pi^*(s) = \operatorname*{argmax}_{\pi} V^\pi(s) \tag{3.1}$$

for all states $s$.

**Bellman Equation.** The value function $V^\pi$ can be determined using *dynamic programming*, which is a computational technique for solving a complicated problem by recursively breaking it down into simpler sub-problems [32]. The value function $V^\pi$ can be determined by solving the *Bellman equation*:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^\pi(s') \tag{3.2}$$

for all states $s$. Similarly, the value function of the optimal policy $V^{\pi^*}$ can be determined by solving the *Bellman optimality equation*:

$$V^{\pi^*}(s) = max_a\{R(s,a) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^{\pi^*}(s') \tag{3.3}$$

for all states $s$. The Bellman equations are used as the basis of many algorithms for solving MDPs [31, 33].

**Multivariate Heterogeneous States and Actions.** Cyber-physical systems involve interactions with physical processes and thus are often most naturally modeled using multivariate heterogeneous states and actions. That is, the states and actions are vectors where each dimension can be continuous or finite valued. Indeed, the MDPs in this thesis are of this type (i.e., those in Chapter 8 and Chapter 9). In the multivariate heterogeneous MDP setting, $S$ and $A$ become infinite sets, and $P$ becomes a probability density function. Other aspects of the MDP formulation above remain the same. The Bellman equations hold for multivariate heterogeneous states and actions, however the sum over next states becomes an integral over next states.

## 3.2 Partially Observable Markov Decision Process

In cases where the underlying process is Markov, but the agent cannot fully resolve its state based on its observation, the problem is a partially observable Markov decision process (POMDP) [31]. Formally, a POMDP is a 7-tuple $\langle S, A, P, R, \Omega, O, \gamma \rangle$, where $S$, $A$, $P$, $R$, and $\gamma$ is defined as in an MDP; $\Omega$ is the set of possible observations; and $O$ is the observation function, where $O(o \mid s)$ gives the probability of observing $o \in \Omega$ in state $s$. Instead of choosing $a$ based on observing the state $s$ as in an MDP, the agent chooses $a$ based on only observing $o \in O$. While the observation $o$ depends on $s$, it does not contain enough information for the agent to fully determine what state it is in. Multiple states $s \in S$ can produce the same $o$. The POMDP is illustrated in Figure 3.2.

## 3.3 Reinforcement Learning

Reinforcement learning (RL) algorithms can optimize sequential decision problems through sampling of the transition and reward functions alone without the need to leverage a full probabilitic transition model of the environment [33, 34]. Instead, reinforcement learning algorithms rely on a generative model $G$, which represents all the information about the state transitions and the rewards. To transition from one time step to the next, samples from $G$ are drawn, which gives $(s', r) \sim G(s, a)$. The ability to rely only on samples from a generative model makes reinforcement learning algorithms very suitable for black box applications, where the internal details of the environment are not available to the learning agent, but the

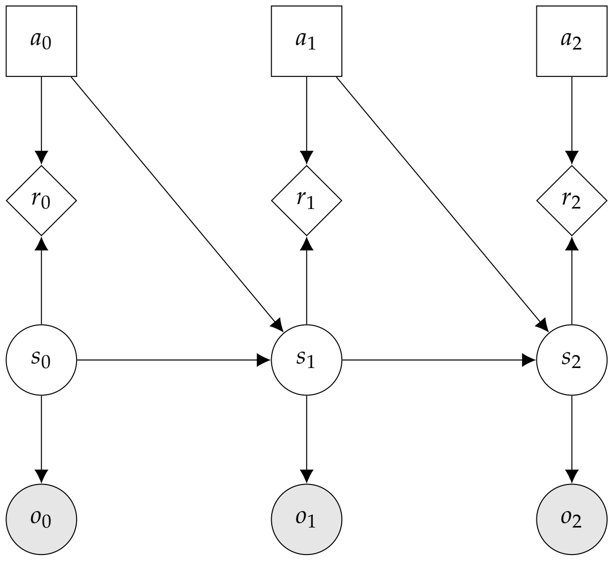


Figure 3.2: Partially observable Markov decision process (POMDP) problem structure.

environment can be sampled. Since the agent does not have a full model, it must simultaneously explore the space as it optimizes its decision-making behavior. *Model-based reinforcement learning* aims to estimate the missing information by learning a model of the transition function. *Model-free reinforcement learning* methods attempt to bypass the missing information by learning the optimal behaviors directly.

## 3.4 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that is based on *temporal differencing*, which is a bootstrapping approach that updates estimates of the value function toward other estimates of the value function [33, 35]. Rather than estimating the value of a state $V(s)$, Q-learning estimates the *state-action value* function $Q(s,a)$, or the *Q-function*, which is the expected utility of being in state $s$ and executing action $a$ and following a policy $\pi$ thereafter. The algorithm uses sample trajectories to incrementally update estimates of the Q-function according to

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)), \tag{3.4}$$

where $\alpha > 0$ is the *learning rate*, which is a parameter that governs how fast the system learns. Equation 3.4 incrementally updates the current estimate of the Q-value towards a value that is more consistent with the Bellman equation. Under mild assumptions, $Q$ is guaranteed to converge to the optimal state-action value function $Q^*$. Then the optimal policy is given by

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a). \tag{3.5}$$

The Q-learning algorithm is shown in Algorithm 1. The algorithm begins by initializing the Q-function $Q$ (line 2), which, for example, can be initialized randomly. The first loop initializes the current state $s$ and then enters the second loop that steps through the transitions of each episode. For each step in the episode, action $a$ is chosen based on the current estimate of $Q$ and some exploration strategy (line 6). An example of an exploration strategy is *$\epsilon$-greedy*, which chooses the action with the highest Q-value, i.e., $\operatorname{argmax}_a Q(s,a)$, with probability $1-\epsilon$ and a random action with probability $\epsilon$. The parameter $\epsilon \in (0,1)$ is typically chosen to be small. Then, a transition is sampled from $G$ (line 7) and the Q-learning incremental update rule is applied to update the estimate of the Q-function (line 8). Finally the state transitions to the next state (line 9) and the loop continues.

**Algorithm 1** Q-learning

```
1: function Q-LEARNING
2:     Initialize $Q$
3:     for each episode
4:         Initialize $s$
5:         for each step in episode
6:             Choose action $a$ based on $Q$ and some exploration strategy
7:             $(s', r) \sim G(s,a)$
8:             $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
9:             $s \leftarrow s'$
```

## 3.5 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a state-of-the-art heuristic search algorithm for optimizing sequential decision processes [36, 37]. MCTS incrementally builds a search tree using a combination of directed sampling based on estimates of the state-action value function and undirected sampling based on a fixed distribution, called *rollouts*. Each node of the tree is a single-state reinforcement learning agent, called a *multi-armed bandit* [38]. During the search, sampled paths from the simulator are used to incrementally estimate $Q(s,a)$ and the optimal policy at nodes in the tree. To account for the uncertainty in the estimates, which may lead to premature convergence, MCTS encourages exploration by adding an exploration term to the action value function to encourage choosing paths that have not been explored as often. The exploration term optimally balances the selection of the best action estimated so far with the need for exploration to improve the quality of current value estimates. By doing so, MCTS adaptively focuses the search towards more promising areas of the search space. The effect of the exploration term diminishes as the number of times the state is visited increases. For a detailed survey of MCTS, we encourage the reader to see [37].

MCTS has been successful in many applications, including computer Go [39, 40], dynamic resource allocation [41], trajectory planning [42], expression discovery [43], and playing video games [44, 45]. It is conceptually simple, scales to large problems, accommodates a variety of computational budgets, and is very flexible. One reason for its scalability is that MCTS is an online search algorithm, which approximates the solution to the MDP only for the current state rather than for all states. As a result, the search is focused only on the reachable state space, which can be substantially smaller than the entire state space. Computationally, MCTS is an anytime algorithm. The algorithm can be stopped at anytime and provide a solution, but the solution can be improved with additional computations. The flexibility of MCTS has led to many variants of the algorithm in support of different problem requirements [37]. In Chapter 5 of this thesis, we leverage this flexibility to extend the MCTS algorithm to support systems with hidden state by using the node position in the tree as a substitute for the state.

**Double Progressive Widening.** The original MCTS algorithm is suitable for small discrete state and action spaces. For large or continuous state and action spaces, a variant of MCTS with *double progressive widening* (DPW) can be used [41, 46]. In progressive widening, not all possible children (either states or actions) are considered initially. The set of considered children is grown incrementally with the number of visits. Double progressive widening applies progressive widening to both states and actions. When the state and action spaces are large (or infinite), visited states and actions are not revisited sufficiently through sampling alone, which hinders the quality of value estimates. The benefit of MCTS-DPW's progressive widening is that it forces revisits to existing states and slowly allows new states to be added as the total number of visits increase. Progressive widening stabilizes value estimates and prevents explosion of the branching factor of the tree.

The MCTS-DPW algorithm is shown in Algorithm 2. The algorithm consists of a main loop that repeatedly performs forward simulations of the system while building the search tree and updating the state-action value estimates. The search tree $\mathcal{T}$ is initially empty. Each simulation runs from initial state to terminal state. The path is determined by the actions of algorithm and the state transitions sampled from $G$. There are three stages in each simulation: search, expansion, and rollout.

- *Search.* In the search stage, which is implemented by SIMULATE (line 7), the algorithm starts at the root of the tree and recursively selects a child to follow. At each visited state node, the first progressive widening criteria (line 15) determines whether to choose amongst previously visited actions, tracked by $A(s)$, or to sample a new action from GETACTION. The criterion limits the number of actions at a state $s$ to be no more than polynomial in the total number of visits to that state. Specifically, a new action $a$ is sampled if $\|A(s)\| < kN(s)^{\alpha}$, where $k$ and $\alpha$ are parameters, $\|A(s)\|$ is the number of previously visited actions at state $s$, and $N(s)$ is the total number of visits to state $s$. Otherwise, the existing action from $A(s)$ that maximizes

$$Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{3.6}$$

is chosen (line 19), where $c$ is an *exploration parameter* that controls the amount of exploration in the search, and $N(s,a)$ is the total number of visits to action $a$ in state $s$. Equation 3.6 is the upper confidence tree (UCT) equation [36]. The second term in Equation 3.6 is an *exploration bonus* that encourages selecting actions that have not been tried as frequently. At each action node, the second progressive widening criteria (line 20) determines whether to reuse a previously encountered transition, tracked by $V(s,a)$, or to sample a new transition from $G$. Similar to the first widening, the second widening criterion limits the number of transitions at an action node, defined by the state-action pair $(s,a)$, to be no more than polynomial in the total number of visits to that state-action pair. Specifically, a new transition (next state) is sampled from $G$ if $\|V(s,a)\| < k'N(s,a)^{\alpha'}$, where $k'$ and $\alpha'$ are parameters, $\|V(s,a)\|$ is the number of previously visited transitions at $(s,a)$, and $N(s,a)$ is the total number of visits to $(s,a)$. Otherwise, an existing transition is sampled from SAMPLE. The search stage continues in this manner until the system transitions to a state that is not in the tree $\mathcal{T}$.

- *Expansion.* Once we have reached a state that is not in the tree $\mathcal{T}$, we create a new node for the state and add it (line 18). The set of previously taken actions from this state, $A(s)$, is initially empty and the number of visits to this state $N(s)$ is initialized to zero.
- *Rollout.* Starting from the state created in the expansion stage, we perform a *rollout* that repeatedly samples state transitions until the desired termination is reached (line 37). In the ROLLOUT function (line 36), actions are drawn according to a rollout policy $\pi_0$ and state transitions are drawn from $G$. The rollout policy is often simply a uniform distribution over possible actions.

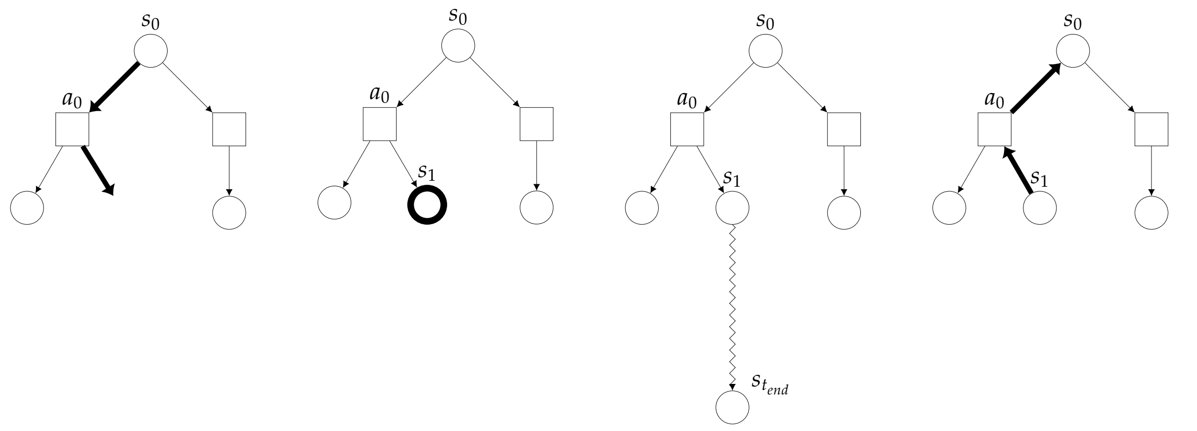
At each transition in the simulation, the reward is used to update estimates of the state-action value function $Q(s,a)$ (line 34). The values are used to guide the search in the UCT selection criterion (line 19). Simulations are run until the stopping criterion is met, which is most commonly a fixed number of iterations or a fixed amount of time. The algorithm returns the best action at the current time step, which occurs at the root of the search tree.

Figure 3.3 illustrates the four primary operations of the MCTS algorithm. The three stages of each simulation, which include search, expansion, and rollout, are shown in Figures 3.3a–3.3c and the update

**Algorithm 2** Monte Carlo tree search with double progressive widening

```
1: ▷ Inputs: State $s$, depth $d$
2: ▷ Returns: Action with highest estimated $Q$-value at the root $a$
3: function MCTS-DPW($s, d$)
4:     loop                                                    ▷ Repeat until stopping criterion
5:         SIMULATE($s, d$)
6:     return $argmax_a Q(s, a)$
7: function SIMULATE($s, d$)
8:     if $d = 0$ or $s \in S_{terminal}$
9:         return 0
10:    if $s \notin \mathcal{T}$
11:        $\mathcal{T} \leftarrow \mathcal{T} \cup \{s\}$                            ▷ Expansion, add new node
12:        $N(s) \leftarrow 0$
13:        return ROLLOUT($s, d$)
14:    $N(s) \leftarrow N(s) + 1$
15:    if $\|A(s)\| < kN(s)^{\alpha}$                                 ▷ Progressive widening of actions
16:        $a \leftarrow$ GETNEXT($s, Q$)                                       ▷ New action
17:        $(N(s,a), Q(s,a), V(s,a)) \leftarrow (0,0,0)$
18:        $A(s) \leftarrow A(s) \cup \{a\}$
19:    $a \leftarrow \text{argmax}_a\, Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$                   ▷ UCT selection criterion
20:    if $\|V(s,a)\| < k'N(s,a)^{\alpha'}$                             ▷ Progressive widening of states
21:        $(s', r) \sim G(s,a)$                                        ▷ Sample a new transition
22:        if $s' \notin V(s,a)$
23:            $V(s,a) = V(s,a) \cup \{s'\}$
24:            $R(s,a,s') \leftarrow r$
25:            $N(s,a,s') \leftarrow 0$
26:        else
27:            $N(s,a,s') \leftarrow N(s,a,s') + 1$
28:    else
29:        $s' \leftarrow$ SAMPLE($N(s,a,\cdot)$)                           ▷ Sample an existing transition
30:        $r \leftarrow R(s,a,s')$
31:        $N(s,a,s') \leftarrow N(s,a,s') + 1$
32:    $q \leftarrow r + \gamma$SIMULATE($s', d-1$)
33:    $N(s,a) \leftarrow N(s,a) + 1$
34:    $Q(s,a) \leftarrow Q(s,a) + \frac{q - Q(s,a)}{N(s,a)}$                          ▷ Update estimate of Q
35:    return $q$
36: function ROLLOUT($s, d$)
37:    if $d = 0$ or $s \in S_{terminal}$
38:        return 0
39:    $a \sim \pi_0(s)$                                         ▷ Sample action from rollout policy
40:    $(s', r) \sim G(s,a)$
41:    return $r + \gamma$ROLLOUT($s', d-1$)
```

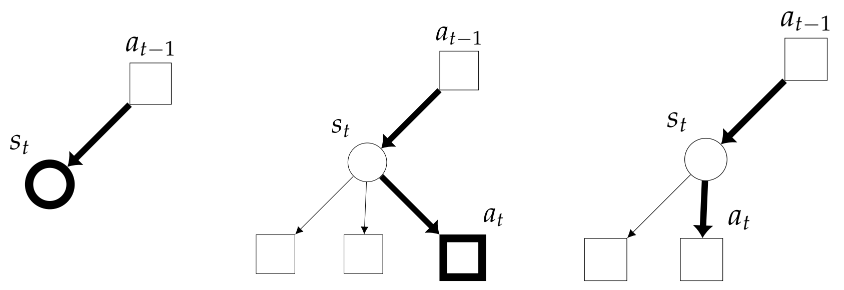
operation is shown in Figure 3.3d. State nodes are denoted as circles and action nodes are denoted as squares. In the search stage, shown in Figure 3.3a, the algorithm recursively selects a child to follow using the progressive widening and UCT criteria until it reaches a state that has not been previously visited. The selected nodes are labeled $s_0$ and $a_0$ in the example. A new node representing the unvisited state, labeled $s_1$ in Figure 3.3b, is added to the tree in the expansion stage. Starting from the newly added node, a rollout continues the simulation until a terminal state ($s_{t_{end}}$ in Figure 3.3c) is reached by selecting actions using a fixed rollout policy $\pi_0$. Finally, in Figure 3.3d, the observed rewards are used to update the Q-values at the nodes along the simulated path. The Q-values are used in the UCT criterion in the search stage.



(a) Search and follow child node.
(b) Expand node of unvisited state.
(c) Rollout to end using rollout policy $\pi_0$.
(d) Update Q-values for nodes along path.

Figure 3.3: Primary operations of Monte Carlo tree search with double progressive widening (MCTS-DPW): search, expansion, rollout, and update.

Figure 3.4 illustrates the three cases that can occur when visiting a state in the search and expansion stages. If the state $s_t$ does not exist in the tree, then add the node to the tree as shown in Figure 3.4a. If the node exists and the number of child action nodes satisfies the progressive widening criterion, then add a new action node as shown in Figure 3.4b. Specifically, a new child action node is added if $\|A(s_t)\| < kN(s_t)^\alpha$, where $k$ and $\alpha$ are parameters, $\|A(s_t)\|$ is the number of previously visited actions at state $s_t$, and $N(s_t)$ is the total number of visits to state $s_t$. Otherwise, select an existing child action to follow using the UCT criterion (Equation 3.6) as shown in Figure 3.4c.



(a) Add state if unvisited.

(b) Progressive widening adds new action.

(c) Select existing action using UCT.

Figure 3.4: Three cases of search and expansion stages when visiting a state node: expansion, progressive widening, and UCT selection.

Chapter 4

# Interpretable Machine Learning

This chapter provides a broad overview of approaches to interpretability and explainability in machine learning, including rule-based methods and expression optimization. The purpose of this chapter is to provide the context, motivation, and background concepts necessary for understanding grammar-based decision tree (GBDT), a core contribution presented in Chapter 7 of this thesis. In particular, this chapter introduces several core topics used in that chapter, including context-free grammar, genetic programming, and grammatical evolution. Section 4.1 motivates the need for interpretability and explainability in machine learning. Section 4.2 surveys the main approaches to interpretable machine learning. Section 4.3 reviews decision trees. Finally, Section 4.4 reviews expression optimization and various algorithms for optimization.

## 4.1 Introduction

Interpretability and explainability are often important qualities of machine learning models. *Interpretability* is the degree to which a human can understand the cause of a decision, and is often used interchangeably with *explainability*, which is the ability to produce human-understandable justifications. Since there is often a trade-off between predictive accuracy and interpretability of machine learning models, the machine learning literature has largely emphasized black box models, such as deep neural networks [1], in recent times. Black box models can often provide better performance in terms of accuracy, but humans cannot intuitively rationalize the decision process.

Interpretability and explainability are important for a variety of reasons. When a human can verify intuitively how a machine learning system arrived at its decision, the human is much more likely to trust it. Conversely, interpretability also enables the human to confidently disagree with the system if the explanation of the decision is unsatisfactory. Over time, interpretability also allows the human to better understand the limitations of the system, and where the system decisions are more likely to fail. For these reasons, interpretability and explainability are especially important for the analysis and operation of safety-critical and other high-stakes systems.

Interpretability also plays a critical role in the validation and debugging of machine learning systems. Machine learning systems are not perfect. When they do fail, interpretability helps validation engineers understand why the failure occurred and how the system can be changed to mitigate future failures.

Lastly, machine learning algorithms are often much better at pattern recognition and dealing with large and high-dimensional datasets than humans. As such, humans can learn a great deal from machine learning models. Interpretability can be used as a tool for humans to mine knowledge from large complex datasets by first training machine learning models on the large datasets, and then asking the machine learning models for explanations.

One of the core challenges to interpretability and explainability is addressing how best to express explanations. Explanations are, at its core, for human consumption, so they should align with the way humans think and existing domain concepts. One approach is to leverage existing physics-based models that humans are already familiar with as a basis and fit parameters using the data [47]. However, such models may not be generally available. In this thesis, we address this issue by using a context-free grammar to formally define the language of the expressions.

## 4.2 Approaches to Interpretable Machine Learning

A variety of interpretable models for static data have been proposed in the literature. Regression models [48], generalized additive models [49], and Bayesian case models [50] have been recently proposed as models with interpretability. These models aid interpretability by stating decision boundaries in terms of basis functions or representative instances (prototypes). Bayesian networks have also been used for prediction and data understanding [51]. Bayesian networks represent the relationship between variables as conditional probabilities.

Rule-based models, such as decision trees [52], decision lists [53], and decision sets [54] are easy to understand because their decision boundaries are stated in terms of input features and simple logical operators. Decision trees partition the data using a tree structure while decision lists use a *if-then-else* branching structure. Decision sets use independent decision rules to reduce coupling between rules.

In this thesis, our focus is on time series analysis. Interpretability for time series has focused on finding motifs such as shapelets [55] and subsequences [56], which have been applied to time series classification. These approaches search for simple patterns that are most correlated with the class label. Interpretability comes from identifying a prototype of a recurring subsequence pattern. Implication rules [57] and simple logical combinations of shapelets [58] have been proposed to extend the shapelets approach. Hidden Markov models and dictionary-based approaches have also been used [59, 60].

Another approach to interpretability is *expression optimization* [61], which is closely related to program synthesis [62], program induction [63], automatic programming [64], programming by example [65], inductive logic programming [66], and expression discovery [43]. The idea is to learn a symbolic *expression*, or program, that can be executed to represent decision boundary of the system. Expressions are very general and can be mathematical formulas, logic formulas, or even software programs. This approach to interpretable machine learning is very general, and can thus be used to solve a very broad class of problems. However, searching over symbolic expressions is very computationally intensive and finding the optimal expression is generally intractable.

## 4.3 Decision Tree

The decision tree is a popular machine learning model for classification that humans find very intuitive. Decision tree learning recursively partitions the data using a tree structure where the splits are either the features of the data for Boolean features, or simple logical operators for non-Boolean features [52, 67]. We consider a dataset $D$, which consists of $m$ feature vector $x^j$ and label $l^j$ pairs such that $((x^1, l^1), (x^2, l^2), ...(x^m, l^m))$. Each feature vector has $n$ features where feature $i$ is denoted by $x_i$. Algorithm 3 illustrates the decision tree training algorithm for Boolean features and loss function $f$. The loss function $f(i, D)$ takes as input a feature $i$ and the dataset $D$ and returns a real-valued loss. Lower loss values are preferred.

**Algorithm 3** Decision Tree Training

```
1:  ▷ Inputs: Loss function $f$, dataset $D$, depth $d$
2:  function DecisionTree($f, D, d$)
3:      $R \leftarrow$ Split($f, D, d$)
4:      return Tree($R$)
5:  function Split($f, D, d$)
6:      if IsHomogeneous(Labels($D$)) or $d = 0$
7:          return Leaf(Mode(Labels(D)))
8:      $i^* \leftarrow \text{argmin}_i f(i, D), i = 1..n$
9:      $D^+ \leftarrow [(x, l) \mid x_{i^*}, (x, l) \in D]$
10:     $D^- \leftarrow [(x, l) \mid \neg x_{i^*}, (x, l) \in D]$
11:     $child^+ \leftarrow$ Split($f, D^+, d - 1$)
12:     $child^- \leftarrow$ Split($f, D^-, d - 1$)
13:     return INode($x^*, child^+, child^-$)
```

In Algorithm 3, DecisionTree (line 2) is the main entry point to the training algorithm. It returns a

TREE object containing the root node of the decision tree. SPLIT (line 5) attempts to partition the data into two parts. It first tests whether the terminal conditions are met and if so returns a LEAF node that predicts the mode of the labels. The partitioning terminates if the maximum depth has been reached or if all class labels are the same, which is tested by the ISHOMOGENEOUS function (line 7). We determine the best feature to split on by evaluating the loss function conditioned on each feature and the one that minimizes the loss function $f(i, D)$ is chosen (line 8). Then we partition the data into two parts conditioning on the Boolean value of feature $i^*$ (lines 9–10). SPLIT is called recursively on each part (lines 11–12). Finally, the function returns an internal node INODE containing the decision feature and the children of the node (line 13).

Gini impurity [52] is a popular choice for the loss function $f$. The loss function with Gini impurity is given by

$$f_{Gini}(i, D) = \sum_{L \in \{L^+, L^-\}} \sum_{b \in B} f_L^b (1 - f_L^b)$$

where $L^+$ are the labels where feature $i$ is true, i.e., $L^+ = [l \mid x_i, (x, l) \in D]$; $L^-$ are the labels where feature $i$ is false, i.e., $L^- = [l \mid \neg x_i, (x, l) \in D]$; $B = \{True, False\}$, and $f_L^b$ is the fraction of labels in $L$ that are equal to $b$, i.e., $f_L^b = \mathbb{1}\{l=b\}/|L|$, where $\mathbb{1}$ is the indicator function and $|L|$ denotes the number of elements in $L$. We use square brackets with set construction syntax to indicate that $L^+$ and $L^-$ are vectors and can have duplicate elements. Another popular choice for the loss function is maximizing the information gain [67].

## 4.4 Expression Optimization

Expression optimization is an approach to interpretability in machine learning by learning symbolic expressions as the model [61]. Humans can inspect the expressions to understand the model's exact behavior and how the model arrived at its decision. The desirability of an expression is defined by a loss function, and the optimal expression is the one that minimizes loss. Expression optimization relies on a context-free grammar (CFG) to specify the syntax of valid expressions. Since search occurs over syntactically valid expressions, the CFG inherently defines the domain of the search. We assume that the semantics of the symbols in the grammar are known.

The expression optimization problem is to find the expression $e^*$ from a CFG $G$ that minimizes a given loss function $f(e)$, i.e., $e^* = \text{argmin}_{e \in G} f(e)$ [68]. The formulation is extremely general due to the flexibility and expressiveness of grammars and the arbitrary choice of loss function. For example, the grammar can be used to define the syntax of linear temporal logic [7] or the syntax of an entire programming language [69]. Owing to this generality, expression optimization has been applied to a wide variety of applications, including image and signal processing; modeling of medical, economic, and earth science data; and industrial process control [70]. In the following sections, we first review context-free grammars and then we present several algorithms for optimizing expressions from a grammar.

### 4.4.1 Context-Free Grammars

A context-free grammar (CFG) defines a set of rules that govern how to form expressions in a formal language, such as LTL [7]. The grammar defines the syntax of the language and provides a means to generate valid expressions. A CFG $G$ is defined by a 4-tuple $(\mathscr{N}, \mathscr{T}, \mathscr{P}, \mathscr{S})$, where $\mathscr{N}$ is a set of *non-terminals*; $\mathscr{T}$ is a set of *terminals*; $\mathscr{P}$ is a set of *production rules*, which are rules governing the substitution of non-terminals; and $\mathscr{S}$ is the special *start* symbol that begins a derivation.

To generate an expression from the grammar, we begin with the start symbol as an expression, then repeatedly apply production rules to rewrite the expression until no more non-terminals remain. When applying a rule, a non-terminal in the expression is replaced with a substitution defined in a production rule. The rules can be applied any number of times and in any order. The derivation is commonly represented as a tree structure called a *derivation tree*. Most algorithms use derivation trees as the internal representation for the expressions and partial expressions.

**Example: Expression Derivation from a CFG.** Figure 4.1 shows a CFG for generating simple sentences, adapted from [61].

$$\begin{aligned}
S &::= N\ V \\
V &::= V\ A \mid \textit{runs} \mid \textit{reads} \mid \textit{writes} \\
A &::= \textit{rapidly} \mid \textit{efficiently} \\
N &::= \textit{Alice} \mid \textit{Hugo}
\end{aligned}$$

Figure 4.1: Context-free grammar (CFG) example for generating simple sentences.

In this example, the start symbol is $S$; the non-terminals are $S$, $V$, $A$, $N$; and the terminals are *runs*, *reads*, *writes*, *rapidly*, *efficiently*, *Alice*, *Hugo*. The symbol | in the production rules indicates possible selection alternatives. To derive an expression from the grammar, we begin with the start symbol $S$, and then we start substituting non-terminals using the production rules. There is only one possibility for substituting $S$, which is $N\ V$. Now for $N$, we can choose amongst two alternatives *Alice* and *Hugo*. Let's say we choose *Hugo*, which gives *Hugo* $V$. For $V$, we have 4 possibilities: $V\ A$, *runs*, *reads*, and *writes*. Let's say we choose *reads*, then we have our completed expression *Hugo reads* with no remaining non-terminals. If instead, we had chosen $V\ A$ for $V$, then substituting into the partial expression gives *Hugo* $V\ A$, which has two non-terminals. Then one way to complete the expression would be to choose *runs* for $V$ and *rapidly* for

$A$, which gives a complete expression *Hugo runs rapidly*. As we saw, a grammar can define an infinite set of expressions through recursion. The derivation tree for *Hugo runs rapidly* is shown in Figure 4.2.

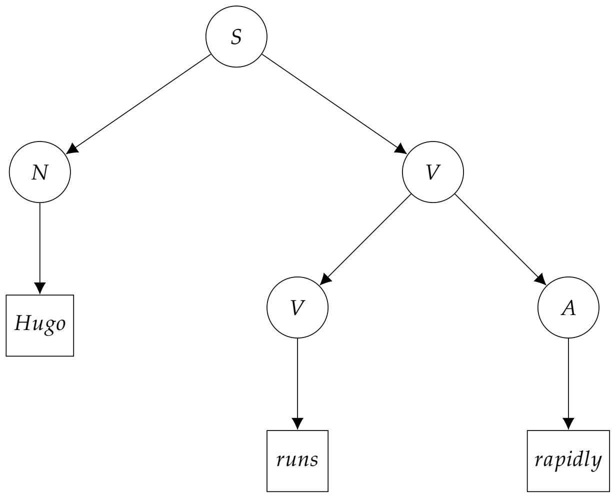


Figure 4.2: Derivation tree example for the context-free grammar (CFG) in Figure 4.1.

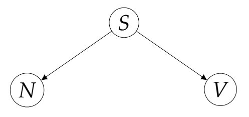
### 4.4.2 Optimization Algorithms

A number of existing algorithms can be used for optimizing expressions. The goal is to find the expression $e^*$ from a CFG $G$ that minimizes a loss function $f$, i.e., $e^* = \text{argmin}_e f(e)$. This section describes three classes of algorithms: Monte Carlo [61], genetic programming [71], and grammatical evolution [72].

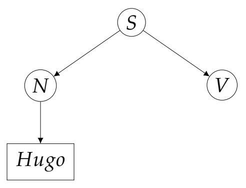
#### Monte Carlo

The Monte Carlo approach for expression optimization generates expressions by repeatedly selecting non-terminals in the partial expression and applying a production rule chosen uniformly at random amongst possible rules. When no non-terminals remain, the loss of the generated expression is evaluated. The expression generation process is iterated and the expression with the best loss is reported. A maximum depth is typically used to prevent infinite nesting and ensure that the process terminates. The Monte Carlo process for generating expressions can be compactly implemented as a recursion on the derivation tree. Each node represents a substitution in the partial expression. Each node makes a uniformly random substitution and then recurses on each non-terminal child.

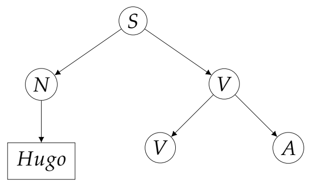
Figure 4.3 illustrates the Monte Carlo approach to expression generation using the example in Figure 4.2. We first start by expanding the start symbol $S$ in Figure 4.3a. There is only one option, so we expand $S$ to $N\ V$. In Figure 4.3b, we select the leftmost non-terminal to expand, which is symbol $N$ in this case. The production rule corresponding to $N$ has two possible substitutions as seen in the fourth rule of $G$ in Figure 4.1. We choose amongst these two substitutions uniformly randomly. Suppose *Hugo* was selected, then we substitute *Hugo* for $N$ by adding a child node in the derivation tree. We continue to expand the next non-terminal node in Figure 4.3c, which is $V$. There are four possible substitutions for $V$ and we choose uniformly randomly among them. Suppose we selected the first option $V\ A$, then we expand $V$ to $V\ A$ in the derivation tree as shown in Figure 4.3c. The process continues in the same manner in Figure 4.3d, where we expand the leftmost non-terminal $V$, and randomly select *runs*. Then, finally in Figure 4.3e, we expand $A$ to *rapidly*. There are no unexpanded non-terminals remaining and the tree is complete.



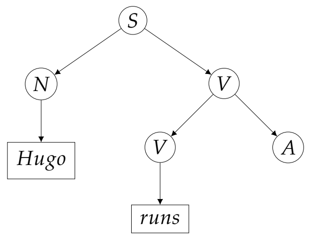
(a) Expand start symbol $S$ to $N\ V$.



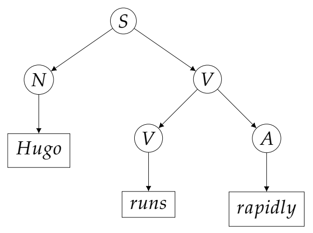
(b) Replace $N$ with *Hugo*.



(c) Expand $V$ to $V\ A$.



(d) Replace $V$ with *runs*.



(e) Replace $A$ with *rapidly*.

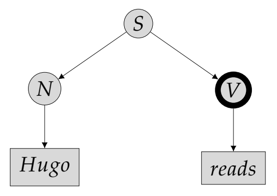
Figure 4.3: Example illustrating Monte Carlo approach to generating an expression.

The major advantage of the Monte Carlo algorithm is simplicity. The algorithm is simple to implement, requires little memory, and is easily parallelizable. However, because Monte Carlo sampling is undirected, identical expressions may be evaluated multiple times, which can be inefficient. Furthermore, due to the nature of the sampling, shallower paths will get sampled more frequently than deeper paths. In fact, deep paths may have a vanishingly small probability of being reached [70].

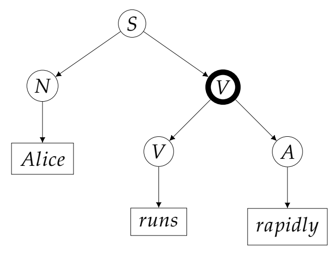
### Genetic Programming

Genetic programming (GP) is an evolutionary algorithm for optimizing trees [73]. As with traditional evolutionary methods, GP maintains a population of solution candidates (here, derivation trees) and applies genetic operators on them to create new and possibly better candidates. Better solutions, as determined by the loss function, are more likely to be retained in the population. The loss function is also known as the fitness function in the evolutionary algorithms literature. The generation of new candidates is performed in a number of rounds, called *generations*. The candidate solution with the best loss value is finally returned. The genetic operators for GP are defined specifically for trees and thus do not require the derivation tree to be transformed into another intermediate representation. The two primary operators in GP are crossover and mutation. The crossover operator takes two trees as input and randomly exchanges a compatible subtree. Derivation subtrees are compatible if they are replacing the same non-terminal symbol. The mutation operator takes a single tree as input and randomly replaces a subtree with one that is randomly generated, for example, by sampling uniformly over possible production rules. This thesis uses standard tournament selection to select the candidates for applying the genetic operators [71].

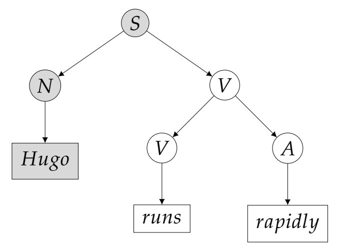
Figure 4.4 illustrates the two genetic operators in GP, crossover and mutation, which serve to generate new expressions from existing ones. The crossover operator takes two parent trees and combines them to create a new child tree. First, a random non-terminal node is selected in the first parent ($V$ in Figure 4.4a). A random node with the same non-terminal symbol is selected from the second parent (Figure 4.4b). The child, shown in Figure 4.4c, is the first parent with the selected subtree from the second parent substituted at the first parent's selected node. The mutation operator takes a single parent tree and modifies it to produce a new child tree. First, a random non-terminal is selected in the parent ($V$ in Figure 4.4d). Then, the subtree is discarded and replaced by a randomly generated subtree using Monte Carlo as shown in Figure 4.4e.



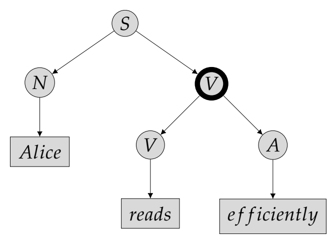
(a) Crossover 1: Select random node from first parent.



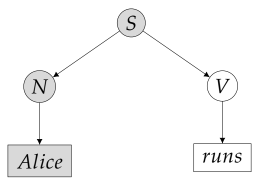
(b) Crossover 2: Select node of same type from second parent.



(c) Crossover 3: Child is first parent with subtree from second parent.



(d) Mutation 1: Select random node from parent.



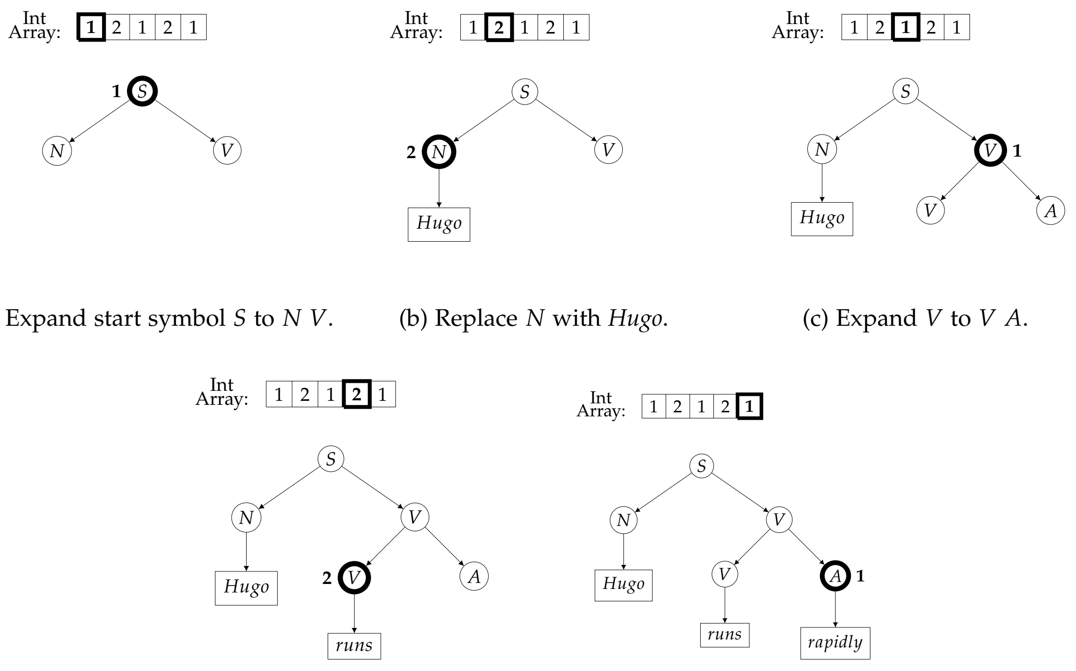
(e) Mutation 2: Child is parent with random subtree.

Figure 4.4: Crossover and mutation operators for generating new derivation trees in genetic programming (GP).

### Grammatical Evolution

Grammatical evolution (GE) [72] is another evolutionary algorithm that operates on trees. Like GP, a population is maintained and genetic operators are applied to candidate solutions to produce the next generation of population. The candidate solution with the best loss value is finally returned. However, unlike GP, the genetic operators do not directly operate on trees. Instead, GE relies on an intermediate representation by transforming the tree into a sequential representation, typically by assuming a depth-first traversal order. Traditional GA [74] can be applied to manipulate the solutions, then the transformation is reversed to recover the derivation tree. Specifically, GE defines a transformation from a variable-length integer array to a sequence of rule selections in a CFG. Then a standard GA is used to search over integer arrays. This thesis one-point crossover and uniform mutation for the genetic operators in the GA [70].

Figure 4.5 illustrates the mapping between the derivation tree and the intermediate sequential representation, which uses an integer array. The elements of the integer array represent the selection in a production rule. For example, a 2 means to select the second option in the production rule. To make the selections in the derivation tree a sequence, a traversal order is chosen, such as a depth-first traversal order. We illustrate in Figure 4.5 using the example from Figure 4.2. In Figure 4.5a, we start with the start symbol $S$ and expand it using the first element in the integer array. The first element is 1, so we expand $S$ using the first (and only) option $N\ V$. Following the depth-first traversal order, we now expand $N$ in Figure 4.5b. The second element of the integer array indicates to use the second option. The option numbers depend on the particular non-terminal symbol. For $N$, the second option is *Hugo* and we make the substitution. Figures 4.5c–4.5e continue the derivation in this manner, traversing through elements three through five of the integer array and replacing the non-terminals $V$, $V$, and $A$ respectively.



(a) Expand start symbol $S$ to $N$ $V$. (b) Replace $N$ with *Hugo*. (c) Expand $V$ to $V$ $A$.

(d) Replace $V$ with *runs*. (e) Replace $A$ with *rapidly*.

Figure 4.5: Mapping between integer array and derivation tree in grammatical evolution (GE).

# Chapter 5

# Finding Failures using Reinforcement Learning

This chapter presents adaptive stress testing (AST), a core component of AdaStress and a core contribution of this thesis. The key innovation behind AST is to improve stress testing by posing it as a sequential decision-making problem thus enabling existing reinforcement learning algorithms to adaptively and efficiently search the space. The first section, Section 5.1, introduces the approach and Section 5.2 describes the general AST framework. Section 5.3 discusses how to apply AST to fully observable systems and Section 5.4 discusses how to apply AST to partially observable systems. Section 5.5 summarizes the contributions made in this chapter.

## 5.1 Introduction

Stress testing in simulation is a popular approach to testing systems for failures due to its flexibility and ease of implementation. In Chapter 2, we described various approaches to generating test inputs in attempt to induce a failure event from the system under test. The input space, or a discretization thereof for continuous spaces, can be exhaustively covered. However, this approach is only tractable for very small problems. An alternative approach would be to use a low-dimensional parameterization to abstract the higher-dimensional input space. However, this approach relies heavily on the design of the abstraction and the structure of the problem. A third approach uses Monte Carlo samples of a stochastic model to capture the system's environment, but this approach can be very sample inefficient for rare corner case failures because it does not leverage the temporal (sequential) structure of the problem. Finally, the problem can be formulated as a trajectory planning problem and an algorithm such as RRT can be applied. However, the RRT approach requires the ability to steer the transition function and a distance measure on states, which may not be available.

This chapter proposes the AST framework, which formulates the stress testing problem as a sequential decision-making problem. This view not only exposes the sequential structure of the problem to be leveraged, it also enables powerful existing reinforcement learning algorithms to be used for search.

## 5.2 Adaptive Stress Testing

Adaptive stress testing (AST) aims to find the most likely path from a start state to a failure state in a discrete-time simulator [75]. We formulate the search as a sequential decision-making problem and then use reinforcement learning to optimize it. Since one of the main applications of this work is the stress testing of safety-critical systems, we take the view that the simulator can be decomposed internally into the system under test and a stochastic model of its environment. However, this view is not required, and the formulations and algorithms presented in this paper can be applied without modification whether this view is adopted or not.

Figure 5.1 illustrates the general AST framework. A simulator $\mathcal{S}$ models the system under test and the environment at each time step. A reinforcement learning agent interacts with the simulator over multiple time steps to maximize the reward it receives. At each time step, the agent makes an observation, takes an action based on the observation, and receives a reward. The reward function is crafted to both encourage failure events and maximize transition probability. In other words, the overall goal of the agent is to search for a sequence of actions that is most *adversarial* to the system under test. AST returns the most likely failure path, which is the sequence of states with the highest probability that ends in a failure event. Without loss of generality, we assume that the simulator has multivariate heterogeneous states and actions as discussed in Chapter 3.

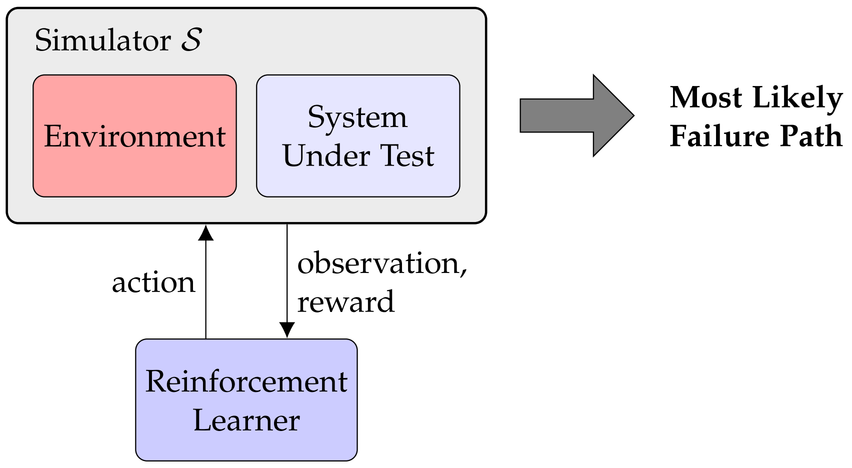


Figure 5.1: Conceptual overview of adaptive stress testing.

Formally, let $E \subset S$ be the set of states where the event of interest occurs, $s_t \in S$ be the state of the simulator at time $t$, $a_t \in A$ be the actions of the reinforcement learner that affect the simulation at time $t$, and $P(s_{t+1} \mid s_t, a_t)$ be the probability of transitioning from state $s_t$ to state $s_{t+1}$. Then, the AST problem is to find the sequence of simulation states $s_0, s_1, ..., s_{t_{end}}$ that results from optimizing:

$$\begin{aligned} \max_{a_0, a_1, ..., a_{t_{end}}} \quad & \prod_{t=0}^{t_{end}-1} P(s_{t+1} \mid s_t, a_t) \\ \text{subject to} \quad & s_{t_{end}} \in E \end{aligned}$$

## 5.3 Fully Observable Systems

When the state of the system is fully observable, we can formulate the problem as an MDP. The state of the MDP is the state of the simulator $s_t$, which includes both the state of the environment and the state of the system under test. The transition behavior of the simulator defines $T(s_t, a_t)$ and the transition probabilities are given by $P(s_{t+1} \mid s_t, a_t)$ assumed to be available from the simulator. The actions $a_t$ are the controllable elements of the simulator and are chosen to be the stochastic elements of the simulator. These elements represent the uncertainties in the simulation models, such as wind speed, sensor noise, or how fast a pedestrian walks. Rather than sampling over these uncertainties, the action $a_t$ directly sets their values and then we account for the probability of the value occurring in the reward function. By setting the stochastic variables through the actions, we remove transition stochasticity, which allows us to first solve the MDP for the optimal policy $\pi^*$ and action sequence $a_0, a_1, ..., a_{t_{end}}$, and then replay the actions to find the sequence of states $s_0, s_1, ..., s_{t_{end}}$. If the transitions were stochastic, then the resulting sequence of states would not be unique.

Figure 5.2 illustrates the AST framework for one time step in the fully observable case. At each time step, the agent observes the simulator state $s_t$ and chooses an action $a_t$ based on $s_t$. The system then transitions to the next state $s_{t+1}$ and the agent receives a reward $r_{t+1}$. To properly account for the transition probabilities in the reward function described in the following section, we assume a finite horizon problem and set $\gamma = 1$.

**Reward Function.** The reward function is designed to find failure events as the primary objective and maximize the path probability as a secondary objective. Let $R_E \in \mathbb{R}^+$ be the reward for reaching an event, $d_t \in \mathbb{R}^+$ be the miss distance, $E \subset S$ be the event states, and $t_{end} \in \mathbb{N}$ be the time at which the simulation terminates. Then the reward function is given by:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} R_E & \text{if } s_t \in E \\ -d_t & \text{if } s_t \notin E,\ t \geq t_{end} \\ \log(P(a_t \mid s_t)) & \text{if } s_t \notin E,\ t < t_{end} \end{cases} \tag{5.1}$$

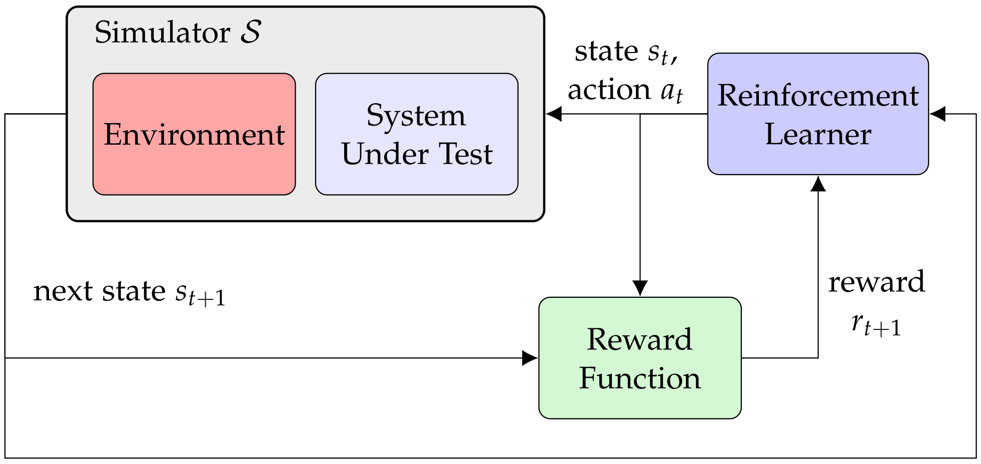


Figure 5.2: Adaptive stress testing of a fully observable system.

The reward function has three terms that provides a big reward for finding an event, penalizes the learner proportionately according to a miss distance for not finding an event, and penalizes individual transitions according to their likelihood of occurrence. The first term of Equation 5.1 assigns a positive reward $R_E$ if the path terminates and a failure event occurs. If the path terminates and an event does not occur, the second term of Equation 5.1 penalizes the agent by assigning the negative of the miss distance $d_t$ to the agent. The miss distance $d_t$ is some measure defined by the user that depends on $s_t$ and indicates how close the simulation came to a failure. If such a measure is not available, then $-d_t$ can be set to a large negative constant. However, providing an appropriate miss distance can greatly accelerate the search by giving the agent the ability to distinguish the desirability of two paths that do not contain failure events. The third term of Equation 5.1 maximizes the overall path probability by awarding the log probability of each transition. The transition probability is given by $P(s_{t+1} \mid s_t, a_t)P(a_t \mid s_t)$. Since we have chosen our actions to deterministically set the stochastic variables of the simulator, the transitions are deterministic, i.e., $P(s_{t+1} \mid s_t, a_t) = 1$, and the transition probability simplifies to $P(a_t \mid s_t)$. Recall that reinforcement learning maximizes the expected sum of rewards. By choosing a reward of the log probability at each step, the reinforcement learning algorithm then maximizes the sum of the log probabilities, which is equivalent to maximizing the product of the probabilities.

By viewing the simulator $\mathcal{S}$ as a generative model $G$ that produces the next state and reward from the current state and action such that $(s_{t+1}, r_{t+1}) \sim G(s_t, a_t)$, the problem conforms to a traditional reinforcement learning problem as discussed in Section 3.3. Table 5.1 summarizes the mapping of the stress testing problem to the reinforcement learning problem. Existing reinforcement learning algorithms such as Q-learning (Section 3.4) [35] and MCTS (Section 3.5) [36] can be applied to optimize the decision process and find the optimal path. Concurrent work has explored using deep reinforcement learning (DRL), i.e., reinforcement learning algorithms that use neural network function approximators, with AST to find collisions between autonomous cars and pedestrians at crosswalks [76].

Table 5.1: Fully observable stress testing to RL problem mapping

| RL component | Stress testing component |
|---|---|
| State space $S$ | State space of simulator $\mathcal{S}$ |
| Action space $A$ | Stochastic variables of the simulator $\mathcal{S}$ |
| Generative model $G$ | Simulator state transition and reward function $R$ (Eq. 5.1) |
| Initial state $s_0$ | Initial state $s_0$ |

## 5.4 Partially Observable Systems

The formulation described in Section 5.3 requires access to the full state of the simulator. In some applications, some or all of the state variables may not be accessible. For example, black box components in the simulator, such as software binaries, can maintain state without exposing it externally. Incomplete state information leads to different states being aliased to the same observation, which can confuse the learner, hinder learning, and lead to poor optimization performance.

To deal with the partial observability, we introduce an abstraction that relaxes the need for the simulator to expose its state. Instead of explicitly representing and passing the state into and out of the simulator, we now assume that the simulator maintains state internally and the state is updated in-place. In other words, we have previously assumed that the simulator is stateless, but now we assume that the simulator is stateful. Furthermore, rather than explicitly choosing and passing values of the stochastic variables as the actions, we use a pseudorandom seed $\bar{a}_t$ as a proxy. The simulator uses $\bar{a}_t$ to seed an internal random process that draws a sample of the stochastic variables $a_t$. We assume that setting the seed makes the sampling process deterministic and sampling uniformly over all pseudorandom seeds returns the natural distribution of the stochastic models in the simulator.

### 5.4.1 Seed-Action Simulators

A *seed-action simulator* $\bar{\mathcal{S}}$ is a stateful simulator that uses a pseudorandom seed input to update its state in-place. The hidden state $s_t$ is not exposed externally, making the simulator appear non-Markovian to external processes, such as the reinforcement learning agent. The simulator uses $\bar{a}_t$ to draw a sample of the stochastic variables $a_t$ and transition to the next state $s_{t+1}$. The next state replaces the current state in-place. The simulator returns the transition probability $p_{t+1}$ given by $P(a_t \mid s_t)$; a Boolean indicating whether an event occurred $e_{t+1}$; and the miss distance $d_{t+1}$. While the state cannot be observed or set, the simulator transitions are deterministic given the pseudorandom seed $\bar{a}_t$. This property allows a previously visited state to be revisited by replaying the sequence of pseudorandom seeds $\bar{a}_0, \ldots, \bar{a}_t$ that leads to it starting from the initial state $s_0$, which we assume can be deterministically reached. In other words, we can use the sequence of pseudorandom seeds as the state. We use $s_t$ to denote the hidden state of the simulator and $\bar{s}_t$ to denote the sequence of seeds that induces $s_t$.

The seed-action simulator $\bar{\mathcal{S}}$ exposes the following simulation control functions:

- Initialize($\bar{\mathcal{S}}$) resets the simulator $\bar{\mathcal{S}}$ to a deterministic initial state $s_0$. The simulation state is modified in-place.
- Step($\bar{\mathcal{S}}, \bar{a}_t$) advances the state of the simulator by pseudorandom sampling. First, the pseudorandom seed $\bar{a}_t$ is used to set the state of the simulator's pseudorandom process. Second, a sample $a_t$ of the stochastic variables in $\bar{S}$ is drawn according to the natural distribution of the stochastic model. Third, the simulator transitions to the next state $s_{t+1}$ given the current state $s_t$ and action $a_t$, and the hidden state $s_t$ of the simulator is updated. The simulator returns the transition probability $p_{t+1}$ given by $P(a_t \mid s_t)$; whether an event occurred $e_{t+1}$; and the miss distance $d_{t+1}$.
- IsTerminal($\bar{\mathcal{S}}$) returns true if the current state of the simulator is a terminal state and false otherwise. The simulator terminates if an event occurs, i.e., $s_t \in E$, or if the simulation has reached a maximum number of time steps $t_{max}$.

Figure 5.3 illustrates the AST framework under the pseudorandom seed abstraction, where the simulator has been replaced by a seed-action simulator $\bar{\mathcal{S}}$, which maintains a hidden state. The seed input is used to draw a random action and state transition and the state is updated in-place. The simulator outputs the transition probability $p_{t+1}$, a Boolean indicating whether an event occurred $e_{t+1}$, and the miss distance $d_{t+1}$. The reward function translates the simulator outputs into a reward and the optimization algorithm learns from the reward to optimize over discrete pseudorandom seeds.

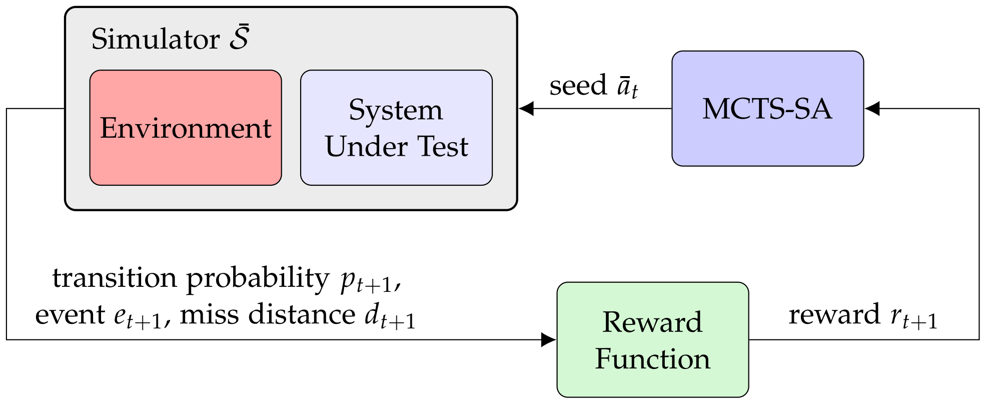

Simulator $\bar{\mathcal{S}}$
Environment
System Under Test
seed $\bar{a}_t$
MCTS-SA
transition probability $p_{t+1}$, event $e_{t+1}$, miss distance $d_{t+1}$
Reward Function
reward $r_{t+1}$

Figure 5.3: Adaptive stress testing of a partially observable system.

In addition to supporting hidden states, this abstraction also provides some practical benefits. Large software simulators are often written in a distributed and modular fashion where each component maintains its own state. The simulator may consist of many of these components. Since the simulator state is the concatenation of the states of all the individual components, explicitly assembling and handling the state can break modularity and be a major implementation inconvenience. This seed-action abstraction, which uses in-place state update and pseudorandom seeds as a proxy of the actions, enables the simulator to maintain modularity of the components. From an implementation perspective, it is very convenient that we do not need to expose all the stochastic models to the optimization algorithm. We only need the ability to set the global pseudorandom seed, which is generally easy to do in software simulators. From an algorithmic perspective, the ability to set the pseudorandom seed gives optimizers the ability make a stochastic simulator deterministic. The algorithm can deterministically return the simulator to any previously visited state by replaying the sequence of pseudorandom seeds.

### 5.4.2 Reward Function for Seed-Action Simulator

The reward function for AST of a seed-action simulator is given in Equation 5.2. The reward is expressed as a function of seed-action simulator outputs and a Boolean variable $\tau_t$ which indicates whether the simulator has terminated, i.e., $\tau_t = \textsc{IsTerminal}(\bar{\mathcal{S}})$. The three components mirror those in Equation 5.1. The indicator function $\mathbb{1}\{x\}$ returns 1 if $x$ is true and 0 otherwise.

$$\begin{aligned} R(p_t, e_t, d_t, \tau_t) = &+ R_E &&\cdot \mathbb{1}\{e_t\} \\ &- d_t &&\cdot \mathbb{1}\{\neg e_t \wedge \tau_t\} \\ &+ \log p_t &&\cdot \mathbb{1}\{\neg e_t \wedge \neg \tau_t\} \end{aligned} \tag{5.2}$$

### 5.4.3 Monte Carlo Tree Search for Seed-Action Simulator

In the fully observable case, we defined a generative model $G$, such that $(s_{t+1}, r_{t+1}) \sim G(s_t, a_t)$. That definition assumes that $G$ is stateless. However, due to the definition of the seed-action simulator, which assumes a simulator with hidden state, it is no longer possible to define a suitable $G$. Instead, we can define a stateful deterministic transition model $\bar{G}$, such that $(\bar{s}_{t+1}, r_{t+1}) \sim \bar{G}(\bar{s}_t, \bar{a}_t)$, where $\bar{a}_t$ is a pseudorandom seed, and $\bar{s}_t$ and $\bar{s}_{t+1}$ are sequences of pseudorandom seeds. The state $\bar{s}_t$ is the *action history* up to that time given by the sequence $\bar{a}_0, ..., \bar{a}_t$. That means $\bar{s}_{t+1}$ is independent of the state and computations inside $\bar{G}$, and can be computed solely from the inputs of $\bar{G}$ using the relation $\bar{s}_{t+1} = [\bar{s}_t, \bar{a}_t]$, i.e., appending $\bar{a}_t$ to the sequence. The only reliance on the internals of $\bar{S}$ comes from the evaluation of the reward function, which needs to know whether an event occurred, a miss distance, and a transition probability (for maximizing path probability). Table 5.2 summarizes the mapping from the problem of stress testing a seed-action simulator to its corresponding RL problem. While our proposed algorithm to be presented next does not make explicit use of $\bar{G}$ but rather refers to the functions of the seed-action simulator directly, this view is conceptually useful for understanding the design of the algorithm.

Table 5.2: Stress testing of seed-action simulator to RL problem mapping

| RL component | Stress testing component |
|---|---|
| State space $S$ | Sequences of pseudorandom seeds |
| Action space $A$ | Pseudorandom seeds |
| Stateful generative model $\bar{G}$ | Seed-action simulator functions INITIALIZE, STEP, and ISTERMINAL and seed-action reward function $R$ (Eq. 5.2) |
| Deterministic initial state $s_0$ | Achieved by calling INITIALIZE |

We now present Monte Carlo tree search for seed-action simulators (MCTS-SA). We base the algorithm on the double progressive widening variant of MCTS, which was discussed in Section 3.5. MCTS-DPW is used because the actions of the reinforcement learning agent are pseudorandom seeds, which are vast [46]. The transition behavior of the simulator is deterministic given a pseudorandom seed input. Consequently, only a single next state is possible and there is no need to limit the number of next states. We simplify the algorithm by removing the progressive widening of the state and its associated parameters, $k'$ and $\alpha'$. The action space is the space of all pseudorandom seeds. Since the seeds are discrete and do not have any semantic relationship, there is no need to distinguish between them. We choose the rollout policy and the action expansion function of MCTS to uniformly sample over all seeds. Sampling seeds uniformly generates unbiased samples of the next state from the simulator. We use $\mathbb{U}_{seed}$ to denote the discrete uniform distribution over all possible seeds. The hidden state $s_t$ of the simulator is not available to the reinforcement learning agent. However, since the simulator is deterministic given the pseudorandom seed input $\bar{a}_t$, we can revisit a previous state by replaying the sequence of seeds that leads to it starting from the initial state. As a result, we use the sequence of actions $\bar{a}_0, \ldots, \bar{a}_t$ as the state $\bar{s}_t$ in the algorithm. The path with the highest utility, that is, the sum of all the rewards received over the entire path, may, and likely will, be from a path that is encountered during a rollout. Since rollouts are not individually recorded, information about the best path can be lost. To ensure that the algorithm returns the best path seen over the entire search, we explicitly track the highest utility seen, $U^*$, and the corresponding action sequence, $\bar{s}^*$.

The MCTS-SA algorithm is shown in Algorithm 4. The algorithm takes as input a seed-action simulator $\bar{\mathcal{S}}$ and returns the most likely failure path $\bar{s}^*$. The algorithm consists of a main loop that repeatedly performs forward simulations of the system while building the search tree and updating the state-action value estimates. The search tree $\mathcal{T}$ is initially empty. Each simulation runs from initial state to a terminal state. The path is determined by the sequence of pseudorandom seeds chosen by the algorithm, which falls into three stages for each simulation:

- *Search.* In the search stage, which is implemented by SIMULATE (line 13), the algorithm starts at the root of the tree and recursively selects a child to follow. At each visited state node, the progressive widening criterion (line 22) determines whether to choose amongst existing actions (seeds) or to expand the number of children by sampling a new action. The criterion limits the number of actions at a state $\bar{s}$ to be no more than polynomial in the total number of visits to that state. Specifically, a new action $\bar{a}$ is sampled from a discrete uniform distribution over all seeds $\mathbb{U}_{seed}$ if $\|A(\bar{s})\| < kN(\bar{s})^{\alpha}$, where $k$ and $\alpha$ are parameters, $\|A(\bar{s})\|$ is the number of previously visited actions at state $\bar{s}$, and $N(\bar{s})$ is the total number of visits to state $\bar{s}$. Otherwise, the existing action that maximizes

$$Q(\bar{s}, \bar{a}) + c\sqrt{\frac{\log N(\bar{s})}{N(\bar{s}, \bar{a})}} \tag{5.3}$$

  is chosen (line 27), where $c$ is a parameter that controls the amount of exploration in the search, and $N(\bar{s}, \bar{a})$ is the total number of visits to action $\bar{a}$ in state $\bar{s}$. Equation 5.3 is the UCT equation [36]. The second term in the equation is an *exploration bonus* that encourages selecting actions that have not been tried as frequently. The action is used to advance the simulator to the next state and the reward is evaluated. The search stage continues in this manner until the system transitions to a state that is not in the tree.
- *Expansion.* Once we have reached a state that is not in the tree $\mathcal{T}$, we create a new node for the state and add it (line 18). The set of previously taken actions from this state, $A(\bar{s})$, is initially empty and the number of visits to this state $N(\bar{s})$ is initialized to zero.
- *Rollout.* Starting from the state created in the expansion stage, we perform a *rollout* that repeatedly samples state transitions until the desired termination is reached (line 20). In the ROLLOUT function (line 36), state transitions are drawn from the simulator with actions chosen according to a rollout policy, which we set as the discrete uniform distribution over all seeds $\mathbb{U}_{seed}$.

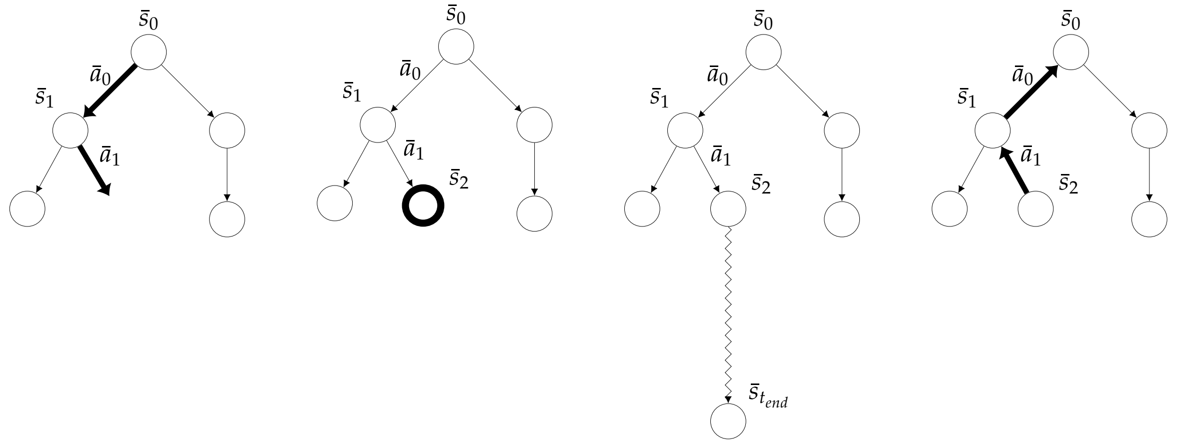
At each step in the simulation, the reward function is evaluated and the reward is used to update estimates of the state-action values $Q(\bar{s}, \bar{a})$ (line 34). The values are used to direct the search. At the end of each simulation, the best utility $U^*$ and best path $\bar{s}^*$ are updated (lines 10–11). Simulations are run until the stopping criterion is met. The criterion is a fixed number of iterations for all our experiments in this thesis except for the performance study in Section 8.6 where we used a fixed computational budget.

**Algorithm 4** MCTS for seed-action simulators

```
▷ Inputs: Seed-action simulator $\bar{\mathcal{S}}$
▷ Returns: Sequence of seeds that induces path with highest utility $\bar{s}^*$
function MCTS-SA($\bar{\mathcal{S}}$)
    global $\bar{s}_T \leftarrow \emptyset$
    $(\bar{s}^*, U^*) \leftarrow (\emptyset, -\infty)$
    loop
        $\bar{s} \leftarrow \emptyset$
        INITIALIZE($\bar{\mathcal{S}}$)
        $U \leftarrow$ SIMULATE($\bar{\mathcal{S}}, \bar{s}$)
        if $U > U^*$
            $(\bar{s}^*, U^*) \leftarrow (\bar{s}_T, U)$
    return $\bar{s}^*$
function SIMULATE($\bar{\mathcal{S}}, \bar{s}$)
    if ISTERMINAL($\bar{\mathcal{S}}$)
        $\bar{s}_T \leftarrow \bar{s}$
        return 0
    if $\bar{s} \notin \mathcal{T}$
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{\bar{s}\}$                    ▷ Expansion, add new node
        $(N(\bar{s}), A(\bar{s})) \leftarrow (0, \emptyset)$
        return ROLLOUT($\bar{\mathcal{S}}, \bar{s}$)
    $N(\bar{s}) \leftarrow N(\bar{s}) + 1$
    if $\|N(\bar{s}, \bar{a})\| < kN(\bar{s})^\alpha$                    ▷ Progressive widening of seeds
        $\bar{a} \sim \mathbb{U}_{seed}$                                   ▷ New seed
        $(N(\bar{s}, \bar{a}), V(\bar{s}, \bar{a})) \leftarrow (0, \emptyset)$
        $Q(\bar{s}, \bar{a}) \leftarrow 0$
        $A(\bar{s}) \leftarrow A(\bar{s}) \cup \{\bar{a}\}$
    $\bar{a} \leftarrow \text{argmax}_a\, Q(\bar{s}, a) + c\sqrt{\frac{\log N(\bar{s})}{N(\bar{s}, a)}}$     ▷ UCT selection criterion
    $(p, e, d) \leftarrow$ STEP($\bar{\mathcal{S}}, \bar{a}$)                  ▷ Deterministic transition
    $\tau \leftarrow$ ISTERMINAL($\bar{\mathcal{S}}$)
    $r \leftarrow$ REWARD($p, e, d, \tau$)
    $\bar{s}' \leftarrow [\bar{s}, \bar{a}]$
    $q \leftarrow r +$ SIMULATE($\bar{\mathcal{S}}, \bar{s}'$)
    $N(\bar{s}, \bar{a}) \leftarrow N(\bar{s}, \bar{a}) + 1$
    $Q(\bar{s}, \bar{a}) \leftarrow Q(\bar{s}, \bar{a}) + \frac{q - Q(\bar{s}, \bar{a})}{N(\bar{s}, \bar{a})}$          ▷ Update estimate of Q
    return $q$
function ROLLOUT($\bar{\mathcal{S}}, \bar{s}$)
    if ISTERMINAL($\bar{\mathcal{S}}$)
        $\bar{s}_T \leftarrow \bar{s}$
        return 0
    $\bar{a} \sim \mathbb{U}_{seed}$                                       ▷ Sample seeds uniformly
    $(p, e, d) \leftarrow$ STEP($\bar{\mathcal{S}}, \bar{a}$)
    $\tau \leftarrow$ ISTERMINAL($\bar{\mathcal{S}}$)
    $r \leftarrow$ REWARD($p, e, d, \tau$)
    $\bar{s}' \leftarrow [\bar{s}, \bar{a}]$
    return $r +$ ROLLOUT($\bar{\mathcal{S}}, \bar{s}'$)
```

The algorithm returns the path with the highest utility represented as a sequence of pseudorandom seeds. The sequence of seeds can be used to replay the simulator to reproduce the failure events.

Figure 5.4 illustrates the four primary operations of the MCTS-SA algorithm. The three stages of each simulation, which include search, expansion, and rollout, are shown in Figures 5.4a–5.4c and the update operation is shown in Figure 5.4d. The discussion follows closely that of Figure 3.3 with the exception that actions are no longer explicitly represented as nodes since state transitions are now deterministic. The actions are pseudorandom seeds and the states are sequences of seeds. In the search stage, shown in Figure 5.4a, the algorithm recursively selects a child to follow using the progressive widening and UCT criteria until it reaches a state that has not been previously visited. The selected states and actions are labeled $\bar{s}_0$, $\bar{a}_0$, $\bar{s}_1$, and $\bar{a}_1$ in the example. A new node representing the unvisited state, labeled $\bar{s}_2$ in Figure 5.4b, is added to the tree in the expansion stage. Starting from the newly added node, a rollout continues the simulation until a terminal state ($\bar{s}_{t_{end}}$ in Figure 5.4c) is reached by selecting actions uniformly from $\mathbb{U}_{seed}$. Finally, in Figure 5.4d, the observed rewards are used to update the Q-values at the nodes along the simulated path. The Q-values are used in the UCT criterion in the search stage.



(a) Search and follow child node. (b) Expand node of unvisited state. (c) Rollout to end using actions from $\mathbb{U}_{seed}$. (d) Update Q-values for nodes along path.

Figure 5.4: Primary operations in Monte Carlo tree search for state-action simulators.

**Computational Complexity.** Each iteration of the MCTS main loop simulates a path from initial state to terminal state. As a result, the number of calls to the simulator is linear in the number of loop iterations. The computation time thus varies as $O(N_{loop} \cdot (T_{\text{INITIALIZE}} + N_{steps} \cdot T_{\text{STEP}}))$, where $N_{loop}$ is the number of loop iterations, $T_{\text{INITIALIZE}}$ is the computation time of INITIALIZE, $N_{steps}$ is the average number of steps in the simulation, and $T_{\text{STEP}}$ is the computation time of the STEP function.

## 5.5 Summary

This chapter proposed a novel approach for finding the most likely failure scenarios of a system. The approach, called AST, formulates the stress testing problem as a sequential decision problem so that reinforcement learning algorithms can be applied for search. We presented an AST formulation for systems with fully observable state, which can be directly mapped to a traditional reinforcement learning problem. We also presented an AST formulation for systems with partially or fully hidden state. In this case, we proposed an abstraction called a seed-action simulator, that uses pseudorandom seeds to control the pseudorandom number generation process of the simulator. This abstraction creates a deterministic but non-Markov model, which can be optimized using our proposed MCTS-SA algorithm. By searching the simulation space using pseudorandom number seeds, the simulator can be a black box with both the simulation state and transition behavior hidden.

# Chapter 6

# Finding Failures Relative to a Baseline System

The previous chapter presented AST, a framework for finding failure scenarios in a system. This chapter presents differential adaptive stress testing (DAST), an extension to AST for comparing the failure behavior of two systems. DAST is a core contribution of this thesis. After providing a brief introduction and motivation in Section 6.1, Section 6.2 presents the DAST formulation and solution method. Section 6.3 discusses the contributions of the chapter.

## 6.1 Introduction

The previous chapter presented AST, a framework for efficiently finding failure scenarios of a system under test. In some applications, it may also be very valuable to identify failure scenarios not in absolute terms, but compared to another system—that is, areas where the system is *relatively* weak compared to a baseline system. In other words, we are not so interested in the cases where both systems perform poorly as cases where the system under test performs poorly but the baseline system performs well. We call this type of analysis *differential analysis* as shown in Table 6.1. Such situations arise, for example, when comparing two candidate solutions to see which one may be more desirable for release. Another use case is for regression testing where a new version of a system is compared to a previous one to see whether any new issues have been introduced.

Table 6.1: Differential analysis

|  | System Under Test Succeeds | System Under Test Fails |
|---|---|---|
| Baseline Succeeds | - | *Differential Analysis* |
| Baseline Fails | - | - |

One way to compare the behavior of two systems is to evaluate them against a common set of testing scenarios. For example, testing inputs can be randomly drawn using Monte Carlo from a stochastic model of the system's operating environment. Then, the inputs can be applied to both systems and the scenarios where failure occurs in the system under test but not in the baseline system are kept. While this method can generate failures, the undirected nature of this approach can be very inefficient due to the size and complexity of the state space, and the rarity of failure events. Moreover, the method does not find the most likely path to a failure, which is very valuable in the analysis of failure events.

Another, perhaps slightly better, approach is to use a stress testing method, such as AST, to identify failure scenarios in the system under test, and then replay the scenario on the baseline system. If the scenario does not fail on the baseline system, then accept the scenario. This method improves upon the first method in that at least the failure scenarios on the system under test are being optimized. However, the optimization process does not take into account the behavior of the baseline system.

We propose a novel stress testing method, called DAST, that extends the AST framework to search both the system under test and the baseline system simultaneously. DAST extends the AST framework to the differential analysis setting while retaining its desirable properties, including scalability, efficiency, and support for black box systems. Similar to the AST approach, we formulate the search as a sequential decision-making problem and then use reinforcement learning algorithms to optimize it.

## 6.2 Differential Adaptive Stress Testing

This section presents differential adaptive stress testing (DAST), a stress testing method for efficiently finding the most likely path to a failure event that occurs in the system under test, but not in the baseline system [77]. The key idea behind DAST is to drive two simulators in parallel and maximize the difference in their outcomes. To achieve this, we craft a new reward function that accepts the outputs of two simulators and encourages failures in one simulator but not the other. For optimization, we regard the two parallel simulators as a larger combined simulator, then we follow the AST approach to formulate stress testing as a sequential decision-making problem and optimize it using reinforcement learning. One of the core advantages of the seed-action formulation introduced in Section 5.4.1 is that it uses control of the pseudorandom seed to abstract the internal state and transition behavior from the optimization procedure. By composing two seed-action simulators into a combined simulator that is also a seed-action simulator, we can apply the MCTS-SA algorithm described in Algorithm 4 for optimization without any modification.

Figure 6.1 illustrates the DAST framework. We create two instances of the simulator $\bar{\mathcal{S}}^{(1)}$ and $\bar{\mathcal{S}}^{(2)}$. The instances are identical except that $\bar{\mathcal{S}}^{(1)}$ contains the system under test, while $\bar{\mathcal{S}}^{(2)}$ contains the baseline system. In particular, they contain identical models of the environment with which the system under test and the baseline system interface. The simulators are driven by the same pseudorandom seed input, which leads to the same sequence of stochastic variables being drawn from the environment when the behaviors of the test and baseline systems match. Under these circumstances, the states $s_t$ match and thus the distribution $P(a_t \mid s_t)$ also matches leading to identical samples drawn under the same pseudorandom seed. When the behavior of the two systems diverge, the states $s_t$ also diverge. The distribution $P(a_t \mid s_t)$ may change due to conditioning on a different $s_t$, but the seed automatically allows different stochastic variables $a_t$ to be drawn in each simulator without requiring any synchronization between the simulators.

We define a combined simulator that contains the two parallel simulators $\bar{\mathcal{S}}^{(1)}$ and $\bar{\mathcal{S}}^{(2)}$. They are both driven by the same input seed $\bar{a}_t$. Each simulator produces its own set of outputs, which include the transition probability $p_{t+1}$, an indicator of whether an event occurred $e_{t+1}$, and the miss distance $d_{t+1}$. These variables are combined in a reward function, where a single reward is provided to the reinforcement learner. Finally, the MCTS-SA algorithm chooses seeds to maximize the reward it receives. The superscripts on the variables $p_{t+1}, e_{t+1}, d_{t+1}$ and $\tau_{t+1}$ indicate the associated simulator.

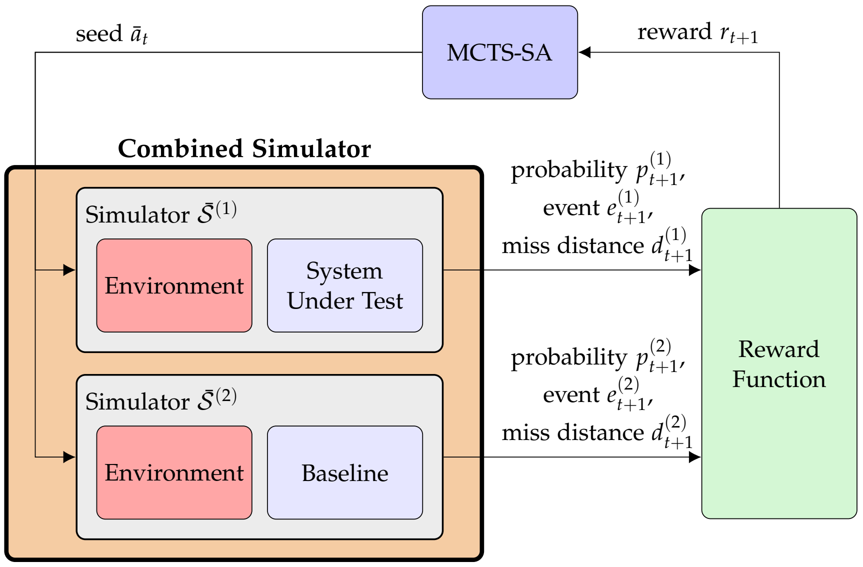

seed $\bar{a}_t$
MCTS-SA
reward $r_{t+1}$
Combined Simulator
Simulator $\bar{\mathcal{S}}^{(1)}$
Environment
System Under Test
Simulator $\bar{\mathcal{S}}^{(2)}$
Environment
Baseline
probability $p^{(1)}_{t+1}$, event $e^{(1)}_{t+1}$, miss distance $d^{(1)}_{t+1}$
probability $p^{(2)}_{t+1}$, event $e^{(2)}_{t+1}$, miss distance $d^{(2)}_{t+1}$
Reward Function

Figure 6.1: Differential adaptive stress testing framework.

### 6.2.1 Reward Function

The reward function combines the output from the two individual simulators to produce a single reward for the MCTS-SA learner. The primary objective of the reward function is to maximize the difference in outcomes of the simulators driving the first simulator to a failure event, while keeping the second simulator away from one. The secondary objective is to maximize the path probabilities of the two simulators to produce the most likely paths. The DAST reward function is given by Equation 6.1. The indicator function $\mathbb{1}\{x\}$ returns 1 if $x$ is true and 0 otherwise.

$$\begin{aligned}
R(p_t^{(1)}, e_t^{(1)}, d_t^{(1)}, \tau_t^{(1)}, p_t^{(2)}, e_t^{(2)}, d_t^{(2)}, \tau_t^{(2)}) = &+ R_E && \cdot \mathbb{1}\{e_t^{(1)}\} \\
&- d_t^{(1)} && \cdot \mathbb{1}\{\neg e_t^{(1)} \wedge \tau_t^{(1)}\} \\
&- R_E && \cdot \mathbb{1}\{e_t^{(2)}\} \\
&+ d_t^{(2)} && \cdot \mathbb{1}\{\neg e_t^{(2)} \wedge \tau_t^{(2)}\} \\
&+ (\log p_t^{(1)} + \log p_t^{(2)}) && \cdot \mathbb{1}\{\neg e_t^{(1)} \wedge \neg e_t^{(2)} \wedge \neg \tau_t\}
\end{aligned} \tag{6.1}$$

The DAST reward function extends the AST reward function to the differential setting and has a similar structure. The first term in Eq. 6.1 gives a non-negative reward $R_E$ to the learner if the first simulator $\bar{\mathcal{S}}^{(1)}$ terminates in an event. If $\bar{\mathcal{S}}^{(1)}$ terminates and an event did not occur, then the second term penalizes the learner by giving the negative miss distance $-d_t^{(1)}$. The third and fourth terms are the negations of the first and second terms, respectively, applied to the second simulator $\bar{\mathcal{S}}^{(2)}$. The third term gives $-R_E$ if $\bar{\mathcal{S}}^{(2)}$ terminates in an event and the fourth term gives $d_t^{(2)}$ if $\bar{\mathcal{S}}^{(2)}$ terminates and an event did not occur. To maximize the probabilities of the paths, the fifth term gives the sum of the log transition probabilities of both simulators. The terminal state of the simulators are treated as *absorbing*. That is, once a simulator enters a terminal state, it stays there for all subsequent transitions and collects zero reward for these transitions. The Boolean variables $\tau_t^{(1)}$ and $\tau_t^{(2)}$ indicate whether $\bar{\mathcal{S}}^{(1)}$ and $\bar{\mathcal{S}}^{(2)}$ have terminated, respectively. The combined simulator terminates when both simulators have terminated, i.e., $\tau_t = \textsc{IsTerminal}(\bar{\mathcal{S}}) = \tau_t^{(1)} \wedge \tau_t^{(2)}$, where $\tau_t$ (without superscript) indicates whether the combined simulator has terminated.

### 6.2.2 Optimization

Our DAST formulation combines two seed-action simulators into a single combined simulator that is also a seed-action simulator by defining the behavior of the combined simulator in terms of the individual simulators. For example, the reward function in Eq 6.1 combines the output of the two simulators into a single reward output. As a result, we can optimize the problem using the same MCTS-SA algorithm described in Algorithm 4. Because the algorithm is based on scalar rewards and pseudorandom seeds, the internal details of the simulator are abstracted from the reinforcement learner. As a result, no modifications to the algorithm presented in Section 5.4.3 are necessary.

## 6.3 Summary

This chapter proposed a novel approach for performing differential failure analysis between two systems. We find the most likely failure scenarios of a system that occurs in one system, but not in another baseline system. The approach, called DAST, searches two simulators simultaneously and drives their final outcomes apart. The problem is posed as a sequential decision-making problem in a similar manner to AST and the same seed-action simulator abstraction is used. As a result, the same algorithm, MCTS-SA, can be used for optimization as in the case of AST in Chapter 5. DAST addresses the problem of comparing the failure behaviors of two systems, which arises naturally in candidate selection and regression testing applications. DAST can search much more efficiently than existing methods because it accounts for the behavior of both systems during the optimization process.

# Chapter 7

# Categorizing and Explaining Failures

The previous two chapters discussed methods for finding the most likely failure scenarios for a system under test. This chapter discusses methods for analyzing failure datasets to identify common patterns that may reveal underlying issues in the system. We present grammar-based decision tree (GBDT), an interpretable model for automatically categorizing the failure data and explaining relevant properties to a human. GBDT is a core contribution of this thesis. The first section, Section 7.1, motivates the need for automatic organization and explanation of failure data. Section 7.2 presents the GBDT framework, and Section 7.3 presents a training algorithm for GBDT. Section 7.4 demonstrates GBDT on an example dataset on Australian sign language, and the final section, Section 7.5, summarizes the contributions of this chapter.

## 7.1 Introduction

Large and high-dimensional datasets have necessitated increased automation to help humans in analyzing data. Automated analysis tools can help humans extract useful insights from the data by automatically organizing the data into a more human-friendly form, and finding interesting patterns in the data. For example, analysis of large datasets of failures can reveal common failure patterns, identify underlying issues, and ultimately lead to better mitigations. In Chapter 4, we provided a general overview of approaches to interpretable machine learning. This chapter focuses on interpretable machine learning approaches for knowledge discovery from heterogeneous multivariate time series data.

Heterogeneous multivariate time series data arises in many applications including driverless cars, smart homes, robotic servants, and aviation. Understanding these datasets is important for designing better systems, validating safety, and analyzing failures. However, knowledge discovery from heterogeneous multivariate time series datasets is very challenging because it typically requires two major data mining problems to be addressed simultaneously. The first problem is how to handle multivariate *heterogeneous* time series data, where the variables are a mix of numeric, Boolean, and categorical types. The second problem is the need for *interpretability*, which is closely related to concepts such as explainability, understandability, and trustworthiness. That is, humans must be able to understand and reason about the information captured by the model. While these problems have been explored separately in the literature, we are not aware of any methods that address both interpretability and heterogeneous multivariate time series datasets together.

Rule-based methods such as decision trees [52] and decision lists [53] are very intuitive because they use symbolic rules for decision boundaries. However, they do not traditionally support time series data. Motif discovery methods, such as shapelet [55] and subsequence [56] discovery, find recurring patterns in sequential data. These methods are less interpretable than rule-based methods as they report prototypes of patterns rather than state relationships. These methods have also not been applied to heterogeneous multivariate time series data. On the other hand, models that support heterogeneous multivariate time series, such as recurrent neural networks [78], are not interpretable. If familiar physics-based models exist, parameters of the model can be fit using data [47]. The advantage of this approach is that parameters often have physical meaning and can be independently tested and verified. However, a physics-based model may not be generally available.

To simultaneously address the problems of interpretability and heterogeneous time series data, we increase the expressiveness of decision trees by allowing their decision rules or splitting criteria to be any logical expression. Traditional decision trees, as introduced in Section 4.3, partition the input space using the Boolean feature itself or using simple thresholds on real-valued features, such as $(x_1 < 2)$ [52]. However, these partitioning rules have limited expressiveness and cannot be used to express more complex logical relationships, such as those between heterogeneous features or across time. Allowing more expressive rules in the decision tree increases its modeling capability while retaining interpretability. In the past, first order logic rules have been used in decision trees [79]. We generalize this idea and allow any logical language that can be specified using a context-free grammar. In particular, we show that decision trees combined with temporal logic can be very effective as an interpretable model for heterogeneous multivariate time series data. Temporal logic, in combination with other algorithms, has been applied to heterogeneous time series data in aviation [80], but not in our decision tree setting.

This chapter presents grammar-based decision tree (GBDT), which combines decision trees with formal grammars. The innovative idea of GBDT is to generalize the decision rules of a decision tree to arbitrary logical expressions that are derived from a CFG. While a CFG defining any language can be used, we make the particular choice of using expressions derived from temporal logic, which can elegantly handle heterogeneous multivariate time series data [7]. Deriving expressions from a CFG also aligns well with the expression optimization framework presented in Chapter 4. Due to the use of a decision tree as a model, GBDT can be used for both interpretable classification and *categorization*, which is the combined task of clustering data into similar groups and providing explanations for them. We elaborate on the categorization capability later in Section 7.2.

The combination of decision trees and grammars has been proposed in the past [81, 82]. These prior works use a grammar to optimize traditional decision trees as a whole where the splits are simple thresholds over individual features. The resulting tree, after machine learning, is a traditional decision tree. In contrast, GBDT uses a grammar and expression optimization to optimize decision expressions, and the resulting decision tree contains a (temporal) logic rule at each decision node. This distinction is very important because the added expressivity of the logical expressions is what allows support for multivariate heterogeneous time series datasets. While the GBDT model generally has higher complexity, the complexity is localized to logical expressions with higher representational abstraction and the expressions are organized in an easy-to-understand tree structure. As a result, GBDT is much more interpretable than, say, a random forest classifier, which increases model expressivity through increasing the number of trees.

## 7.2 Grammar-Based Decision Trees

Grammar-based decision tree (GBDT) is an interpretable machine learning model that extends decision trees with a grammar framework to allow general logical expressions to be used as the branching rules of a decision tree. The domain of the logical expressions is constrained using a CFG given by the user, which gives the user fine control of the underlying syntax and semantics of the expressions. In this thesis, we consider grammars based on temporal logic for the modeling of heterogeneous multivariate time series data. However, any CFG producing logical expressions can be used. While decision trees can be and are traditionally used for classification, we show that GBDT can also be very useful for categorization. In particular, our primary interest is in knowledge discovery by first training a GBDT on a large dataset, and then analyzing the categories and the explanations of the categories that were learned by the model.

### 7.2.1 Notation

A multi-dimensional time series dataset $D$ consists of $m$ records of feature matrix $X^j$ and label $l^j$ pairs $((X^1, l^1), (X^2, l^2), ...(X^m, l^m))$. Each feature matrix $X^j$ is a two-dimensional $n \times T$ matrix of $n$ features by $T$ time steps and a label $l^j$ is its discrete class label. A feature trace $\vec{x}_i$ is the $i$'th row of a feature matrix and represents the time series of feature $x_i$. Logical and comparison operators are applied elementwise. For example, the logical operator in $\vec{x}_i \wedge \vec{x}_j$ takes the elementwise conjunction of the two vectors. Elementwise operators are given broadcast semantics where appropriate. Broadcast semantics extend an elementwise operator to handle a vector and scalar input (rather than two vectors) by replicating the scalar input so that it has the same number of elements as the vector input. For example, the comparison operator in $\vec{x}_i < c$ compares each element of $\vec{x}_i$ to $c$ and returns a vector of the same size as $\vec{x}_i$. The temporal operators F and G are *eventually* and *globally*, respectively. *Eventually* returns true if any value in the argument vector is true. *Globally* returns true if all values in the argument vector are true.

### 7.2.2 Model

GBDT is a binary decision tree where each non-leaf node contains a logical expression. The children of a non-leaf node are associated with each of the Boolean outcomes when evaluating the logical expression at that node. Leaf nodes are associated with a particular class label for prediction. We use the following definition: A GBDT $\mathcal{T}$ is either a leaf with class $l$, which we denote by $\mathcal{T} = \textsc{Leaf}(l)$; or it is an internal node with expression $e \in G$, true branch $\mathcal{T}^+$, and false branch $\mathcal{T}^-$, which we denote by $\mathcal{T} = \text{I}\textsc{Node}(e, \mathcal{T}^+, \mathcal{T}^-)$. Both $\mathcal{T}^+$ and $\mathcal{T}^-$ are GBDTs.

#### GBDT Prediction

As in a traditional decision tree, GBDT prediction is performed in a top-down fashion starting at the root. At each node, the logical expression is evaluated on the feature matrix and the result dictates which branch is followed. The process continues recursively until a leaf node is reached where the class label at the leaf node is used for prediction. The process is the same as with a traditional decision tree except for the more sophisticated logical expressions applied at each internal node.

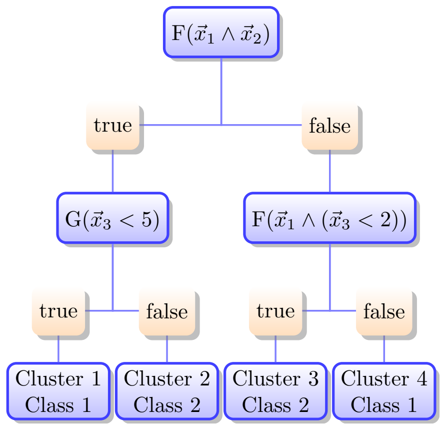
#### GBDT Categorization

While decision trees are traditionally used for classification, they can also be used for categorization, which is the combined task of clustering and explaining data. A categorization of the data can be extracted from a GBDT by considering each leaf node of the tree to be a separate category. The description of the category is then the conjunction of all branch expressions between the root and the leaf node of interest. The members of the cluster are the records where the cluster's description holds true. Since partitions are mutually exclusive, the clusters do not overlap.

**Example GBDT Model**

Figure 7.1 shows an example of a GBDT with an accompanying CFG. The splitting rules learned for the nodes in the tree in Figure 7.1a are in fact expressions of the grammar in Figure 7.1b. The CFG describes a simple temporal logic. It assumes that the data has four features $x_1, x_2, x_3, x_4$, where the feature traces $\vec{x}_1$ and $\vec{x}_2$ are Boolean vectors and $\vec{x}_3$ and $\vec{x}_4$ are vectors of real numbers. The grammar contains a set of production rules. The rules can contain expressions with a mix of terminal and non-terminal symbols such as $\mathrm{F}(\vec{B})$, where $\vec{B}$ is a non-terminal symbol and F is a terminal symbol for operator $\mathrm{F}(\cdot)$. Non-terminal symbols are substituted further using the appropriate production rules. The production rules can contain the symbol $|$, such as $Ev \mid Gl$, which delineates the different possible substitutions.

Each non-leaf node of the GBDT contains a logical expression derived from the grammar. The expression dictates which branch should be followed. Leaf nodes show the predicted class label. In Figure 7.1, a data record where $\mathrm{F}(\vec{x}_1 \wedge \vec{x}_2)$ is true and $\mathrm{G}(\vec{x}_3 < 5)$ is false would be predicted to have class label 2. We also label each leaf node with a unique cluster number. For example, the rightmost leaf node in Figure 7.1a is labeled as cluster 4. The overall expression that describes cluster 4 is $\neg\mathrm{F}(\vec{x}_1 \wedge \vec{x}_2) \wedge \neg\mathrm{F}(\vec{x}_1 \wedge (\vec{x}_3 < 2))$.


$\mathrm{F}(\vec{x}_1 \wedge \vec{x}_2)$
true
false
$\mathrm{G}(\vec{x}_3 < 5)$
$\mathrm{F}(\vec{x}_1 \wedge (\vec{x}_3 < 2))$
true
false
true
false
Cluster 1 Class 1
Cluster 2 Class 2
Cluster 3 Class 2
Cluster 4 Class 1

(a) Tree

$$\begin{aligned}
B &::= Ev \mid Gl \\
Ev &::= \mathrm{F}(\vec{B}) \\
Gl &::= \mathrm{G}(\vec{B}) \\
\vec{B} &::= \vec{x}_1 \mid \vec{x}_2 \mid And \mid Lt \\
And &::= \vec{B} \wedge \vec{B} \\
Lt &::= \vec{R} < C \\
\vec{R} &::= \vec{x}_3 \mid \vec{x}_4 \\
C &::= 1 \mid 2 \mid 3 \mid 4 \mid 5
\end{aligned}$$

(b) Grammar

Figure 7.1: Grammar-based decision tree example. The node expressions in the tree (a) are derived from the grammar (b). Each leaf node is a separate cluster.

### 7.2.3 Grammar Design for Heterogeneous Time Series

Grammar design is crucial to the effectiveness of GBDT. The ideal grammar should produce expressions that are interpretable, tailored to the application, and support the feature types present in the dataset. This section aims to offer design suggestions to the user to achieve these goals. The exact details of the grammar will depend on the specific needs of the user.

In the GBDT framework, logic expressions are evaluated on traces and produce a Boolean output. The symbols of the expressions can refer to features, constants, or functions. We adopt a subset of linear temporal logic (LTL), a formal logic with temporal operators often used in the analysis of time series data [7]. Temporal logic provides an elegant framework for defining expressions that support heterogeneous multivariate time series data. For example, the logic rule $\mathrm{F}(\vec{x}_1 \wedge (\vec{x}_3 < 2))$ in Figure 7.1 contains a Boolean vector $\vec{x}_1$ and a vector of real values $\vec{x}_3$. The rule integrates these two different data types elegantly by first evaluating $(\vec{x}_3 < 2)$ using an elementwise comparison to produce a Boolean vector. Then, the result is combined elementwise via conjunction with $\vec{x}_1$ to produce another Boolean vector. Finally, the temporal operator F collapses the Boolean vector to a single Boolean output. The constants in the grammar can be preselected based on discretizing over the range of each real-valued feature in the dataset. Alternatively, some expression optimization algorithms, such as Monte Carlo and genetic programming, also support drawing random values during search [71].

When designing a grammar, it is important to manage data types. For example, the grammar in Figure 7.1 is organized by the data type of its non-terminals. The production rule for non-terminal $B$ selects amongst functions that return a Boolean, whereas the production rule for non-terminal $\vec{B}$ selects amongst functions and expressions that return a vector of Booleans. Similarly, the non-terminal $\vec{R}$ represents a vector of real values while $C$ represents a constant. Organization of types is not only important to provide functions with valid inputs but also key to meaningfully nest and combine heterogeneous types.

For simplicity, the example in Figure 7.1 included only a small number of operators. The grammar can be readily extended to include other operators including disjunction $\vee$, greater than $>$, equals $=$, where equality is important for categorical features, and even custom user-defined functions. Depending on the application, a generic temporal logic grammar can be used (as we will see in Section 7.4) or the grammar can be tailored to the application (as we will see later in Section 8.5). This paper considers very simple temporal operators that operate over traces. More sophisticated temporal logics, such as metric temporal logic (MTL) [83], can be used with GBDT to discover more intricate patterns in the data. Nesting of temporal operators can also be permitted.

### 7.2.4 Natural Language Descriptions

Logical expressions can sometimes be dense and hard to parse for humans. In many cases, we can improve human interpretability by providing natural language descriptions of the expressions. One method to automatically translate expressions into English sentences is to map expression rules and terminal symbols in the CFG to corresponding sentence fragments and then use the structure of the expression's derivation tree to assemble the fragments. Figure 7.2 shows an example mapping that could be used with the grammar in Figure 7.1 for mapping temporal logic expressions into English sentences.

$$\begin{aligned}
\mathrm{F}(\vec{B}) &:= \text{“at some point, } \vec{B}\text{"} \\
\mathrm{G}(\vec{B}) &:= \text{“for all time, } \vec{B}\text{"} \\
\vec{B} \wedge \vec{B} &:= \text{“} \vec{B} \text{ and } \vec{B}\text{”} \\
\vec{R} < C &:= \text{“} \vec{R} \text{ is less than } C\text{"} \\
\vec{x}_1 &:= \text{“advisory is active"} \\
\vec{x}_2 &:= \text{“pilot is responding"} \\
\vec{x}_3 &:= \text{“vertical rate"} \\
\vec{x}_4 &:= \text{“airspeed"}
\end{aligned}$$

Figure 7.2: Natural language map example.

Applying the mapping in Figure 7.2, the decision rules in Figure 7.1 become the following natural language descriptions: $\mathrm{F}(\vec{x}_1 \wedge \vec{x}_2)$ is translated to “at some point, [advisory is active] and [pilot is responding]"; $\mathrm{G}(\vec{x}_3 < 5)$ is translated to “for all time, [vertical rate] is less than 5"; and $\mathrm{F}(\vec{x}_1 \wedge (\vec{x}_3 < 2))$ is translated to “at some point, [advisory is active] and [[vertical rate] is less than 2]". We include square brackets to help the reader disambiguate nested sentence components.

## 7.3 Training a GBDT

Training a GBDT is performed top-down as in traditional decision tree training. However, rather than cycling through the features to determine the best split, we use expression optimization as a subroutine to find the best partition expression. The training algorithm begins with a single (root) node containing the entire dataset $D$. Expression optimization is then used to search a CFG $G$ for the partitioning expression $e$ that yields the best loss given by the loss function $f$. The expression is evaluated on each record and the dataset is partitioned into two child nodes according to the results of the evaluation. The process is applied recursively to each child until all records at the node are either correctly classified or a maximum tree depth $d$ is reached. The mode of the training labels is used for class label prediction at a leaf node. The GBDT training algorithm is shown in Algorithm 5.

In Algorithm 5, GBDT (line 2) is the main entry point to the training algorithm. It returns a Tree object containing the root node of the induced decision tree. Split (line 5) attempts to partition the data into two parts. It first tests whether the terminal conditions are met and if so returns a Leaf node that predicts the mode of the labels. The partitioning terminates if the maximum depth $d$ has been reached or if all class labels are the same, which is tested by the IsHomogeneous function (line 6). The ExprOpt function

**Algorithm 5** Grammar-Based Decision Tree Training

```
1: ▷ Inputs: CFG $G$, loss function $f$, dataset $D$, depth $d$
2: function GBDT($G, f, D, d$)
3:     $R \leftarrow$ SPLIT($G, f, D, d$)
4:     return TREE($R$)
5: function SPLIT($G, f, D, d$)
6:     if ISHOMOGENEOUS(LABELS($D$)) or $d = 0$
7:         return LEAF(MODE(LABELS(D)))
8:     $e \leftarrow$ EXPROPT($G, f, D$)
9:     $D^+ \leftarrow [(X, l) \mid$ EVALUATE$(e, X), (X, l) \in D]$
10:    $D^- \leftarrow [(X, l) \mid \neg$EVALUATE$(e, X), (X, l) \in D]$
11:    $child^+ \leftarrow$ SPLIT($G, f, D^+, d - 1$)
12:    $child^- \leftarrow$ SPLIT($G, f, D^-, d - 1$)
13:    return INODE($e, child^+, child^-$)
```

(line 8) uses expression optimization (presented in Section 4.4) to search for the expression that minimizes the loss function $f(e)$, i.e., $e = \text{argmin}_{e'} f(e')$, given the grammar $G$ and the dataset $D$. We partition the data into two parts conditioning on the result of evaluating $e$ on $D$ (lines 9–10). The function EVALUATE evaluates the expression $e$ on the feature matrix $X$ and produces a Boolean result. Then, SPLIT (lines 11–12) is called recursively on each part. Finally, the function returns an internal node INODE containing the decision expression and the children of the node (line 13).

### 7.3.1 Loss Function

In the expression optimization subroutine EXPROPT (line 9), we set the loss function to trade off between two competing objectives. On the one hand, we want expressions that partition the data so that the resulting leaf nodes are homogeneous, i.e., they have the same ground truth class labels. Splits that induce high homogeneity tend to produce shallower trees and thus shorter global expressions at leaf nodes, which improves interpretability. They also produce classifiers with better predictive accuracy when the maximum tree depth is constrained. To quantify homogeneity, we use the Gini impurity metric [52] discussed in Chapter 4. On the other hand, we want to encourage interpretability by minimizing the length and complexity of node expressions. Shorter and simpler node expressions are generally easier to interpret. We use the number of nodes in the derivation tree as a proxy for the complexity of an expression. The two objectives are combined linearly into a single loss function $f_{GBDT}$ given by

$$f_{GBDT}(e, D) = w_1 I_{Gini}(e, D) + w_2 N_{nodes}(e)$$

where $w_1 \in \mathbb{R}$ and $w_2 \in \mathbb{R}$ are weights, and $N_{nodes}(e)$ is the number of nodes in the derivation tree of $e$. The total Gini impurity, $I_{Gini}(e, D)$, is the sum of the Gini impurity of each partition that results from splitting the data using expression $e$. It is given by

$$I_{Gini}(e, D) = \sum_{L \in \{L^+, L^-\}} \sum_{b \in B} f_L^b (1 - f_L^b)$$

where $L^+$ are the labels of the records on which $e$ evaluates to true, i.e., $L^+ = [l \mid \textsc{Evaluate}(e, X), (X, l) \in D]$; $L^-$ are the labels of the records on which $e$ evaluates to false, i.e., $L^- = [l \mid \neg\textsc{Evaluate}(e, X), (X, l) \in D]$; $B = \{\text{True, False}\}$; and $f_L^b = {}^{\mathbb{1}\{l=b\}}/_{|L|}$, where $\mathbb{1}$ is the indicator function and $|L|$ denotes the number of elements in $L$. We use square brackets with set construction syntax to indicate that $L^+$ and $L^-$ are vectors and can have duplicate elements.

### 7.3.2 Computational Complexity

The most computationally expensive part of GBDT training is evaluating the loss of an expression at a node, since it involves visiting each record in the dataset and then computing statistics. Expression optimization also requires a large number of expression evaluations to optimize the decision expression at each decision node. The deeper the tree, the more decision node expressions need to be optimized. However, as a the depth of a node increases, it operates on an increasingly smaller fraction of the dataset. As a result, while the number of nodes grows exponentially with tree depth, the number of records that must be evaluated at each level remains constant (equal to the size of the dataset). Overall, the computational complexity of GBDT training is $O(m \cdot N_{EO} \cdot d)$, where $m$ is the number of records in the dataset, $N_{EO}$ is the number of logical expressions evaluated in each ExprOpt call, and $d$ is the depth of the decision tree.

## 7.4 Example: Australian Sign Language

We demonstrate the use of GBDT on the Australian Sign Language ("Auslan") dataset from the UC Irvine (UCI) repository [59, 84]. The data originates from participants wearing instrumented gloves while signing specific words in Australian sign language. The dataset is a multivariate time series of varying episode length containing 22 continuous features, 95 words with 27 examples each. We extract eight words (classes) from the dataset: *hello, please, yes, no, right, wrong, same, different*. First, we present the grammar used to define the expressions. Then we discuss the evaluation metrics and baseline algorithms used for evaluating the GBDT results. Finally, we present the experimental results.

### 7.4.1 Grammar

We use a generic temporal logic grammar that includes all features and the following operators: eventually F, globally G, implies $\implies$, negation $\neg$, disjunction $\vee$, conjunction $\wedge$, equals $=$, less than $<$, less than or equal to $\leq$, greater than $>$, and greater than or equal to $\geq$. Since features may have different ranges, constant values used in comparison operators must be specialized to each feature. To address this issue, we consider the range of each feature in the dataset and uniformly discretize it into 10 points. Features are compared with their corresponding set of discretized values. The data itself is not being discretized. Discretization is only used to generate the threshold constants in the grammar.

Figure 7.3 shows the grammar used in the Auslan experiment. The non-terminals of the grammar loosely represent a Boolean variable $B$, a Boolean vector $\vec{B}$, a real vector $\vec{R}$, a feature index $i_x$, and an index into a table of discretized constants $i_v$. The expression $X[i_x]$ retrieves the feature trace at index $i_x$, i.e., $\vec{x}_{i_x}$, from the data record $X$. The function $f_{op}(i_x, i_v)$ compares the feature trace at index $i_x$ to the $i_v$'th constant corresponding to that feature using operator $op$. Specifically, the function returns $X[i_x]$ $op$ $V[i_x, i_v]$, where $V$ is a table containing the discretized constants computed by feature from the data. The row of $V$ indicates the feature and the column of $V$ indicates the order position of the constant. For example, $V[roll\_1, 5]$ indicates the $5^{th}$ constant of the feature $roll\_1$. We use the shorthand $|(1:10)$ to denote that all integers between 1 and 10 inclusive are possible values.

$$
\begin{aligned}
&B ::= \mathrm{G}(\vec{B}) \mid \mathrm{F}(\vec{B}) \mid \mathrm{G}(\vec{B} \implies \vec{B})\\
&\vec{B} ::= (\vec{B} \wedge \vec{B}) \mid (\vec{B} \vee \vec{B}) \mid \neg\vec{B}\\
&\vec{B} ::= (\vec{R} < \vec{R}) \mid (\vec{R} \leq \vec{R}) \mid (\vec{R} >\mid R) \mid (\vec{R} \geq \vec{R})\\
&\vec{B} ::= f_{<}(i_x, i_v) \mid f_{\leq}(i_x, i_v) \mid f_{>}(i_x, i_v) \mid f_{\geq}(i_x, i_v)\\
&\vec{R} ::= X[i_x]\\
&i_x ::= x\_1 \mid y\_1 \mid z\_1 \mid roll\_1 \mid pitch\_1 \mid yaw\_1 \mid thumbbend\_1\\
&i_x ::= forebend\_1 \mid middlebend\_1 \mid ringbend\_1 \mid littlebend\_1\\
&i_x ::= x\_2 \mid y\_2 \mid z\_2 \mid roll\_2 \mid pitch\_2 \mid yaw\_2 \mid thumbbend\_2\\
&i_x ::= forebend\_2 \mid middlebend\_2 \mid ringbend\_2 \mid littlebend\_2\\
&i_v ::= |(1:10)
\end{aligned}
$$

Figure 7.3: Auslan grammar.

### 7.4.2 Evaluation Metrics

For classification performance, we report the accuracy, precision, recall, and F1 scores. We use *macro* averaging, which calculates the mean of the binary metrics, giving equal weight to each class. For interpretability, we report the average number of terminals in each rule of the decision tree. For example, $\text{F}(\vec{x}_1 \wedge \vec{x}_2)$ has four terminals: F, $\vec{x}_1$, $\wedge$, and $\vec{x}_2$. Rules containing fewer terminals are considered easier to interpret.

### 7.4.3 Baselines

We compare our algorithm against decision tree, random forest, and Gaussian naive Bayes classifiers from Scikit-Learn [85]. Random forest and Gaussian naive Bayes models have low interpretability. Because these baseline algorithms cannot directly ingest heterogeneous time series data, we preprocess the data before applying the baseline algorithms as follows: (1) $z$-score normalize each real-valued feature; (2) convert Boolean features to [-1.0,1.0]; (3) zero-pad so that all records have the same length in the time dimension; and (4) reshape the time series data (size $m \times n \times T$) into non-time series data (size $m \times nT$) by concatenating time slices, essentially converting time series data into static data. GBDT can directly handle heterogeneous multivariate time series data of varying length and thus does not require this preprocessing.

### 7.4.4 Results

We performed 20 random trials for each algorithm. A *trial* is a run of an experiment initialized with a unique random seed and train-test split. We use randomly-sampled 70/30 train-test splits of the data and report the mean of the evaluation metrics in Table 7.1. The best value for each metric is indicated in bold. Tree-based classifiers, i.e., GBDT, decision tree, and random forest models, were trained to a maximum depth $d = 6$ and random forest used 32 trees. Results for the various variants of GBDT are separately listed. GBDT-MC uses Monte Carlo algorithm, GBDT-GE uses grammatical evolution, and GBDT-GP uses genetic programming for expression optimization. Overall, GBDT-GP performed the best both in terms of classification and interpretability metrics. Decision tree performed poorly on this dataset because key characteristics of the features may be shifted in time due to variations in the speed of signing. Random forest can partially overcome this issue by using a larger number of trees. The temporal logic rules in GBDT capture the underlying temporal properties of the data much more effectively.

The strength of GBDT lies in its ability to categorize and explain data. GBDT not only provides interpretable expressions about the data's distinguishing properties but also a hierarchical organization of the properties. Figure 7.4 shows the resulting GBDT from the best-performing trial. We compare the learned model to the videos of the signed words to intuitively verify correctness[1]. At the root node, GBDT

[1] https://www.youtube.com/watch?v=_5NbYyUlcHU

Table 7.1: GBDT performance on Auslan dataset

| | GBDT-MC | GBDT-GE | GBDT-GP | Decision Tree | Random Forest | Gaussian NB |
|---|---|---|---|---|---|---|
| Accuracy | 0.9840 | 0.9847 | **0.9868** | 0.7585 | 0.9838 | 0.6862 |
| F1-Score | 0.9840 | 0.9845 | **0.9867** | 0.7085 | 0.9838 | 0.6674 |
| Precision | 0.9857 | 0.9872 | **0.9884** | 0.6869 | 0.9851 | 0.7025 |
| Recall | 0.9840 | 0.9847 | **0.9868** | 0.7672 | 0.9841 | 0.6867 |
| Avg. Terminals | 3.70 | 3.74 | **3.59** | — | — | — |

partitions *same* and *different* from the other words identifying the fully bent left pinky as the distinguishing property. Subsequently, *different* is distinguished from *same* by looking at whether the right palm is ever facing upwards. Of the remaining words, *hello* is isolated using the fully straight middle finger and then *wrong* is identified using the straight right pinky. The remaining four words are grouped *right* with *yes* and *please* with *no* using the yaw position of the right hand. The word *yes* is distinguished from *right* by the bent right thumb. Lastly, *no* is distinguished from *please* using the combined property on right forefinger bend, right yaw angle, and $x$-position of the right hand.

## 7.5 Summary

This chapter proposed a novel approach for knowledge discovery for heterogeneous multivariate time series datasets. We introduced a novel interpretable machine learning model, called GBDT, which extends decision trees with a grammar framework. In particular, we generalize the decision rules of the traditional decision tree to Boolean expressions derived from a context-free grammar, which allows temporal logic rules to be used within the decision tree framework. Expression optimization is used as a subroutine for training the GBDT. Our experiments applying GBDT on the Australian Sign Language dataset from UCI showed that GBDT not only provides good prediction accuracy, but also produces hierarchical and intuitive explanations.

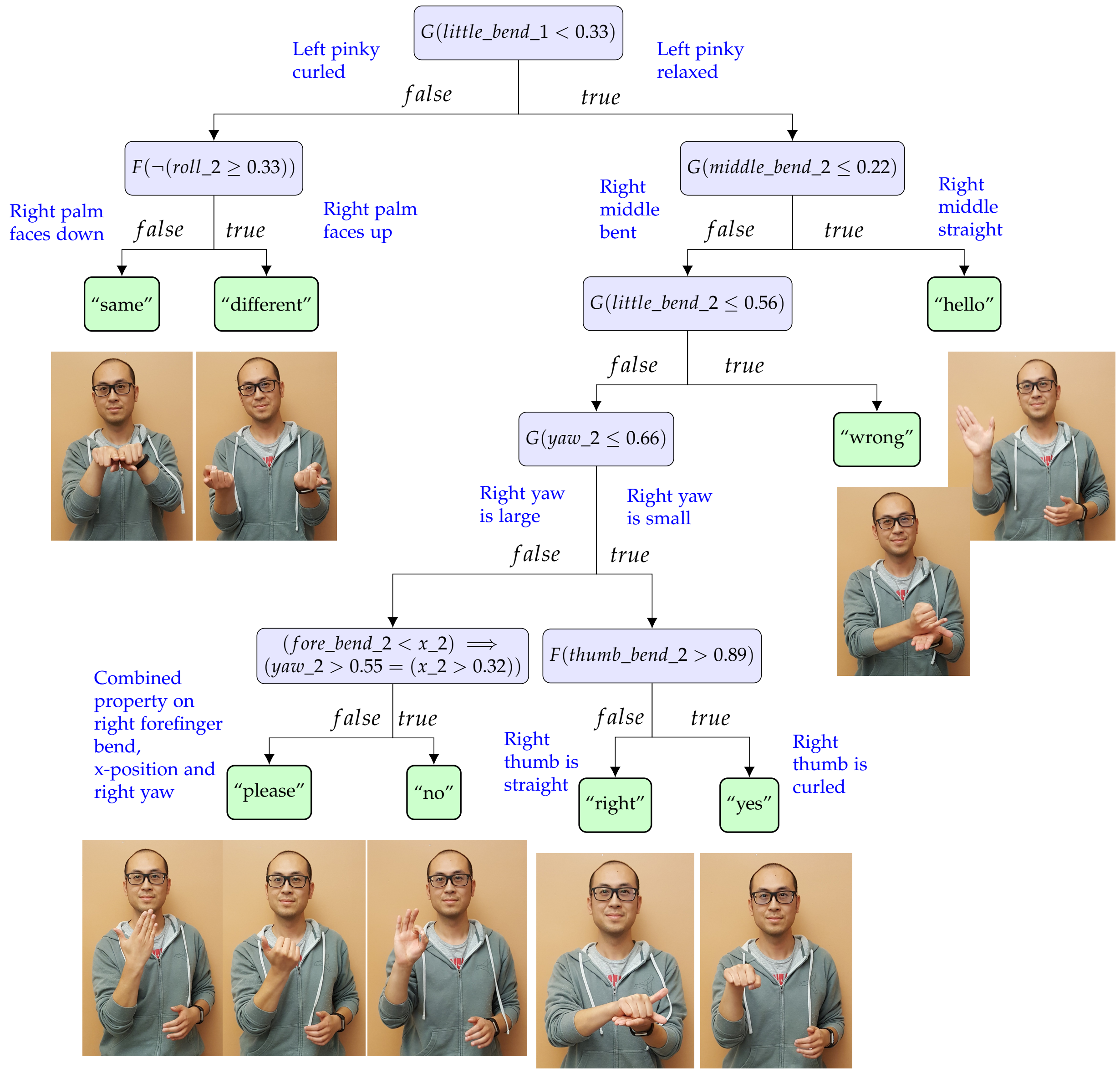


Figure 7.4: GBDT categorization of the Auslan dataset.

# Chapter 8

# Application: Aircraft Collision Avoidance

This chapter applies the algorithms of AdaStress—AST, DAST, and GBDT—on an aircraft collision avoidance system, where we find and analyze scenarios of near mid-air collisions. Section 8.1 introduces the Airborne Collision Avoidance System (ACAS X) and presents evidence from operational studies of its benefits over the existing system. Section 8.2 summarizes the basic operation of ACAS X. Section 8.3 describes experiments in applying AST to find NMACs in aircraft encounters. Then, Section 8.4 describes experiments in applying DAST to compare ACAS X and TCAS. Section 8.5 presents experiments in categorizing failure scenarios using GBDT, and Section 8.6 presents a performance evaluation of AST in finding NMACs. The final section, Section 8.7 summarizes the results and contributions of this chapter.

## 8.1 Introduction

Aircraft collision avoidance systems are mandated on all large transport and cargo aircraft in the United States and other countries around the world to help prevent mid-air collisions. Their operation has played a crucial role in the exceptional level of safety in the national airspace [86]. The Traffic Alert and Collision Avoidance System (TCAS) is currently deployed in the United States and many countries around the world. TCAS has been very successful at protecting aircraft from mid-air collisions. However, studies have revealed fundamental limitations in TCAS that prevent it from operating effectively in the next-generation airspace where the number of aircraft is expected to increase significantly [86]. To address the growing needs of the national airspace, the Federal Aviation Administration (FAA) has decided to create a new aircraft collision avoidance system. The next-generation Airborne Collision Avoidance System (ACAS X) was created promising a number of improvements over TCAS including a reduction in collision risk while simultaneously reducing the number of unnecessary alerts [87]. This thesis work was performed on prototypes of ACAS X while the system was still being developed and tested. On September $20^{th}$, 2018, the RTCA[1] accepted ACAS X to replace TCAS as the next standard for airborne collision avoidance and the system is expected to be widely deployed in the near future.

**Operational Studies.** ACAS X has been shown to be much more operationally suitable than TCAS [88]. Table 8.1 is a comparison of ACAS X and TCAS on several key operational metrics evaluated on a number of simulated aircraft encounter datasets. The data is excerpted from the results of many studies performed at MIT Lincoln Laboratory (MIT-LL) and presented at the RTCA [88]. The general trend in Table 8.1 is that ACAS X shows significant improvements in overall safety and alert rates compared to TCAS. The primary metric of safety is the probability of a near mid-air collision (NMAC), $P_{NMAC}$. An NMAC is defined as two aircraft coming closer than 500 feet horizontally and 100 feet vertically. On large encounter datasets, ACAS X is shown to reduce the probability of NMAC by 16.82% to 54.37% compared to TCAS. Alert metrics show significant improvements over TCAS as well, including the aggregate alert rate, which is the number of times the collision avoidance system alerts (counted by aircraft); and the 500 feet corrective alert rate, which is the number of alerts issued in encounters where the aircraft are flying level and vertically separated by exactly 500 feet. ACAS X also significantly improves metrics on generally undesirable scenarios such as the altitude crossing rate, which is the number of encounters where two aircraft cross in altitude; and the reversal rate, which is the number of times the collision avoidance system first advises the pilot to maneuver in one direction, then later advises the pilot to maneuver in the opposite direction.

Table 8.1: Comparison of ACAS X and TCAS operational metrics on various encounter datasets

| Metric | Dataset | Number of encounters | TCAS v7.1 | ACAS Xa 0.10.3 | Improvement over TCAS |
|---|---|---|---|---|---|
| Safety ($P_{NMAC}$) | LLCEM | 5,956,128 | $2.179 \cdot 10^{-4}$ | $1.744 \cdot 10^{-4}$ | 19.57% |
| Safety ($P_{NMAC}$) | SAVAL | 75,173,906 | $4.361 \cdot 10^{-4}$ | $3.627 \cdot 10^{-4}$ | 16.82% |
| Safety ($P_{NMAC}$) | SA01 | 100,000 | $4.106 \cdot 10^{-2}$ | $1.873 \cdot 10^{-2}$ | 54.37% |
| Alert Rate (by aircraft) | TRAMS | 293,101 | 252,656 | 121,267 | 52.00% |
| 500′ Corrective Alert Rate | TRAMS | 175,184 | 14,912 | 9,919 | 33.48% |
| Altitude Crossing Rate | TRAMS | 293,101 | 3,196 | 1,582 | 50.5% |
| Reversal Rate | TRAMS | 293,101 | 1,029 | 556 | 45.97% |

**Near Mid-Air Collisions.** Studies have shown that the risk of NMAC is extremely small and moreover ACAS X reduces the risk even further over TCAS overall. However, the risk of NMAC cannot be completely eliminated due to factors such as surveillance noise, pilot response delay, and the need for an acceptable alert rate [87]. Because NMACs are such important safety events, it is important to study and understand the rare circumstances under which they can still occur even if they are extremely unlikely. Understanding the nature of the residual NMAC risk is important for certification and to inform

[1]RTCA, Inc. was formerly known as the Radio Technical Commission for Aeronautics, but is now known simply as the RTCA.

the development of future iterations of the system. This paper uses AdaStress to find and analyze the rare corner cases where an NMAC can still occur.

## 8.2 ACAS X Operation Overview

There are several versions of ACAS X under development. For example, ACAS Xu is a version of ACAS X for unmanned aircraft. This paper considers a development (and not final) version of ACAS Xa, which uses active surveillance and is designed to be a direct replacement to TCAS. ACAS Xa and TCAS are used on commercial passenger and cargo aircraft. Despite the internal logics of ACAS X and TCAS being derived completely differently, the input and output interfaces of these two systems are identical. As a result, the following description of aircraft collision avoidance systems applies to both ACAS X and TCAS.

Airborne collision avoidance systems monitor the airspace around an aircraft and issue alerts to the pilot if a conflict with another aircraft is detected. These alerts, called resolution advisories (RAs), instruct the pilot to maneuver the aircraft to a certain target vertical velocity and maintain it. The advisories are typically issued when the aircraft are within approximately 20–40 seconds to a potential collision. Table 8.2 lists the primary RAs. We use $\dot{z}_{own}$ to denote the current vertical velocity of own aircraft.

Table 8.2: Primary ACAS X advisories

| Abbreviation | Description | Rate to Maintain (ft/min) |
|---|---|---|
| COC | Clear of conflict | N/A |
| DND | Do not descend | 0 |
| DNC | Do not climb | 0 |
| DND$x$ | Do not descend at greater than $x$ ft/min | $max(-x, \dot{z}_{own})$ |
| DNC$x$ | Do not climb at greater than $x$ ft/min | $min(x, \dot{z}_{own})$ |
| MAINTAIN | Maintain current rate | $\dot{z}_{own}$ |
| DS1500 | Descend at 1,500 ft/min | $-1500$ |
| CL1500 | Climb at 1,500 ft/min | $+1500$ |
| DS2500 | Descend at 2,500 ft/min | $-2500$ |
| CL2500 | Climb at 2,500 ft/min | $+2500$ |

The COC advisory stands for "clear of conflict" and is equivalent to no advisory. The pilot is free to choose how to control the aircraft. The DND and DNC advisories stand for "do not descend" and "do not climb", respectively. They restrict the pilot from flying in a certain direction. The DND$x$ and DNC$x$ advisories extend DND and DNC, respectively, with a maximum vertical rate $x$. They respectively restrict the pilot from descending or climbing at a vertical rate greater than $x$ feet per minute. The MAINTAIN advisory is preventative and instructs the pilot to maintain the current vertical rate of the aircraft. The advisories DS1500 and CL1500 instruct the pilot to descend or climb at 1,500 feet per minute. The pilot is expected to maneuver the aircraft at $\frac{1}{4}g$ acceleration until the target vertical rate is reached, and then maintain that vertical rate. The DS2500 and CL2500 advisories instruct the pilot to descend or climb at an increased rate of 2,500 feet per minute. The DS2500 and CL2500 advisories are strengthened advisories and a stronger response from the pilot is expected. For these strengthened RAs, the pilot is expected to maneuver at $\frac{1}{3}g$ acceleration until the target vertical rate is reached then maintain that vertical rate. Strengthened RAs must follow a weaker RA of the same vertical direction. They cannot be issued directly. For example, a CL1500 advisory must precede a CL2500 advisory.

Advisories issued by collision avoidance systems on different aircraft are not independent. When an RA is issued, a coordination message is broadcasted to other nearby aircraft to prevent other collision avoidance systems from accidentally issuing an RA in the same vertical direction.

## 8.3 Stress Testing using AST

This section describes experiments applying AST to find the most likely NMAC scenarios in aircraft mid-air encounters. We first present the general setup of the experiments and then we present the studies in stress testing pairwise (two-aircraft) encounters and multi-threat (three-aircraft) encounters.

### 8.3.1 Experimental Setup

This section presents the details of the simulation model and the reward function used in AST for stress testing. These are used throughout the experiments performed in this chapter with minor modifications depending on the study. Deviations will be made clear in the respective discussions of the studies.

#### Simulation Model

We construct a seed-action simulator (Section 5.4.1) modeling the evolution of an aircraft mid-air encounter in discrete time. The simulator is Markovian by construction but the system state is not used explicitly by the optimizer. Rather, the optimizer, following the seed-action framework in Section 5.4.1, relies on controlling the system evolution deterministically via control of the pseudorandom seed. The overall architecture of the simulator for two aircraft is shown in Figure 8.1. The simulation models capture the key aspects of the encounter, including the initial state, sensors, collision avoidance system, pilot response, and aircraft dynamics. Each model has its own set of state variables that can be continuous or finite valued. The system state is the concatenation of the states in all the models, and thus the simulation state is in general multivariate and heterogeneous. Due to our seed-action construction, the system state is never explicitly constructed. We now discuss these the models in more detail.

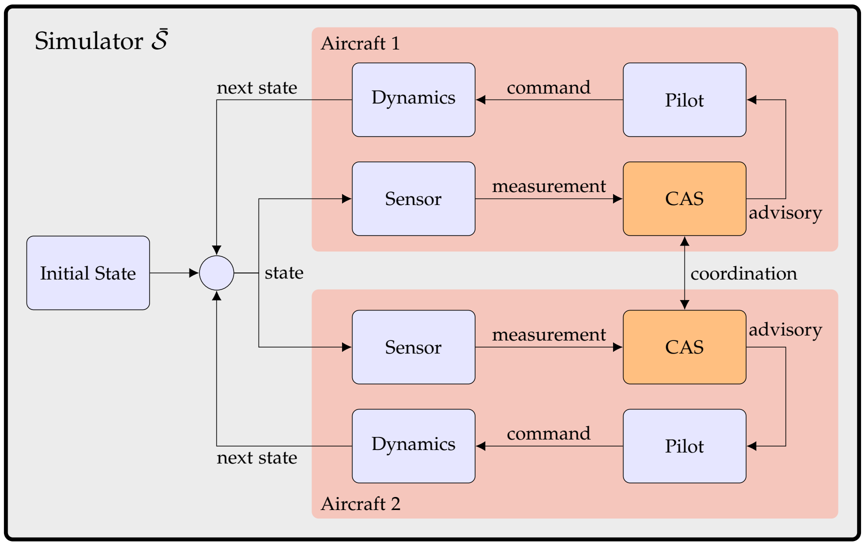


Figure 8.1: System diagram of seed-action simulator for pairwise encounters.

**Initial State.** The initial state of an encounter includes initial positions, velocities, and headings of the aircraft. The initial state is drawn from a distribution that gives realistic initial configurations of aircraft that are likely to lead to NMAC. Once the initial state is sampled, it is fixed for the duration of the search. In our experiments, pairwise (two-aircraft) encounters are initialized using the Lincoln Laboratory Correlated Aircraft Encounter Model (LLCEM) [89, 90]. LLCEM is a statistical model learned from a large body of radar data of the entire national airspace. We follow the encounter generation procedure described in the paper [89]. Multi-threat (three-aircraft) encounters use the *Star model*, which initializes aircraft on a circle heading towards the origin spaced apart at equal angles. Initial airspeed, altitude, and vertical rate are sampled from a uniform distribution over a prespecified range. The horizontal distance from the origin is set such that without intervention, the aircraft intersect at approximately 40 seconds into the encounter.

**Sensor Model.** The sensor model captures how the collision avoidance system perceives the world. We assume active, beacon-based radar capability with no noise. For own aircraft, the sensor measures the vertical rate, barometric altitude, heading, and height above ground. For each intruding aircraft, the sensor measures slant range (relative distance to intruder), bearing (relative horizontal angle to intruder), and relative altitude.

**Collision Avoidance Systems.** The collision avoidance system is the system under test. We use a prototype of ACAS X in the form of a binary library obtained from the FAA. The binary has a minimal interface that allows initializing and stepping the state forward in time given inputs at each step. The system maintains internal state, but does not expose it. However, ACAS X and the library are deterministic given its inputs. The primary output of the ACAS X system is the RA. ACAS X has a coordination mechanism to ensure that issued RAs from different aircraft are compatible with one another, i.e., that two aircraft are not instructed to maneuver in the same vertical direction. The messages are communicated to all nearby aircraft through coordination messages.

Our differential studies compare ACAS X with TCAS. Our implementation of TCAS is also a binary library obtained from the FAA. Both binaries have identical input and output interfaces making them interchangeable in the simulator.

**Pilot Model.** The pilot model consists of a model for the pilot's intent and a model for how the pilot responds to an RA. The pilot's intent is how the pilot would fly the aircraft if there are no RAs. To model intended commands, we use the pilot command model of LLCEM, which gives a realistic stochastic model of aircraft commands in the airspace [89]. The pilot response model defines how pilots respond to an issued RA. The pilots in our model are assumed to respond deterministically to an RA with perfect compliance after a fixed delay [91]. Pilots respond to initial RAs with a five-second delay and subsequent RAs (i.e., strengthenings and reversals) with a three-second delay. During the initial delay period, the pilot continues to fly the intended trajectory. During response delays from subsequent RAs, the pilot continues responding to the previous RA. Multiple RAs issued successively are queued so that both their order and timing are preserved. In the case where a subsequent RA is issued within 2 seconds or less of an initial RA, the timing of the subsequent RA is used and the initial RA is skipped. The pilot command includes commanded airspeed acceleration, commanded vertical rate, and commanded turn rate.

**Aircraft Dynamics Model.** The aircraft dynamics model determines how the state of the aircraft propagates with time. The aircraft state includes the airspeed, position north, position east, altitude, roll angle, pitch angle, and heading angle. The aircraft state is propagated forward at 1 Hertz using forward Euler integration.

In our experiments, the commands of the pilots when not responding to an advisory are stochastic and are being optimized by the algorithm. When no RA is active, the pilot commands follow the stochastic dynamic model of LLCEM. When an RA is active, the vertical component of the pilot's command follows the pilot response model, while the other components follow LLCEM. Other simulation components are deterministic in our experiments. However, in the future, we may consider stochastic models for sensors, aircraft dynamics, and pilot response.

### Reward Function

We use the reward function defined in Equation 5.2 for optimization. The event space $E$ is defined to be an NMAC, which occurs when two aircraft are closer than 100 feet vertically and 500 feet horizontally. We define the miss distance $d$ to be the distance of closest approach, which is the Euclidean distance of the aircraft at their closest point in the encounter. The distance of closest approach is a good metric because it is monotonically decreasing as trajectories get closer and reach a minimum at an NMAC. Because the models in the simulator use continuous distributions, we use the transition probability density in the reward function.

### 8.3.2 Stress Testing Single-Threat Encounters

We apply AST to analyze NMACs in encounters involving two aircraft. We search 100 encounters initialized using samples from LLCEM. The configuration is shown in Table 8.3. Of the 100 encounters searched, 18 encounters contained an NMAC, yielding an empirical find rate of 18%. When the optimization algorithm completes, it returns the path with the highest reward regardless of whether the path contains an NMAC or not. When the returned path does not contain an NMAC, it is uncertain whether an NMAC exists and the algorithm was unable to find it, or whether no NMAC is reachable given the initial state of the encounter. We manually cluster the NMAC encounters and present our findings.

Table 8.3: Single-threat encounter configuration for AST

| Simulation | |
|---|---|
| Number of aircraft | 2 |
| Initialization | LLCEM |
| Sensors | Active, beacon-based, noiseless |
| Collision avoidance system | ACAS Xa Run 13 libcas 0.8.5 |
| Pilot response model | Deterministic 5s–3s |
| **MCTS** | |
| Maximum steps | 50 |
| Iterations | 2000 |
| Exploration constant | 100.0 |
| $k$ | 0.5 |
| $\alpha$ | 0.85 |

NMAC encounter plots contain horizontal tracks (top-down view) and vertical profile (altitude versus time). Aircraft numbers are indicated at the start and end of each trajectory. RA codes are labeled at the time of occurrence and preceded by the aircraft number. Marker colors indicate the RA issued: blue for COC, orange for CL1500, red for CL2500, cyan for DS1500, purple for DS2500, and grey for DNC/DND/MAINTAIN/MULTITHREAT. Symbols inside the markers indicate the state of the pilot response: no symbol for not responding, dash for responding to previous RA, and asterisk for responding to current RA.

**Crossing Time.** We observed a number of NMACs resulting from well-timed vertical maneuvers. In particular, several encounters included aircraft crossing in altitude during the pilot response delay to an initial RA. Figure 8.2 shows one such encounter that eventually ends in an NMAC at 36 seconds into the encounter. The probability density of the encounter evaluated under LLCEM is $5.3 \cdot 10^{-18}$. This measure can be used as an unnormalized measure of likelihood of occurrence. In this encounter, the aircraft cross in altitude during pilot 1's response delay. The crossing leads to aircraft 1 starting the climb from below aircraft 2. The subsequent reversal later in the encounter is unable to resolve the conflict due to the pilot response delay.

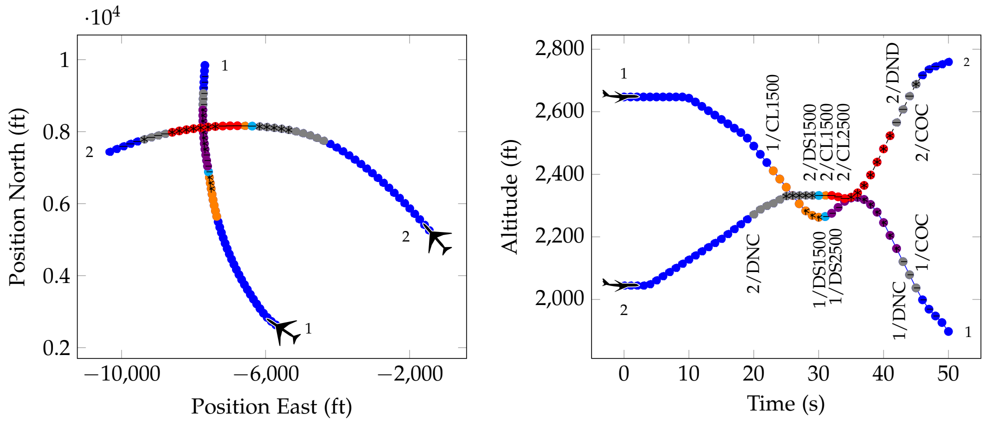


Figure 8.2: NMAC encounter where the aircraft cross in altitude during pilot delay at 26 seconds and an NMAC occurs at 36 seconds.

**High Turn and Vertical Rates.** Turns at high rates quickly shorten the time to closest approach. ACAS X does not have full state information about its intruder and must estimate it by tracking relative distance, relative angle, and the intruder altitude. Figure 8.3 shows an example of an encounter that has similar crossing behavior as Figure 8.2 but exacerbated by the high turn rate of aircraft 2 (approximately 1.5 times the standard turn rate). In this scenario, the aircraft become almost head-on at the time of closest approach and a reversal is not attempted. An NMAC with a probability density of $6.5 \cdot 10^{-17}$ occurs at 48 seconds into the encounter. Aircraft 1 is also coincidently descending at a high vertical rate. The combination makes this encounter operationally very unlikely.

**Maneuvering Against RA.** Our analysis revealed a number of NMAC encounters where the pilot initially maneuvers against the issued RA before complying. That is, after the pilot receives the RA, they maneuver the aircraft in the opposite direction of what is instructed by the RA for the duration of the pilot response delay before subsequently complying and reversing direction. Pilots do not normally maneuver in this manner and so this scenario represents a very operationally rare case. Even so, ACAS X does seem to be able to resolve the majority of these initially disobeying cases. In most cases, the maneuvering must

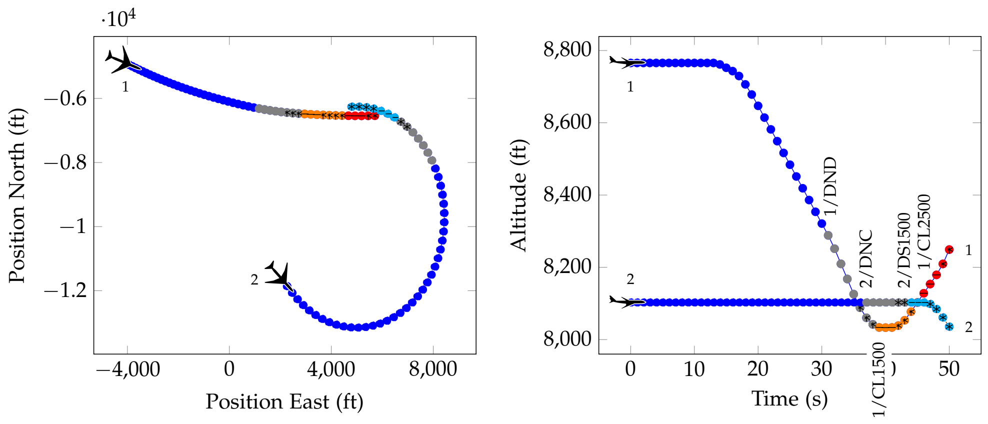


Figure 8.3: NMAC encounter where aircraft 2 is simultaneously turning and descending at high rate and an NMAC occurs at 48 seconds.

be very aggressive against the RA to result in an NMAC.

**Sudden Aggressive Maneuvering.** Sudden maneuvers can lead to NMACs if they are sufficiently aggressive. In particular, we observed some NMAC encounters where two aircraft are approaching one another separated in altitude and flying level, then one aircraft suddenly accelerates vertically towards the other aircraft as they are about to pass. Under these circumstances, given the pilot response delays and dynamic limits of the aircraft, there is insufficient time and distance remaining for the collision avoidance system to resolve the conflict. Pilots do not normally fly so aggressively in operation, so this case is extremely unlikely. In fact, they are even more rare than our model predicts. ACAS X issues traffic alerts (TAs) to alert pilots to nearby traffic, so that pilots are made aware of intruding aircraft well before the initial RA. These advance warnings increase the pilot's situational awareness and reduce blunders like these. Our simulator does not model the effect of such TAs, however. As a designer of ACAS X, one course of action would be to tune ACAS X to intervene preemptively in such scenarios. While this reduces the risk of possible sudden behavior, it also increases the alert rate of the system. Ultimately, the ACAS X designer must assess the probabilities of various scenarios and find the delicate balance between risk of collision and issuing too many advisories. Since these scenarios are extremely rare, we must trade off accordingly.

**Combined Factors.** In our experiments, it is rare for an NMAC to be attributable to a single cause. More commonly, a combination of factors contribute to the NMAC. Figure 8.2 shows an example of an encounter where multiple factors contribute to the NMAC. In Figure 8.2, crossing time played a crucial role in the NMAC. However, there are a number of other factors that are important as well. The horizontal behavior where they are turning into each other is significant as it reduces the time to closest approach. The two vertical maneuvers of aircraft 1 before receiving an RA are also important. Similar observations can be made for many of the other NMAC encounters found.

### 8.3.3 Stress Testing Multi-Threat Encounters

We apply AST to analyze NMACs in three-aircraft encounters. We search 100 encounters initialized using samples from the Star model. The configuration is shown in Table 8.4. We found 25 NMACs out of 100 encounters searched, yielding a find rate of 25%.

Table 8.4: Multi-threat encounter configuration for AST

| Simulation | |
|---|---|
| Number of aircraft | 3 |
| Initialization | Star model |
| Sensors | Active, beacon-based, noiseless |
| Collision avoidance system | ACAS Xa Run 13 libcas 0.8.5 |
| Pilot response model | Deterministic 5s–3s |
| **MCTS** | |
| Maximum steps | 50 |
| Iterations | 1000 |
| Exploration constant | 100.0 |
| $k$ | 0.5 |
| $\alpha$ | 0.85 |

**Pairwise Coordination in Multi-Threat.** Our algorithm discovered a number of NMAC encounters where all aircraft are issued a MULTITHREAT (MTE) RA and asked to follow an identical climb rate. Complying with the RA results in the aircraft closing horizontally without gaining vertical separation. Figure 8.4 shows an example of such an encounter where an NMAC occurs with probability density $5.8 \cdot 10^{-7}$ at 38 seconds into the encounter. In discussing these results with the ACAS X development team, we learned that this behavior is a known issue that can arise when performing multi-aircraft coordination using a pairwise coordination mechanism. The pairwise coordination messages in essence determine which aircraft will climb and which will descend in an encounter. Since coordination messaging occurs pairwise, under rare circumstances it is possible for each aircraft to receive conflicting coordination messages from the other aircraft in the scenario. In nominal encounters, the aircraft that receives conflicting coordination messages from two aircraft remains level and lets the other aircraft climb or descend around it. However, in these encounters, all three aircraft receive conflicting coordination messages. Although very rare, this is an important case that is being addressed by both TCAS and ACAS X development teams.

**Limited Maneuverable Space.** In general, multi-threat encounters are more challenging to resolve than pairwise encounters because there is less open space for the aircraft to maneuver. Figure 8.5 shows

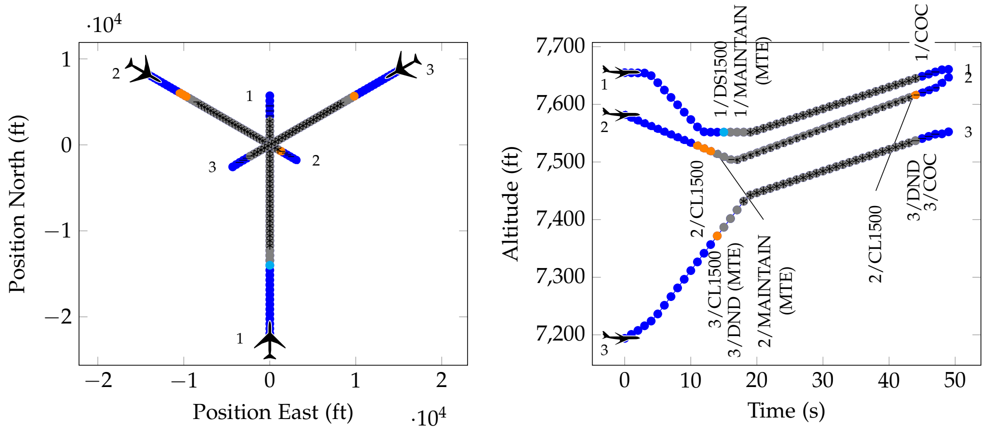


Figure 8.4: NMAC encounter where all aircraft receive conflicting coordination messages and an NMAC occurs at 38 seconds.

an example of an NMAC encounter where aircraft 1 (the aircraft in the middle altitude between 10 and 36 seconds) needs to simultaneously avoid an aircraft below and a vertically closing aircraft from above. An NMAC with a probability density of $1.0 \cdot 10^{-16}$ occurs at 39 seconds into the encounter. Aircraft 2's downward maneuver greatly reduces the maneuverable airspace of aircraft 1. These encounters are undoubtedly extremely challenging for a collision avoidance system and it is unclear whether any satisfactory resolution exists. Nevertheless, we gain insight by observing how the collision avoidance system behaves under such circumstances.

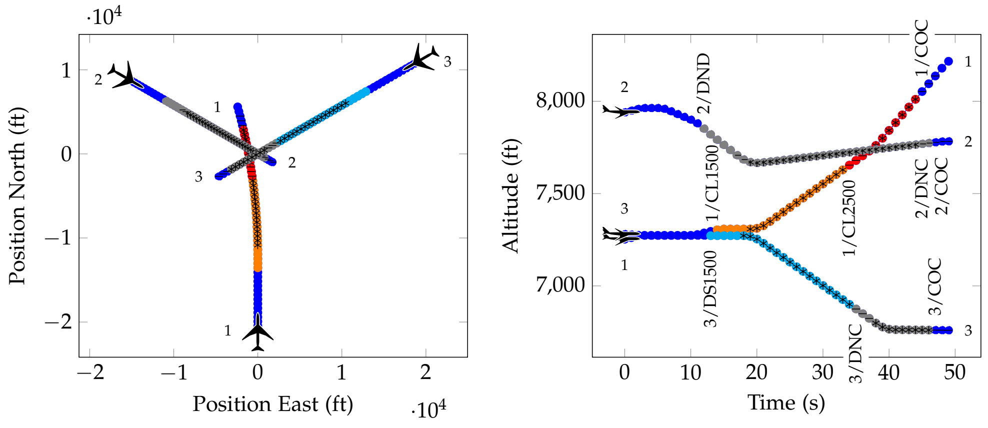


Figure 8.5: NMAC encounter where the aircraft have limited maneuverable airspace. Aircraft 1 must simultaneously avoid collision with aircraft 3 from below and aircraft 2 from above. NMAC occurs at 39 seconds.

**Pairwise Phenomena.** Phenomena that appear in pairwise encounters also appear in multi-threat encounters. The presence of the third aircraft typically exacerbates the encounter. In our multi-threat analysis, we noted similar phenomena related to crossing time, maneuvering against RA, and sudden aggressive maneuvering as discussed previously. We did not observe any cases related to high turn rates in the multi-threat setting due to our use of the Star model.

## 8.4 Differential Stress Testing using DAST

We present two studies applying DAST to analyze aircraft collision avoidance systems. The first study (Section 8.4.1) finds the most likely failure scenarios in ACAS X relative to a TCAS baseline, and the second study (Section 8.4.2) finds the most likely failure scenarios in TCAS when compared to ACAS X. The following sections present figures containing multiple panels. For ease of discourse, we use the following abbreviations when referring to a specific panel in a figure: top row (TR), bottom row (BR), far left (FL), center left (CL), center right (CR), and far right (FR).

### 8.4.1 Stress Testing ACAS X Relative to TCAS Baseline

We apply DAST to perform differential stress testing of ACAS X relative to a TCAS baseline. We seek to find NMAC encounters that occur in ACAS X but not in TCAS. The search is extremely difficult. Not only are both ACAS X and TCAS systems extremely safe, which means that NMACs are extremely rare, but also ACAS X is a much safer system than TCAS overall, which makes cases where ACAS X has NMAC but TCAS does not extremely rare. We seek to find those extremely rare corner cases.

We searched 2700 pairwise encounters initialized with samples from LLCEM. The configuration is shown in Table 8.5. The top 10 highest reward paths were returned for each encounter initialization producing a total of 27,000 paths. Of these paths, a total of 28 contained NMACs, which originated from 10 encounter initializations. We analyzed the scenarios and confirmed that all were operationally rare scenarios. We present examples of NMAC found by the algorithm and discuss their properties.

Table 8.5: Differential stress testing configuration comparing ACAS X and TCAS

| Simulation | |
|---|---|
| Number of simulators | 2 |
| Number of aircraft per simulator | 2 |
| Initialization | LLCEM |
| Sensors | Active, beacon-based, noiseless |
| Collision avoidance system (test) | ACAS Xa Run 15 libcas 0.10.3 |
| Collision avoidance system (baseline) | TCAS II v7.1 |
| Pilot response model | Deterministic 5s–3s |
| **MCTS** | |
| Maximum steps | 50 |
| Iterations | 3000 |
| Exploration constant | 100.0 |
| $k$ | 0.5 |
| $\alpha$ | 0.85 |

**ACAS X Issues RA, but TCAS Does Not.** We found some NMAC cases where ACAS X issued an RA but TCAS did not. An example is shown in Figure 8.6 where an NMAC occurs at 40 seconds in the ACAS X simulation but no NMAC occurs in the TCAS simulation. The encounter occurs with a probability density of $1.7 \cdot 10^{-19}$. No RA was issued in the TCAS simulation. In the ACAS X simulation (Figure 8.6(TR)), both aircraft are traveling at a high absolute vertical rate exceeding 70 feet per second toward each other. Both aircraft receive an RA to level-off but the aircraft cannot respond in time due to the high vertical rates and the pilot response delay. The aircraft proceed to cross in altitude. After crossing, ACAS X increases the strength of the advisory in the same direction as the previous RA. Since the aircraft have crossed in altitude, responding to the RA results in a loss of vertical separation and an NMAC.

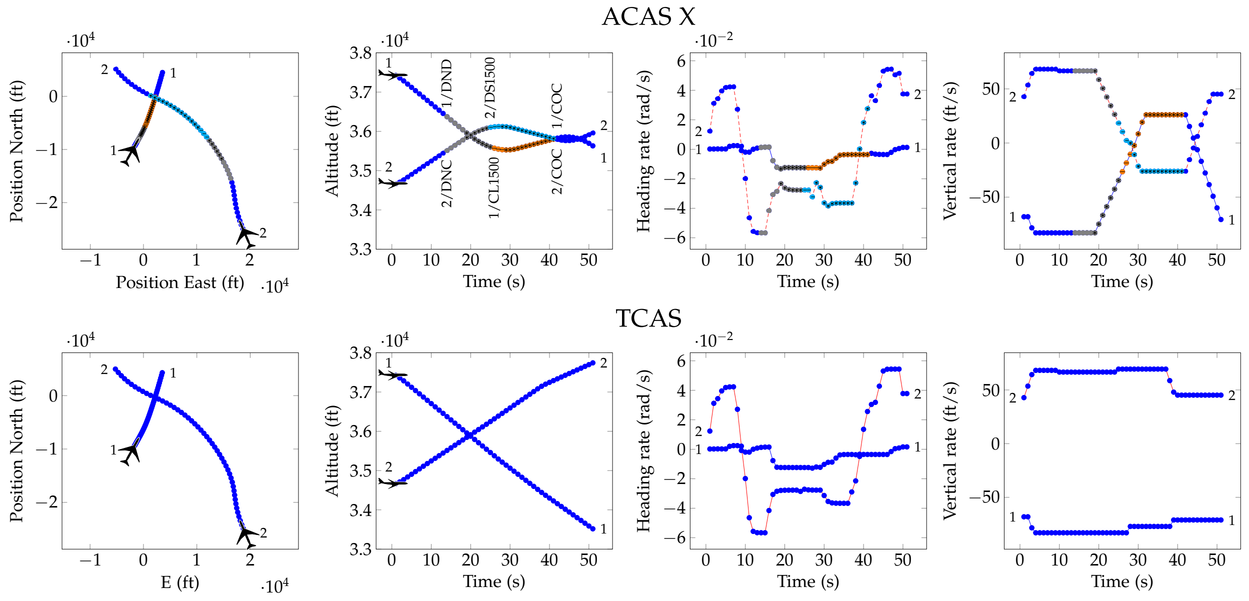


Figure 8.6: NMAC encounter where ACAS X issues an RA but TCAS does not. The aircraft approach at high relative vertical velocity. The aircraft cross in altitude at 20 seconds and NMAC occurs at 40 seconds in ACAS X simulation.

High vertical rates are known to make conflict resolution more difficult, especially for vertical-only collision avoidance systems like TCAS and this version of ACAS X. Aircraft with high vertical rates take longer to reverse direction vertically and more vertical distance is traveled during the pilot's response delay. As a result, advisories take longer to take full effect. Moreover, in cases where both aircraft are maneuvering in the same vertical direction, an aircraft may lose the ability to "outrun" the other aircraft. For example, even if a maximal climb advisory is issued, it may not be sufficient for the aircraft to stay above a second aircraft climbing at an even higher rate. Another interesting feature of this encounter is the horizontal behavior (Figure 8.6(FL)). Aircraft 2 is initially turning away from the other aircraft before turning towards it at 8 seconds into the encounter. The initial RA is issued shortly after that maneuver. Large rapid changes in turn rate around the time of an RA can make it difficult for a collision avoidance system to accurately estimate the time to horizontal intersection.

Overall, ACAS X is much safer and more operationally suitable than TCAS. However, there are some trade-offs between the two systems as highlighted by our methods in Figure 8.6. ACAS X's late altering characteristic, which reduces the number of unnecessary alerts, can sometimes hurt encounters with higher vertical rates, such as seen in this example. In deciding the trade-off, a designer must weigh the relative likelihood of these encounters versus the effect on other more frequently observed trajectories.

**Simultaneous Horizontal and Vertical Maneuvering.** Many of the NMAC encounters found involve an aircraft turning while simultaneously climbing or descending very rapidly. Figure 8.7 shows an example where an NMAC occurs at 45 seconds in the ACAS X simulation but no NMAC occurs in the TCAS simulation. The encounter occurs with probability density $2.7 \cdot 10^{-4}$. In this example, aircraft 1 is flying generally straight and level while aircraft 2 is simultaneously turning and climbing at a vertical rate exceeding 80 feet per second. Aircraft 2 receives an RA to level-off (Figure 8.7(TR)(CL)), but is unable to maneuver in time before losing vertical separation resulting in an NMAC. In this example, TCAS is able to resolve the conflict by issuing RAs to both aircraft earlier in the encounter than ACAS X (Figure 8.7(BR)(CL)).

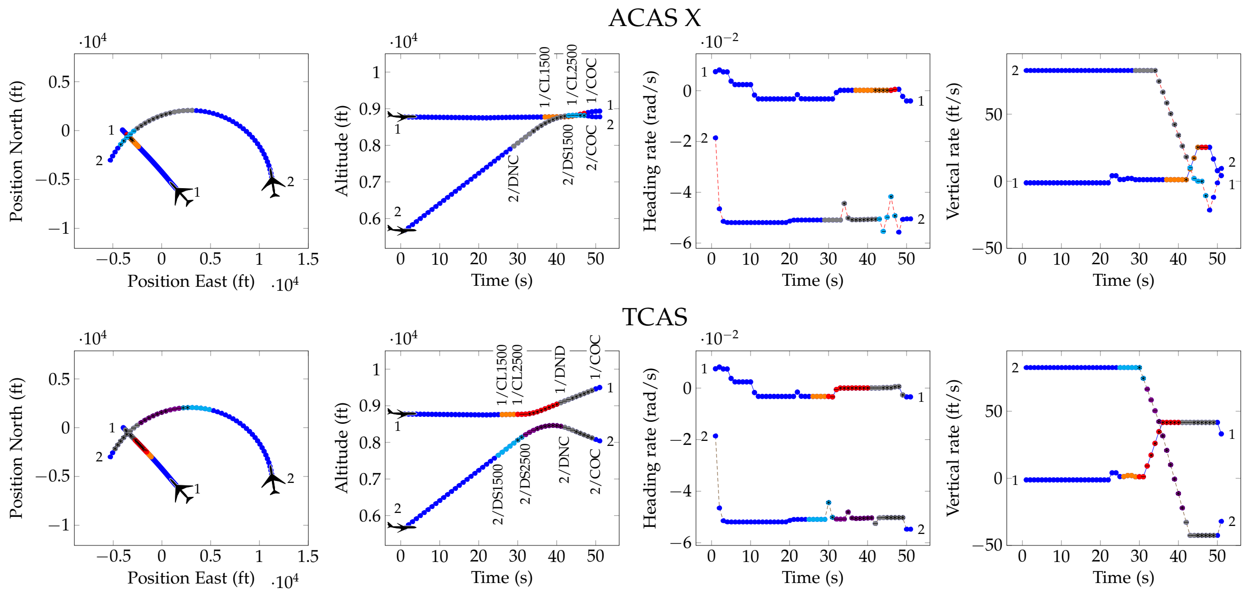


Figure 8.7: NMAC encounter where aircraft 2 turns and climbs very rapidly. NMAC occurs at 45 seconds in the ACAS X simulation. TCAS resolves the conflict by alerting earlier in the encounter.

Horizontally, aircraft 2's turn quickly shortens the time to closest approach between the two aircraft. Vertically, aircraft 2 receives a level-off advisory but the high climb rate and pilot response delay limit how quickly the aircraft can be brought to compliance. Scenarios that involve turning and simultaneously climbing or descending at high rate are operationally very rare.

**Horizontal Maneuvering.** In some very rare cases, NMACs can also result from horizontal maneuvering alone. An example is shown in Figure 8.8, where an NMAC occurs at 40 seconds in the ACAS X simulation but no NMAC occurs in the TCAS simulation. The encounter occurs with probability density $4.3 \cdot 10^{-11}$. The aircraft are initially headed away from each other but they are also turning towards each other. At 25 seconds into the encounter, the aircraft turn more tightly towards each other, rapidly reducing the time to closest approach. Crossing advisories are issued to both aircraft 9 seconds prior to NMAC (Figure 8.8(TR)(CL)). However, there is not enough time remaining to cross safely and an NMAC occurs. In this example, TCAS is able to resolve the conflict by issuing RAs to both aircraft earlier in the encounter (Figure 8.8(BR)(CL)). However, it is unclear how the aircraft got to their initial positions in the encounter and whether this initial position is generally reachable.

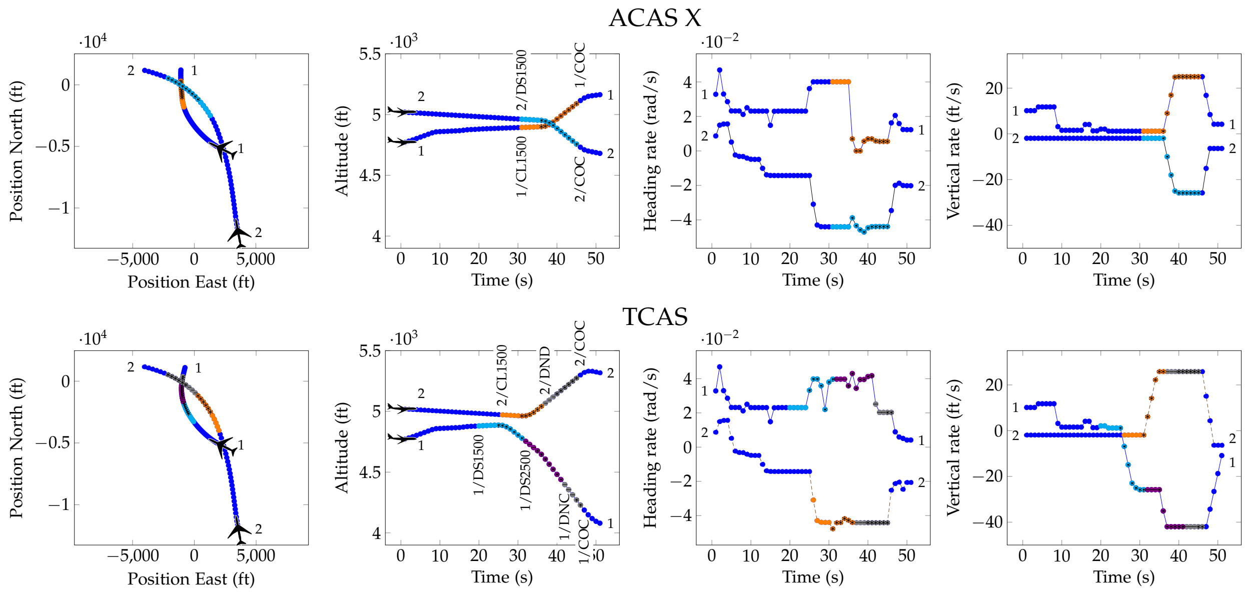


Figure 8.8: NMAC encounter where the aircraft maneuver horizontally at 25 seconds. NMAC occurs at 40 seconds in the ACAS X simulation. TCAS resolves the conflict by alerting earlier in the encounter.

### 8.4.2 Stress Testing TCAS Relative to ACAS X Baseline

For comparison, we perform DAST experiments where we consider TCAS as the system under test and ACAS X as the baseline, which is the inverse of the experiment in Section 8.4.1. We searched 2700 pairwise encounters with samples from LLCEM. We use the same configuration as in Table 8.5 with the exception that the test and baseline systems are reversed. The top 10 highest reward paths are returned for each encounter initialization, producing a total of 27,000 paths as in Section 8.4.1. Of these paths, a total of 39 contained NMACs, which represents an increase of 39.3% compared to Section 8.4.1. It is reassuring that this result, where DAST found 39.3% more NMACs with TCAS compared to ACAS X, is qualitatively similar to the results from other studies which use direct Monte Carlo sampling and also show a safety benefit of ACAS X compared to TCAS [23]. The NMACs also originate from a broader set of encounter initializations as well (22 here versus 10 in Section 8.4.1). We now present examples of NMAC found by the algorithm and discuss their properties.

**Crossing RA Followed by a Reversal.** An example is shown in Figure 8.9 where an NMAC occurs at 41 seconds in the TCAS simulation but no NMAC occurs in the ACAS X simulation. The encounter occurs with a probability density of $6.6 \cdot 10^{-12}$. In the encounter, the aircraft converge vertically while they approach in a perpendicular configuration horizontally (Figure 8.9(FL–CL)). TCAS issues crossing initial advisories to the aircraft, but then reverses the RAs shortly before the aircraft cross in altitude (Figure 8.9(TR)(CL)). The reversal leads to the aircraft reversing vertical direction after they have crossed in altitude, reducing their vertical separation, and eventually resulting in an NMAC. In contrast, ACAS X issues preventative advisories to the aircraft early in the encounter and keeps the aircraft vertically separated throughout the encounter (Figure 8.9(BR)(CL)).

The crossing TCAS advisory combined with a reversal shortly afterwards suggests that the encounter may be operating near a decision boundary and TCAS may be having trouble deciding whether the aircraft should cross in altitude. In this case, it appears the initial crossing RAs CL1500 and DS1500 may have successfully resolved the conflict had it been maintained, and it is the reversal that complicated the resolution. The lateness of the TCAS initial RA likely played a major role in the NMAC, leaving a very difficult decision for the direction selection of the RA. ACAS X gives a desirable outcome in this case by deciding early in the encounter that the aircraft should not cross in altitude by issuing preventative RAs DNC and DND.

**Double Horizontal Crossing.** An example is shown in Figure 8.10 where an NMAC occurs at 50 seconds in the TCAS simulation, but no NMAC occurs in the ACAS X simulation. The encounter occurs with a probability density of $2.2 \cdot 10^{-14}$. In the encounter, the aircraft turn as they converge both horizontally and vertically, eventually resulting in a horizontal double crossing (Figure 8.10(FL)). TCAS issues MAINTAIN advisories to the aircraft predicting that they will cross safely, then further clears the advisory after the first horizontal crossing. However, after the first crossing, the aircraft continue to turn toward each other into a second horizontal crossing. TCAS issues a second sequence of advisories, but it is too late and an NMAC occurs. Sequences of RAs separated by COC are called *split advisories* and are undesirable because they may cause confusion in the pilots. Another contributor to the difficulty of the encounter for TCAS is aircraft 2 increasing its turn rate mid-encounter, which significantly reduces time to collision (see

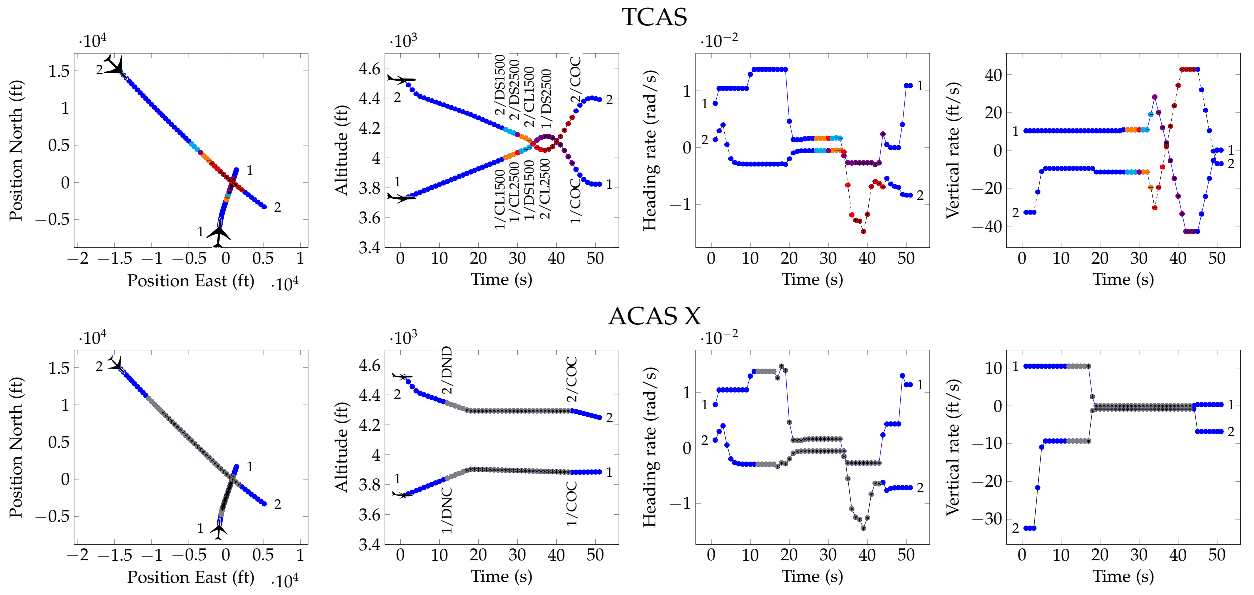


Figure 8.9: NMAC encounter where initial crossing RAs are reversed prior to the aircraft crossing in altitude. NMAC occurs at 41 seconds in the TCAS simulation. ACAS X resolves the conflict by alerting much earlier in the encounter with preventative RAs.

Figure 8.10(TR)(CR)). The ACAS X system handles the encounter much more desirably, choosing a DND advisory to maintain vertical separation as the aircraft cross horizontally (Figure 8.10(BR)(CL)).

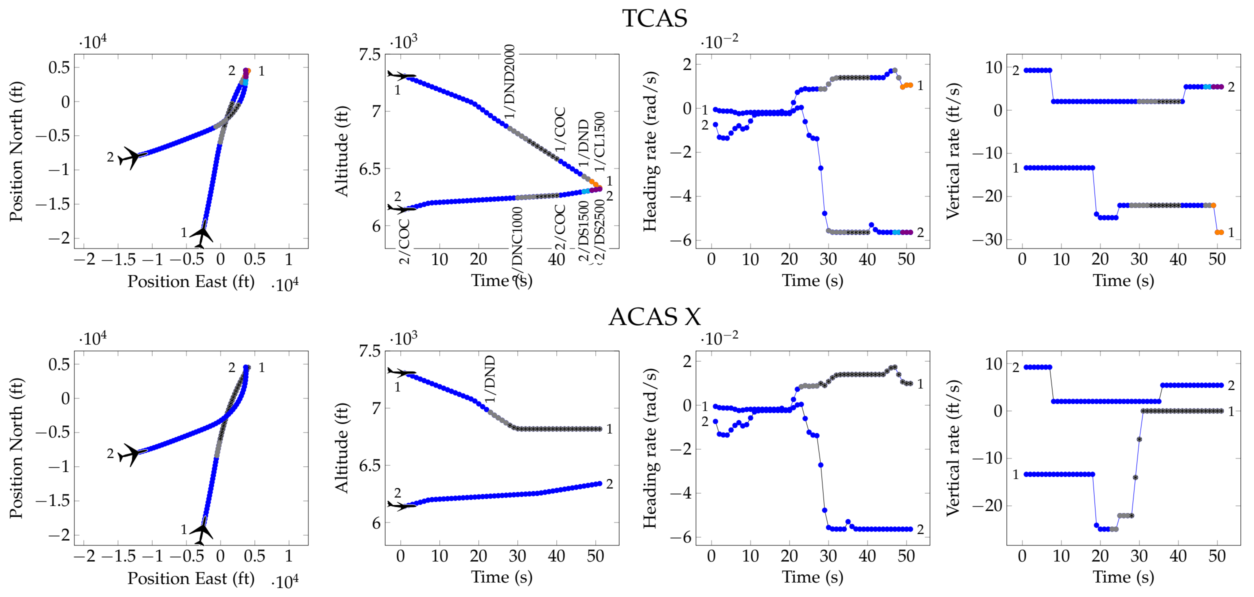


Figure 8.10: NMAC encounter with a double horizontal crossing and a split advisory. In the TCAS simulation, the aircraft cross twice horizontally and split advisories occur at 30 and 47 seconds. NMAC occurs at 50 seconds. ACAS X resolves the encounter using a preventative RA to aircraft 1.

**Triple Altitude Crossing.** An example is shown in Figure 8.11 where an NMAC occurs at 40 seconds in the TCAS simulation but no NMAC occurs in the ACAS X simulation. The encounter occurs with probability density $1.5 \cdot 10^{-9}$, which is more likely than the first two examples presented in this section according to our model. The aircraft approach almost straight, head-on, and co-altitude (Figure 8.11(FL–CL)). TCAS issues initial RAs just as the aircraft cross in altitude (Figure 8.11(TR)(CL)). Responding to the RAs actually results in bringing the aircraft closer together vertically. TCAS subsequently reverses the RAs, but the aircraft cross in altitude for a second time due to the pilot response delay. The aircraft, following the reversal RAs, cross in altitude for a third time, where an NMAC occurs. In contrast, ACAS X is able to resolve the conflict by issuing crossing RAs slightly earlier than TCAS and maintaining the RAs for the duration of the encounter (Figure 8.11(BR)(CL)).

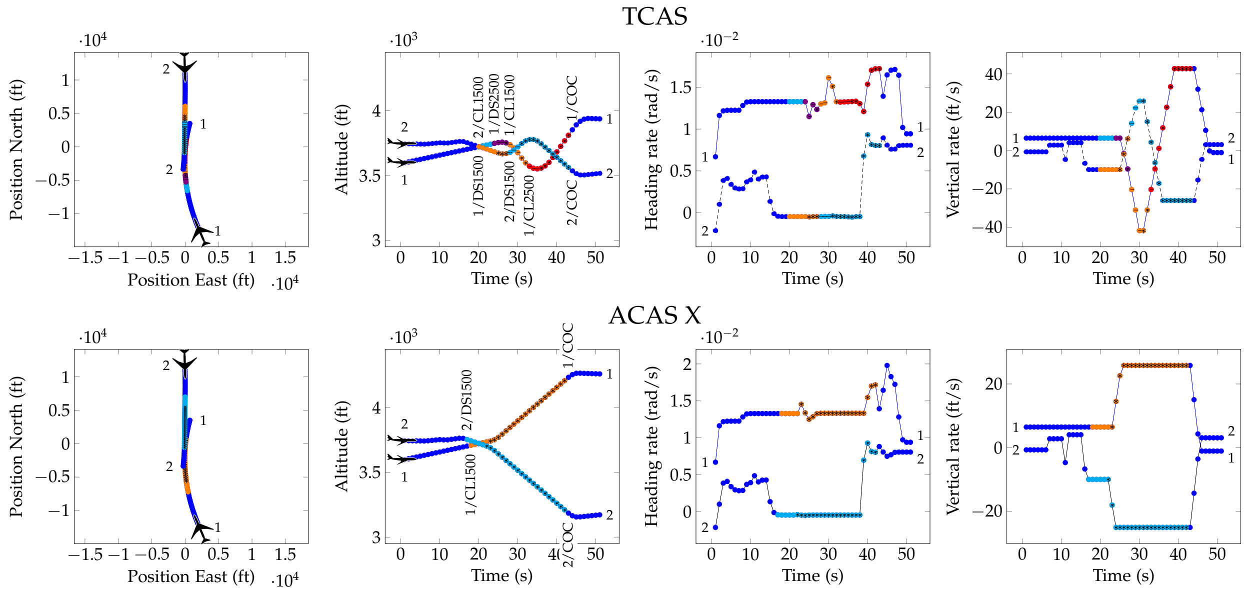


Figure 8.11: NMAC encounter where the aircraft cross in altitude three times at 20, 29, and 40 seconds in the TCAS simulation. NMAC occurs at 40 seconds. ACAS X resolves the conflict by committing to and maintaining a crossing RA.

In the TCAS simulation, interactions between the RA and the pilot response delay lead to oscillations in the aircraft altitudes (Figure 8.11(TR)(CL)). In particular, the response delay causes RAs and their responses to be separated by altitude crossings, resulting in the responses reducing vertical separation rather than increasing it. The oscillations result in multiple altitude crossings and ultimately an NMAC. Another interesting feature of the encounter is the change in vertical rate by aircraft 2 at 17 seconds, which immediately precedes the initial RAs and the oscillations (Figure 8.11(TR)(FR)). Due to the proximity of the maneuver, the effects of state estimation may be playing a significant role in the initial RA. Due to hardware limitations, the collision avoidance system does not directly measure the vertical rate of the intruder and must estimate it by tracking altitude over time. Furthermore, the altitude of the intruder is not known exactly, but quantized to 25 feet increments. To deal with this uncertainty, tracking and filtering techniques are used, which introduce a small amount of tracking error and lag. TCAS uses an alpha-beta filter while ACAS X uses a more sophisticated modified Kalman filter [92]. The modified Kalman filter has been shown to give better tracking error and lag performances. The improved state tracking may be a key contributor to the earlier initial RA and better sense selection in ACAS X in this encounter.

**Vertical Chase.** An example is shown in Figure 8.12 where an NMAC occurs at 43 seconds in the TCAS simulation but no NMAC occurs in the ACAS X simulation. The encounter occurs with probability density $8.4 \cdot 10^{-9}$, which is more likely than the first two examples presented in this section according to our model. In the encounter, the aircraft approach perpendicularly in the horizontal direction and are engaged in a vertical chase. The aircraft are both descending but at different vertical rates. The aircraft cross in altitude and begin to diverge vertically when TCAS issues a crossing RA. Responding to the initial RA, the aircraft cross in altitude for a second time. TCAS then reverses the advisory causing the aircraft to cross in altitude a third time, where an NMAC occurs. In contrast, ACAS X resolves the conflict by issuing a preventative DND advisory early followed by a CL1500 to maintain vertical separation of the aircraft.

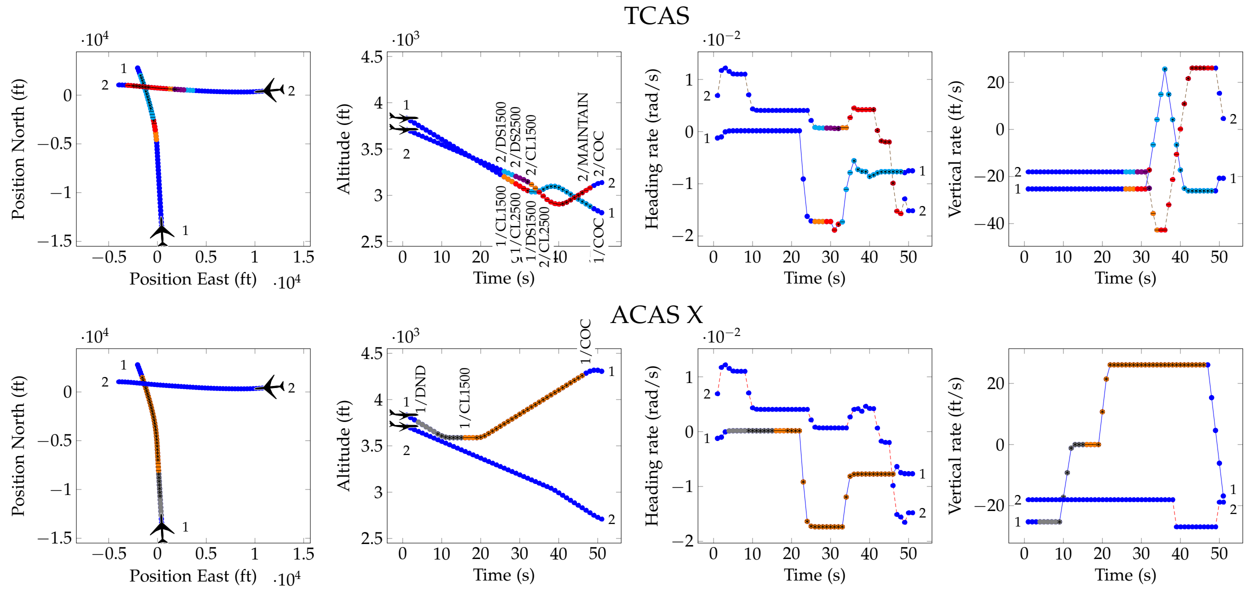


Figure 8.12: NMAC encounter where the aircraft are in a vertical chase configuration resulting in multiple altitude crossings in the TCAS simulation. NMAC occurs at 43 seconds. ACAS X resolves the conflict by issuing RAs much earlier.

The TCAS encounter shows similar oscillations as in the previous example in Figure 8.11 (see Figure 8.12(TR)(CL) and Figure 8.11(TR)(CL)). However, there are a couple of key differences. First, the crossing initial advisories are issued after the aircraft have already crossed in altitude and are beginning to diverge vertically. Second, this encounter has a change in heading rate immediately preceding the initial RAs rather than a change in vertical rate (Figure 8.12(TR)(CR)). As a result, it is likely that both vertical state tracking and the estimation of time to closest horizontal approach are playing a role in the initial RAs.

## 8.5 Categorizing Failure Scenarios using GBDT

Stress testing can produce large datasets of encounters, which are high-dimensional heterogeneous time series. This section aims to discover patterns in the data that may help a domain expert understand common failures and potential underlying issues. Toward this end, we apply GBDT to automatically categorize a dataset of NMAC and non-NMAC encounters and generate explanations that characterize each failure category. In the following sections, we first describe the dataset and the grammar used in the study. Then, we present the categorization results.

### 8.5.1 Dataset

We analyze a dataset that contains simulated two-aircraft mid-air encounters [75]. The dataset contains 10,000 encounters with 863 NMACs and 9,137 non-NMACs. The class imbalance is due to the rarity of NMACs and the difficulty in generating NMAC encounters. Each encounter has 77 features collected at 1 Hertz for 50 time steps. The features include numeric, categorical, and Boolean types representing the state of the aircraft, pilot commands, and the state and output of the collision avoidance system for each aircraft. The data was generated during the stress testing of ACAS X prototype (libcas) version 0.9.8.

Since horizontal separation and vertical separation are features in the data, the NMAC condition (horizontal separation $<$ 500 feet and vertical separation $<$ 100 feet) is directly observable in the data. As a result, GBDT will split on this rule because it gives the best loss. While technically correct, this explanation does not provide useful information about the NMACs. We address the issue by removing the NMAC event from the encounters as follows: For each encounter, we compute the time of closest approach, which is the point where the separation of the two aircraft reaches a minimum, then trim the encounter to retain only data from the start of the encounter to five seconds prior to that time. We choose five seconds to balance between removing NMAC information and leaving sufficient encounter data for prediction.

### 8.5.2 Grammar

We craft a custom CFG for the ACAS X dataset, building on the one presented in Figure 7.1. We include temporal logic operators eventually F and globally G; elementwise logical operators conjunction $\wedge$, disjunction $\vee$, negation $\neg$, and (globally) implies $\implies$; comparison operators less than $<$, less than or equal to $\leq$, and equals $=$; mathematical functions absolute value $|x|$, difference $-$, and sign Sign; and count Count (which returns the number of true values in a Boolean vector).

In addition to dividing the features by data type, the ACAS X grammar further subdivides the features by their physical representations. This enables comparison of features with constant values that have appropriate scale and resolution. For example, even though aircraft heading and vertical rate are both real-valued, aircraft heading should be compared to values between $-180°$ and $180°$, whereas vertical rate should be compared to values between $-80$ and 80 feet per second. The full grammar is included in Appendix A.

### 8.5.3 Results

We performed 20 random trials for each GBDT and baseline algorithm. A trial is a run of an experiment initialized with a unique random seed and train-test split. We use randomly-sampled 70/30 train-test splits of the data and report the mean of the evaluation metrics in Table 8.6. We use the same evaluation metrics and baseline algorithms as in the Australian Sign Language experiment presented in Section 7.4. Evaluation metrics are accuracy, precision, recall, and F1 scores. Baseline algorithms are decision tree, random forest, and Gaussian naive Bayes classifiers. Tree-based classifiers, i.e., GBDT, decision tree, and random forest models, were trained to a maximum depth $d = 4$. Trees that are too deep become difficult to interpret and manage. Random forest used 64 trees. Results for the various variants of GBDT are separately listed in Table 8.6. GBDT-MC uses Monte Carlo algorithm, GBDT-GE uses grammatical evolution, and GBDT-GP uses genetic programming for expression optimization. Overall, GBDT-GP again performed the best. The temporal logic rules in GBDT are able to more effectively capture the underlying patterns and are more robust to temporal variations. Random forest performed very poorly at predicting NMAC. The reason is that random forest randomly subsamples features, which works poorly when there are many features but only a small subset of features is predictive. Random forest uses a voting mechanism for class prediction where each tree in the forest has equal weight. If most of the features are not predictive, then, due to feature subsampling, only a small subset of trees will produce accurate predictions. However, the accurate predictions of these trees will be outweighed by the trees that do not have accurate predictions, resulting in poor predictive performance.

The strength of GBDT lies in its ability to categorize and explain data. GBDT not only provides interpretable expressions about the data's distinguishing properties but also a hierarchical organization of the properties. Figure 8.13 shows a visual overview of the categories extracted from one of the learned

Table 8.6: GBDT performance on ACAS X dataset

| | **GBDT-MC** | **GBDT-GE** | **GBDT-GP** | **Decision Tree** | **Random Forest** | **Gaussian NB** |
|---|---|---|---|---|---|---|
| Accuracy | 0.9577 | 0.9583 | **0.9587** | 0.9378 | 0.9251 | 0.8705 |
| F1-Score | 0.8765 | 0.8768 | **0.8779** | 0.8038 | 0.6130 | 0.7126 |
| Precision | 0.8505 | 0.8546 | 0.8553 | 0.8025 | **0.8961** | 0.6799 |
| Recall | **0.9102** | 0.9050 | 0.9053 | 0.8083 | 0.5778 | 0.8187 |
| Avg. Terminals | 5.54 | 5.56 | **5.38** | — | — | — |

trees. The figure shows plots of altitude versus time, which, while it cannot fully capture the high-dimensionality of the encounter data, is generally most informative since ACAS X issues RAs only in the vertical direction. Since we are mainly interested in categorizing NMACs, we consider only the six NMAC categories out of the 16 total categories produced by the tree. Figure 8.13 shows the first five encounters for each of the six NMAC categories, where each row is a separate category. The categories are labeled in ascending order starting at category 1 at the top row. The first row has only two plots because category 1 only contained these two encounters. A full representation of the GBDT is included in Appendix B. We describe the categories as follows.

**Category 1.** In this category, the two aircraft maintain altitude separation for most of the encounter, then one aircraft accelerates rapidly toward the other aircraft as they approach. The aggressive maneuvering causes vertical separation to be lost rapidly and an NMAC results. Figure 8.14 shows the spike in climb rate at 34 seconds, only five seconds before NMAC. The large spike in vertical rate near NMAC is characteristic of encounters in this category.

Aggressive last-minute maneuvering is known to be problematic. With the pilot's five-second response time, there is insufficient time remaining for the collision avoidance system to resolve the conflict. It is unlikely that any reasonable collision avoidance system would be able to resolve such NMACs. Fortunately, these encounters are extremely rare outside of simulation since pilots do not typically maneuver so aggressively in operations.

**Category 2.** The encounters in this category are also characterized by a maintenance of vertical separation in the first part of the encounter before aggressive maneuvering close to closest point of approach (CPA). The relative vertical rate of the aircraft exceeds 50 feet per second at some point in the encounter. These encounters differ from those in category 1 in that the aircraft maintains a greater altitude separation in the early part of the encounter and both aircraft maneuver much more aggressively toward each other than those in the previous category. The aggressiveness of the pilot's maneuvering is revealed in the GBDT explanation that the commanded vertical rate differs from the advisory's target rate in excess of 30 feet per second. That is, the pilots are very aggressively maneuvering against the RA

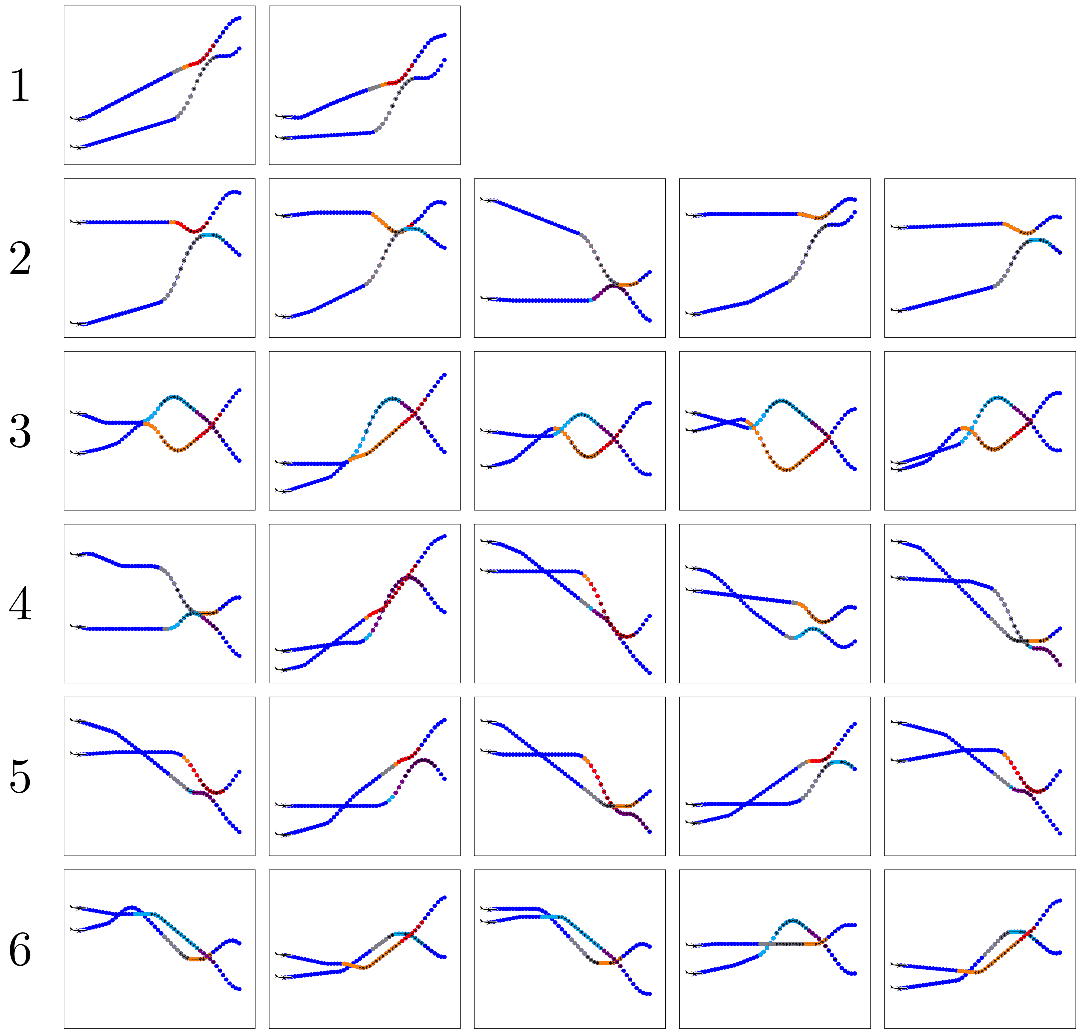


Figure 8.13: Visual overview of categorized ACAS X encounter data. Each row is a category. Columns show altitude profiles of five encounters in each category (except category 1, which only contained two encounters). We see visual similarities within each category.

during pilot response delay after the RA is issued, but before they start complying.

**Category 3.** Similar to categories 1 and 2, the encounters in this category also contain aggressive vertical maneuvering at some point in the encounter. However, the aggressive maneuvering occurs much earlier in the encounter causing one or more altitude crossings early on and results in active RAs throughout most of the encounter. Figure 8.15 shows the vertical rates of an encounter where the large difference in vertical rate can be seen well prior to CPA in the encounter. The large difference in vertical rates exceeding 50 feet per second is seen at 25 seconds and an NMAC occurs at the second altitude crossing at 37 seconds.

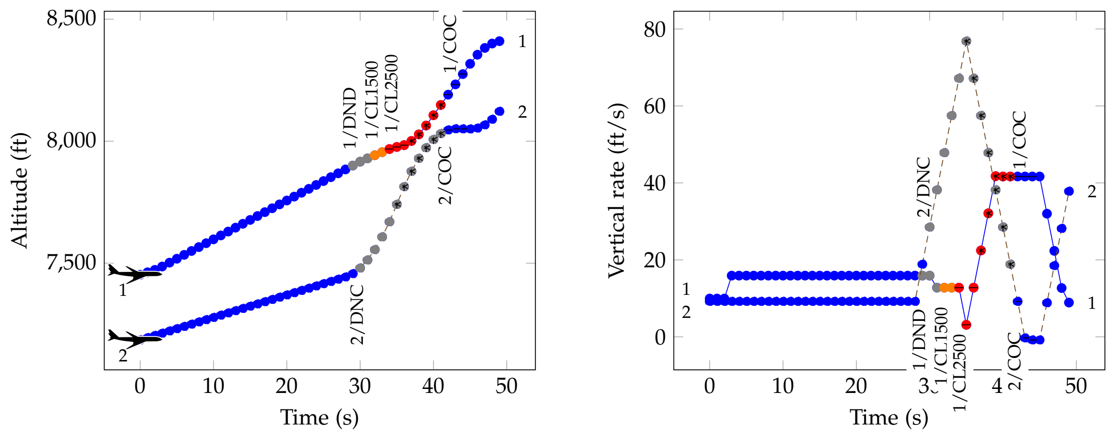


Figure 8.14: Vertical maneuvering just before closest point of approach (CPA). Aircraft 2 climbs aggressively starting at 29 seconds and reaching a peak climb rate of almost 80 feet per second (right) shortly before NMAC at 40 seconds (left).

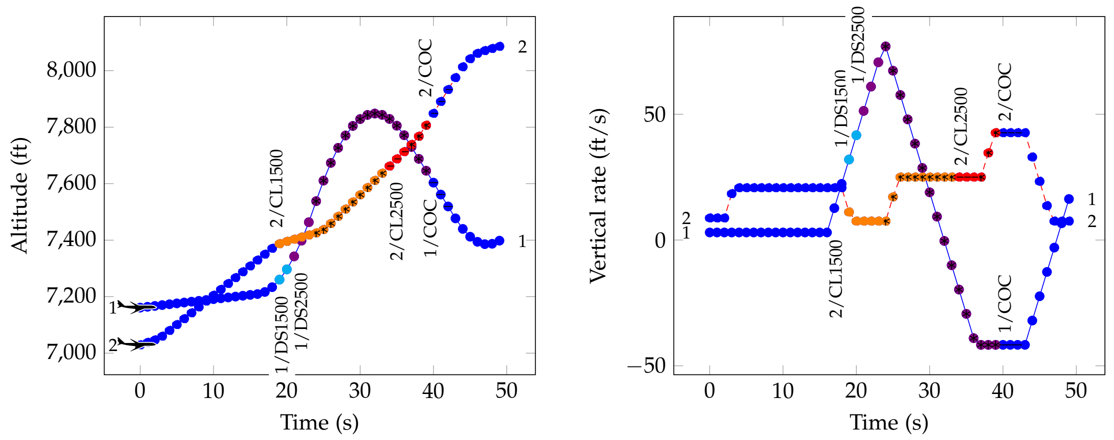


Figure 8.15: Vertical maneuvering early in encounter. Aircraft 1 climbs aggressively starting at 18 seconds and peaking at 25 seconds (right). An NMAC occurs at 37 seconds (left).

**Category 4.** The encounters in this category also contain aggressive vertical maneuvering close to CPA, characterized by the commanded vertical rates of the two aircraft differing by more than 50 feet per second at some point in the encounter. Loss of vertical separation occurs at some point before five seconds prior to CPA and is manifested either very early in the encounter or as an extended period where the aircraft are co-altitude near CPA. Compared to category 3, the initial RA occurs later in the encounter, so the pilots are flying their intended paths for most of the encounter. As with the previous categories, strong aggressive maneuvering against the initial RA is extremely rare in operation since it represents strongly uncooperative pilot behavior.

**Category 5.** In these encounters, the aircraft cross in altitude early in the encounter without active advisories, then maneuver back toward each other to cause an NMAC. Since the aircraft are predicted to cross safely and appear to vertically diverge following the crossing, the collision avoidance system witholds an RA to allow the encounter to resolve naturally and reduce the number of unnecessary alerts. However, after crossing, the aircraft change course and maneuver toward each other, which results in an NMAC. Because the aircraft are already close in altitude, the vertical maneuvering in these encounters is not as aggressive as those in previous categories. Due to the late maneuvering and the pilot response delay, pilot 2 does not start to comply with the issued RA until within five seconds of NMAC. Figure 8.16 shows the vertical profile of an encounter in this category where the aircraft cross in altitude at 19 seconds and then maneuver to NMAC at 38 seconds.

In addition to the RA, the collision avoidance system also has additional mechanisms for alerting the pilot, which we have not modeled. The collision avoidance system provides visual displays of the location of other aircraft, and also issues a traffic alert (TA), which alerts the pilots in advance of an RA. These mechanisms help increase the pilots' situational awareness and reduce blunders.

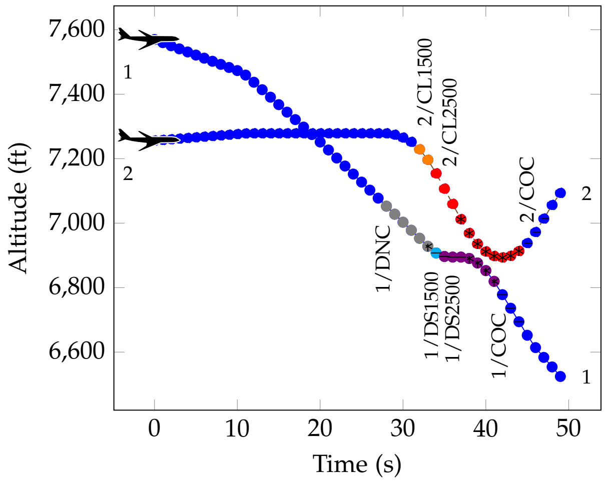


Figure 8.16: Maneuvering following altitude crossing. The aircraft cross in altitude without advisory at 19 seconds then maneuver toward each other to NMAC at 38 seconds.

**Category 6.** In these encounters, the aircraft receive initial RAs at different times and the aircraft cross in altitude during the pilot response delay of the first RA. As the pilots start responding to their advisories, the maneuvers actually result in bringing them closer together rather than further apart. A late revision to the advisory is issued. However, the encounter ultimately results in an NMAC. Figure 8.17 shows the vertical profile of an encounter in this category. The aircraft cross in altitude during pilot 2's response delay period at 21 seconds and NMAC occurs at 39 seconds.

Issuing advisories can be tricky in cases where the aircraft are approximately co-altitude due to uncertainties in the aircraft's estimated positions and future intentions. In these encounters, the problem arises as one aircraft maneuvers to cross in altitude immediately after an initial RA is issued to the other aircraft. Operationally, this is a rare case as the positions of the aircraft and the timing of the maneuver need to coincide.

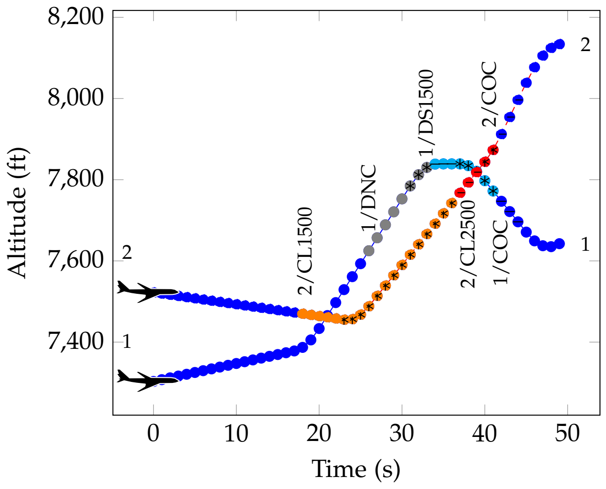
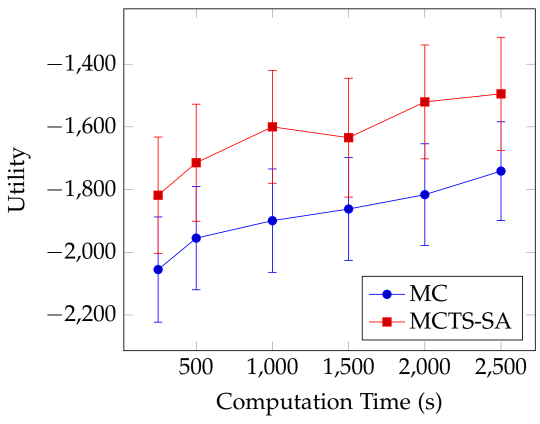


Figure 8.17: Altitude crossing during pilot response delay. The aircraft cross in altitude during pilot 2's response delay at 21 seconds. The aircraft lose vertical separation as they comply with their RAs and an NMAC occurs at 39 seconds.

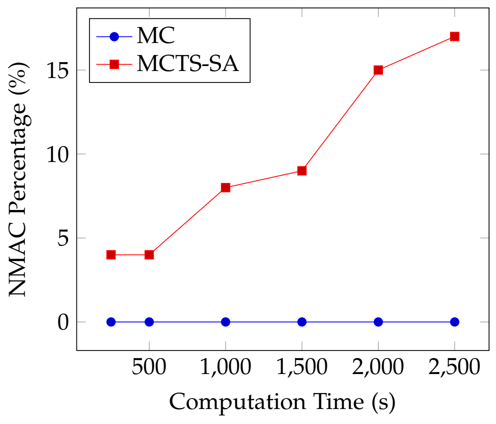
## 8.6 Performance Evaluation of AST

This section evaluates the performance of MCTS-SA against direct Monte Carlo sampling given a fixed computational budget. MCTS-SA was used in Section 8.3 and Section 8.4. The algorithms are given a fixed amount of CPU time and the best path found at the end of that time is returned. We compare the CPU time of the algorithms rather than number of samples to account for the additional computations performed in the MCTS-SA algorithm. We use the same configuration as the single-threat encounter experiments as shown in Table 8.3 except that we limit the search based on computation time instead of a fixed number of iterations. The experiments were performed on a laptop with an Intel i7 4700HQ quad-core processor and 32 GB of memory.

Figure 8.18 shows the performance of the two algorithms as computation time varies. Figure 8.18a compares the utility (as defined in Section 3.1) of the best encounters found by the algorithms. Each data point shows the mean and standard error of the mean of 100 pairwise encounters. Figure 8.18b shows the percentage of NMACs found out of the 100 encounters searched. In both cases, MCTS-SA clearly outperforms the baseline Monte Carlo search. The effectiveness of MCTS-SA in finding NMACs is particularly important and we see that MCTS-SA greatly outperforms the baseline in this regard. As the computational budget increases, MCTS-SA is able to find increasingly many NMACs, whereas within the computational budgets considered, direct Monte Carlo is unable to find any NMACs.



(a)



(b)

Figure 8.18: Performance of MCTS and direct Monte Carlo with computation time. MCTS-SA clearly outperforms direct Monte Carlo in both achieving better utility (left) and NMAC find rate (right).

## 8.7 Summary

This chapter presented several studies that apply AdaStress to analyze near mid-air collision (NMAC) scenarios in the next-generation Airborne Collision Avoidance System (ACAS X). We applied AST to find the most likely NMAC scenarios of ACAS X in single-threat and multi-threat encounters and identified a number of NMAC categories. For single-threat encounters, we identified NMAC scenarios related to crossing during the pilot response delay, high horizontal and vertical rates, maneuvering against the initial RA, sudden aggressive maneuvering, and a combination of the above factors. For multi-threat encounters, we identified NMAC encounters related to pairwise coordination, limited maneuverable airspace, and phenomena we've previously seen in pairwise encounters.

We applied DAST to compare the NMAC behavior of ACAS X and TCAS. In particular, we searched for the most likely NMAC scenarios of ACAS X relative to a TCAS baseline, and also the most likely NMAC scenarios of TCAS relative to an ACAS X baseline. In the first study, we found cases where ACAS X issues an RA but TCAS does not, cases involving simultaneous horizontal and vertical maneuvering, and cases with horizontal maneuvering alone. In the second study, testing TCAS relative to ACAS X as baseline, we found 39.3% more NMACs originating from a broader set of initial encounters than in the first study. These results agree with other statistical studies comparing ACAS X and TCAS and give us additional confidence that ACAS X yields improved safety and performance over TCAS. In this study, we found NMAC scenarios that involve a crossing RA that is reversed just prior to altitude crossing, double horizontal crossing, triple altitude crossing, and vertical chase.

We applied GBDT to categorize and explain a dataset of aircraft encounters, where some encounters had NMAC and some did not. We used a domain-specific grammar that produced Boolean expressions based on a customized linear temporal logic. The GBDT automatically identified and characterized six NMAC categories, including various variations of aggressive maneuvering, crossing altitude and maneuvering back, and altitude crossing during pilot response delay.

We summarize the failure categories identified in our various analyses of ACAS X in Tables 8.7–8.9. Table 8.7 summarizes the failure categories identified in single-threat and multi-threat AST analyses. Table 8.8 summarizes the failure categories identified in ACAS X and TCAS using DAST. Lastly, Table 8.9 summarizes the failure categories identified through GBDT analysis.

Finally, we compared the performance of MCTS-SA with direct Monte Carlo search and showed that MCTS-SA is far more effective at finding NMACs than direct Monte Carlo search. MCTS-SA was also able to find increasingly more NMACs with increasing computational budget, whereas direct Monte Carlo search struggled to find any NMACs at comparable budgets.

| **AST Single-Threat** | | |
|---|---|---|
| **ID** | **Failure Category** | **Description** |
| 1 | Crossing time | Altitude crossing occurs during pilot response delay, executing RA reduces vertical separation |
| 2 | High turn & vertical rates | High turn rate shortens time to closest approach, high vertical rate requires more time to maneuver and comply with RA |
| 3 | Maneuvering against RA | Pilot maneuvers against RA during response delay before complying |
| 4 | Sudden aggressive maneuvering | Aircraft fly straight and level then maneuver aggressively toward each other just before they pass |
| 5 | Combined factors | A combination of the aforementioned behaviors |
| **AST Multi-Threat** | | |
| **ID** | **Failure Category** | **Description** |
| 1 | Pairwise coordination in multi-threat | Coordination issue arising from using pairwise coordination mechanism in multi-threat setting |
| 2 | Limited maneuverable space | Following an RA in a multi-aircraft setting can lead to secondary conflict |
| 3 | Pairwise phenomena | Single-threat issues also appear in multi-threat encounters |

Table 8.7: Summary of failure categories discovered in ACAS X using AST

| **DAST ACAS X vs. TCAS** | | |
|---|---|---|
| **ID** | **Failure Category** | **Description** |
| 1 | ACAS X issues RA, TCAS does not | ACAS X issues RA and NMAC occurs, TCAS does not RA but encounter resolves on its own |
| 2 | Simultaneous horizontal and vertical maneuvering | Aircraft turns shorten time to closest approach, simultaneous climb/descend at high rate increases time to comply |
| 3 | Horizontal maneuvering | Horizontal maneuvering only can lead to NMAC in some encounters |
| **DAST TCAS vs. ACAS X** | | |
| **ID** | **Failure Category** | **Description** |
| 1 | Crossing RA followed by reversal | Crossing RA issued, reversal issued shortly before altitude crossing |
| 2 | Double horizontal crossing | Aircraft cross horizontally then continue turn into NMAC, split advisory |
| 3 | Triple altitude crossing | Multiple altitude crossings resulting from interactions between RAs and pilot response delays |
| 4 | Vertical chase | Both aircraft are climbing/descending while one overtakes, close altitude adds uncertainty |

Table 8.8: Summary of failure categories discovered in ACAS X and TCAS using DAST

| ID | Category Description |
|---|---|
| 1 | Aggressive maneuvering just prior to closest point of approach (CPA) |
| 2 | Aggressive maneuvering near CPA, wider initial separation, more aggressive maneuvering than 1 |
| 3 | Aggressive maneuvering early in the encounter |
| 4 | Aggressive maneuvering near CPA, early loss of vertical separation or extended co-altitude period before CPA |
| 5 | Altitude crossing without RA followed by maneuvering near CPA |
| 6 | Altitude crossing during pilot response delay |

Table 8.9: Summary of failure categories discovered in ACAS X using GBDT

# Chapter 9

# Application: Trajectory Planning for Unmanned Aircraft

This chapter applies AdaStress on a trajectory planning system, where we find and analyze scenarios of obstacle collisions and planning failures in a UAV navigation task. The first section, Section 9.1, introduces the topic. Section 9.2 provides an overview of trajectory planning systems. Section 9.3 discusses how AST can be used to stress test trajectory planning systems, including system architectures, sources of uncertainty, and failure modes. Section 9.4 presents a case study of stress testing a prototype trajectory planning system in a UAV navigation application, and discusses the categories of failure scenarios that the algorithm uncovered. The final section, Section 9.5, summarizes the contributions of this chapter.

## 9.1 Introduction

Trajectory planners play a critical role in many autonomous vehicle systems, such as unmanned aircraft and driverless cars. However, analyzing the safety of these systems is very challenging due to their complexity and operation over time. Trajectory planners vary widely depending on their modeling assumptions, search space, optimization algorithm, and runtime behavior. Existing validation approaches are generally focused on the correctness of the plans in isolation, while ignoring that the planner may not be able to generate a valid plan at all due to unanticipated operational conditions.

The validation of trajectory planning systems has generally relied on simulation [93, 94, 95, 96, 97] and experimental validation [98, 99]. Random perturbations can be used to increase the range of scenarios that are tested. However, the limited coverage of these approaches, together with the large complex state space and rarity of failures, make it difficult to uncover rare corner case failure scenarios. Another approach uses formal mathematical arguments on geometries to prove correctness [100]. More generally, formal

V&V methods have been applied to planning and scheduling problems. These methods apply model checking to verify properties on various artifacts of the planning system such as the plan, the planning (or domain) model, and the planning algorithm [101, 102, 103]. Simulated mission scenarios can often uncover divergences between the intended behavior of a plan and its simulated outcome [104]. Ongoing research work aims to automatically analyze execution time failures to detect modeling flaws and suggest fixes to the planning models and abstractions [104].

This chapter takes a simulation-based approach that models a scenario at the level of components, including the trajectory planning system, vehicle dynamics, sensors, environment, and how they interact in operation. We consider how the scenario evolves under various sources of stochastic disturbances and how the system can fail. However, rather than Monte Carlo sampling over these stochastic sources as is traditionally done in simulation-based testing, we propose to explicitly and adversarially optimize the stochastic elements in the simulator to drive the simulation toward failure events. This approach can be much more efficient at exploring the complex state space and finding failures.

We apply AST to find the most likely failure scenarios by posing the problem as a sequential decision-making problem and then optimizing it using MCTS. We apply the approach presented in Section 5.4.1. Specifically, we create a seed-action simulator of the scenario, and then apply MCTS-SA to search for failure scenarios.

In the following sections, we first give an overview of trajectory planning systems, and then describe how to set up and effectively apply AST to test trajectory planning systems. Then we present experiments applying the proposed approach to find failure scenarios in a prototype trajectory planning system for a small unmanned aircraft [105, 106].

## 9.2 Trajectory Planning Systems Overview

Trajectory planners perceive the environment via sensors and produce path plans for the vehicle to follow. Planning algorithms optimize some objective, such as reaching a goal while minimizing fuel consumption, and may do so under constraints, such as the vehicle not colliding with obstacles or other vehicles. This chapter assumes a two time-scale system architecture, where a trajectory planner produces plans over longer time horizons while a shorter time horizon controller guides the vehicle to follow the plan. While trajectory planning systems can vary widely, they can be generally characterized along the following dimensions.

**Modeling.** Planning is computationally intensive. As a result, planners often make simplifying assumptions and plan using abstract models. For example, the planner may assume simplified vehicle dynamics or approximate complex obstacles using polygons of simple shape. Plans can also depend on preexisting knowledge of the world, such as preloaded terrain maps. Discrepancies between the planning model and the real world can represent significant sources of failures.

**Search Space and Solution Representation.** While trajectory planners all output vehicle paths, they can differ in how they represent these paths. For example, the plan can be expressed as a sequence of waypoints [93, 107] or a spline [105]. The plans can be open-loop plans, which do not account for future information being available, or closed-loop plans that do [31]. The choice of representation defines the space of solutions that can be found.

**Optimization Algorithm.** A broad range of search algorithms exist for trajectory planning. For example, the algorithms can be stochastic [42, 97, 108] or deterministic [109], i.e., give the same output given the same input; search tree-based [93, 108], differential equation-based [109], evolutionary algorithms-based [96], or playbook-based (based on combinations of a small set of base trajectories) [107]. Some optimization algorithms can operate within a given time constraint, i.e., real-time or anytime algorithms, [42, 93, 110], while others must run to completion [109]. Some algorithms rely on heuristics [110]. Different optimization algorithms may be able to handle different types of constraints on the solution.

**Operational Behavior.** Trajectory planners may use additional logic at runtime. For example, planners may update their plans, i.e., *replan*, at regular time intervals; or replan if certain criteria are met. For example, the planner may replan if the vehicle deviates too much from the plan or if the plan is now believed to lead to a collision.

Because we take a simulation-based approach to testing, we can support a broad range of trajectory planning systems without modification to the stress testing algorithm. All that is required is a way to simulate the trajectory planner to produce the plans. Our approach embeds the trajectory planning software system in the simulation.

## 9.3 Adaptive Stress Testing for Trajectory Planning Systems

This section describes how adaptive stress testing (AST) can be applied to stress test trajectory planning systems. We first describe the overall stress testing architecture. Then we describe various failure sources and failure modes relevant to the operation of trajectory planning systems and how these can be expressed in the AST framework.

### 9.3.1 Stress Testing Architecture

The overall system architecture is shown in Figure 9.1. The seed-action simulator $\bar{S}$ models an autonomous vehicle in operation. The vehicle's sensors perceive the external environment, which may include wind, obstacles, and other vehicles, and provide noisy observations to the trajectory planner and the vehicle controller. The planner, which is the system under test, computes a plan based on these observations. The vehicle controller controls the vehicle to follow the plan and the vehicle state evolves according to the vehicle dynamics. The vehicle state is used to update the environment state and the cycle repeats.

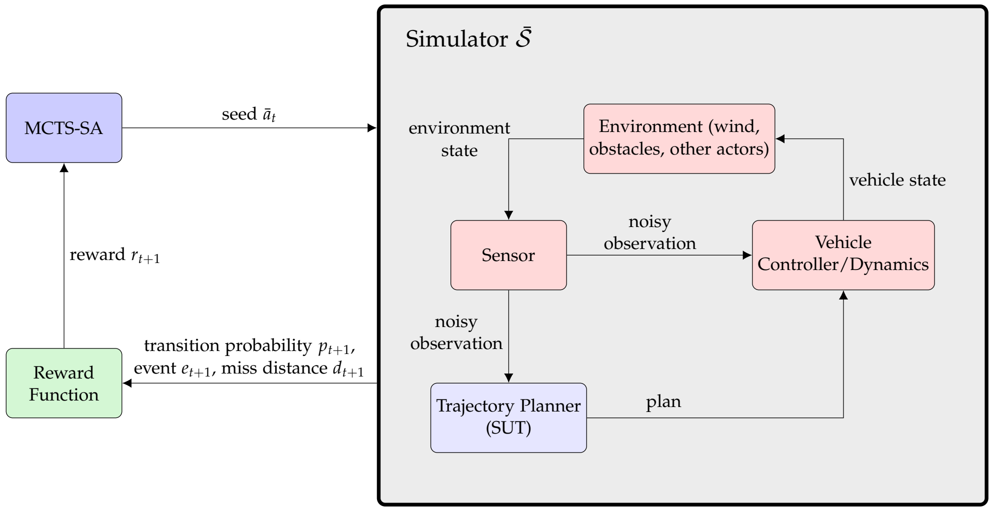


Figure 9.1: Architecture for adaptive stress testing of trajectory planning systems. Seed-action simulator models trajectory planner and its environment. MCTS-SA is used to search the simulator.

The simulation is abstracted following the seed-action framework in Section 5.4.1. The simulator is Markov and maintains a state, but the state is not exposed externally. Without loss of generality, we assume multivariate heterogeneous state and actions internal to the simulator. Following the seed-action framework, the simulator takes a pseudorandom seed $\bar{a}$ as input and uses it to set the state of the simulator's pseudorandom number generator. The output of the simulator consists of three variables: a variable $e_{t+1}$ indicating whether a failure event occurred, the probability $p_{t+1}$ of the state transition, and some measure $d_{t+1}$ of how close the simulator came to an event. The reward function converts these variables into a single reward metric that the reinforcement learner optimizes. We use the MCTS-SA algorithm presented in Section 5.4.3 for optimization.

### 9.3.2 Sources of Uncertainty in Trajectory Planning

A number of sources of uncertainty exist in the operation of trajectory planning systems. We can model these uncertainties using stochastic models inside the various components in the simulator. The stochastic variables are the actions in the AST framework being optimized. The optimization algorithm drives the stochastic variables to maximize the overall adversarial impact and probability of occurrence on the system under test. We consider the following sources of uncertainty in the system.

**Imperfect Sensors.** Trajectory planners rely on sensors to perceive the environment. Sensor observations can be noisy and mislead a planner into producing suboptimal plans. For example, observation noise in an obstacle's location can result in a plan in which the vehicle collides with the obstacle. The effect of noisy sensors is captured in the sensor component of the simulator. The sensor takes an environment state as input and outputs a noisy observation to the trajectory planner and vehicle controller. In addition to being noisy, sensors can give a partial view of the world. Important features may be out of range or occluded. As a result, planners must plan based on incomplete information, which may result in suboptimal plans. For example, the planner may guide the vehicle toward a hazard that is occluded by obstacles. The sensor model can be used to capture the effect of incomplete information mapping the global environment state to local observations.

**Changing Environment.** A changing environment can drastically degrade the quality of plans. The specific stochastic elements present in the environment depend on the application. For example, small aircraft may be susceptible to wind disturbances [111]. Surveillance and rescue missions may encounter natural hazards that evolve stochastically over time, such as tree branches that sway in the wind, or fires and smoke [112]. The environment may also contain elements that behave strategically, such as other vehicle agents [113]. Multi-agent environments, especially under limited communication, are particularly challenging as the other agents may behave drastically and unexpectedly invalidate existing plans. Planners that cannot adequately account for the uncertainty in the environment may put the vehicle at risk under certain disturbance scenarios. These stochastic effects in the vehicle's environment can be modeled in the environment component of the simulation. We can model the stochastic processes or embed external physics simulators. External simulators can be used as long as the pseudorandom seed can be exposed to control any randomness.

**Vehicle Control and Dynamics.** Closed-loop control of unmanned vehicles is generally well-understood. However, there may be small uncertainties in the evolution of the vehicle due to unmodeled dynamics or uncertainties in actuation. In the AST framework, we can model these uncertainties in the vehicle component using a stochastic model over the vehicle's state transition.

**Computation Time of Planning.** Trajectory planning is computationally intensive, so generating a plan may require significant computation time. In many missions, vehicles do not stop to wait for plans to compute. Indeed, some vehicles, such as fixed-wing aircraft, cannot stop arbitrarily in mid-flight. But even for vehicles that are able to do so, this mode of operation is often too disruptive. More commonly, vehicles continue following the existing plan while a new one is being computed, which can lead to a couple of issues. First, the existing plan may be invalid and may become increasingly hazardous to the vehicle as replanning time increases. Second, there needs to be some transition between the existing plan and the new plan. If the plans are very different, for example as a result of replanning taking a very long time, then the transition may introduce additional uncertainties, such as potential collisions.

Planning delays can be modeled in the trajectory planning component using a timed queue. Generated plans do not immediately take effect, but are added to the queue to take effect at a later time. The delay period can be stochastic or fixed. For added realism, the delay can be modeled on the wall clock computation time of the planner. The vehicle continues to follow the previous plan during the delay.

### 9.3.3 Failure Modes in Trajectory Planning

Trajectory planners can fail in a variety of ways. This section describes some of these failure modes. In the AST framework, we can capture the failures of interest in the definition of an event $E$, which is a subset of the state space $S$ (Section 5.2). An event can be defined by an indicator function IsEvent, which considers the simulator state and outputs whether a failure event has occurred. For example, an event can be a collision defined as the vehicle being closer than a certain distance from an obstacle. As another example, an event can be a software crash in the planner. To help guide the search algorithm, we can also define a measure that hints to the optimization algorithm how close the simulator came to an event, called the miss distance. One example is to provide the distance between the vehicle and the closest obstacle. It may be very difficult to define a distance measure for some failure modes, such as software crashes. In these cases, we do not provide a distance measure and the search algorithm cannot explicitly drive the search toward these failures. However, they can still be detected if they are encountered and they often do arise as the system is put under stress. We consider the following types of failures for trajectory planning systems.

**Vehicle Enters a Failed State.** The vehicle enters some failed state, such as a colliding with an obstacle or running out of fuel. For example, significant observation noise in an obstacle's location may result in a plan that travels through the obstacle instead of around it. Another example is a plan that takes the vehicle too close to an obstacle and a wind disturbance pushes the vehicle into the obstacle. Vehicle failed states are generally well-defined, so providing a function that detects them is straightforward. Distance measures to failures are also often available. For example, distances to obstacles can be used for collisions, the amount of fuel remaining can be used for out-of-fuel scenarios, and mission time and distance to goal can be used for not reaching goal within a certain time.

**Replanning Issues.** Trajectory planners can replan at fixed time intervals or when a certain criterion is met, such as if the current plan may lead to a collision. A variety of replanning issues can occur. Noisy observations can lead to excessive triggering of the replanning condition leading to the trajectory planner constantly replanning. Infinite trajectory loops can also arise due to partial observability, where the planner may lead a vehicle into and out of a suboptimal area, then later do it again. Many simple planners that do not retain sufficient information across time are susceptible to this type of failure. In the AST framework, we can detect that the number of replans becomes excessive or if the vehicle does not complete its mission in the allotted time. The number of replans and the mission duration can be used as distance measures to the search algorithm.

**Cannot Find a Plan.** It is not guaranteed that the planner produces a valid plan that satisfies all constraints, e.g., collision-free and meets vehicle dynamic constraints. A valid plan may not exist given the current vehicle state or the algorithm cannot find it. In these cases, the planner may return indicating that it has failed to produce a plan. We can detect and treat such planning failures as failure events. In some cases, given access to the internal details of the planner, we may be able to use some computational metrics such as the number of iterations as a distance measure. In this work, we detect but do not specify a distance metric for planning failures.

**Implementation Issues.** Failures to produce a valid plan can also be caused by implementation issues, such as software crashes, software entering an infinite loop and not completing, or the software producing erroneous output. We treat these events as failure events. However, it is unclear how to define a distance measure that leads to these software issues. In this work, we detect, but do not specify a distance measure for implementation issues.

## 9.4 Stress Testing a Trajectory Planner for a Search and Rescue UAV

We apply AST to stress test a research prototype trajectory planning system [105]. The system is being developed as part of the Autonomy Teaming & TRAjectories for Complex Trusted Operational Reliability (ATTRACTOR) project led by researchers at NASA Langley. A later version of this trajectory planner is planned to be used in a UAV search and rescue demonstration mission [106]. Preliminary testing of the system is being performed on an indoor testbed, called *WireMaze*, which we use as the basis for our simulations and testing. We conduct experiments using AST to perturb various stochastic elements of the simulation to discover examples of failure scenarios of the trajectory planner. Failure events are defined to be collisions and planning failures and we use the distance to the nearest wire obstacle as the miss distance. In the following sections, we first provide a general overview of the trajectory planning system and the WireMaze scenario. Then we present our experiments stress testing the trajectory planning system and discuss the issues we discovered from our testing.

### 9.4.1 Silhouette-Informed Trajectory Generation

We stress test a prototype trajectory planner provided as executable source code [105]. We provide a brief overview of the system, and refer the reader to the original paper for a detailed description [105]. At a high level, the planner takes as input a start position, a goal position, a list of wire obstacles, and a number of parameters. The start and goal positions are specified in three dimensions. Wire obstacles are straight and defined by their start and end points. The parameters govern various aspects of the algorithm. The system outputs a trajectory that connects the start and goal positions if it is found, otherwise outputs an error code indicating the type of failure. The trajectory is represented as a sequence of Bezier curves providing a smooth continuous path for the vehicle. Combined with a speed profile, the planner can sample along the path to provide a sequence of waypoints for the vehicle to follow. We assume in this study a constant speed profile.

Internally, the planner computes the trajectory in three stages. The first stage is a stochastic tree-based trajectory search based on a variant of the rapidly-exploring random tree (RRT) algorithm [114], called RRT$^*$ [115]. The RRT$^*$ algorithm randomly samples points in the planning space and tries to grow a search tree toward that point by steering the nearest point in the tree toward the target point. For optimizing vehicle flight trajectories, which are paths in three-dimensional space, RRT$^*$ samples target points and grows a tree in three-dimensional space. The algorithm continues to grow the tree until the tree reaches a point sufficiently close to the goal position. The algorithm returns the sequence of points that connects the root of the tree to the point closest to the goal position. RRT$^*$ is guaranteed to find the optimal path to the goal with an infinite number of samples. For finite samples, the algorithm may not find any feasible paths to the goal and returns with the search tree so far. In the original RRT$^*$ algorithm, path segments are checked for collision before they are added to the tree. If the new segment collides with an obstacle, then the segment is simply discarded so as to not add collision path segments to the tree.

The ATTRACTOR trajectory planner implements a variant of the RRT$^*$ algorithm that uses extra information about the obstacles to accelerate the search around obstacles. Specifically, this variant, called silhouette-informed tree (SIT), uses silhouette information about the obstacles to generate sample points around the obstacles if a collision is detected, which facilitates planning around the obstacles [105]. This implementation also occasionally uses the goal point as the target point to help the search tree grow toward the goal position.

The second stage of the trajectory planner performs path smoothing. The SIT algorithm generates a piecewise linear trajectory. The path smoothing stage takes this solution and tries to approximate it using Bezier curves so that the final solution is $\mathcal{C}^2$ continuous. A $\mathcal{C}^2$ solution is one where the second derivative of the position with respect to time exists and is continuous everywhere along the path. In the third and final stage, temporal specifications are incorporated through a speed profile with a polynomial structure that determines the arrival time, and allows the evaluation of the vehicle's position at any instance in time. The motivation behind generating $\mathcal{C}^2$ continuous trajectories is threefold: (1) The trajectories result in continuous and nicely behaved control commands when using a path following algorithm to ensure the vehicle tracks the desired path and speed profile; (2) they provide a naturally looking trajectory that can be pulled, clipped, and stretched when interacting with human operators; and (3) they exploit existing algorithms that avoid sampling of the trajectory to make collision queries, and avoids committing to a particular time discretization, which can be problematic when used to answer queries involving multiple time scales.

### 9.4.2 WireMaze Scenario

WireMaze is a three-dimensional structure with 42 wires strung across it at various positions and angles forming a three-dimensional maze [105]. The structure is a cube with a side length of 2.7 meters and stands 8.9 centimeters above the ground. The scenario involves a small UAV equipped with the trajectory planning system described above, using it to navigate from a start position on one side of the maze to a goal position on the other. Figure 9.2 shows a visualization of the maze and scenario.

The scenario simulates closed-loop operation of the trajectory planner as the UAV navigates through the maze and tries to avoid the wire obstacles. At the time of this study, the planner is in its preliminary stages of development and is designed for open-loop planning. As a result, our simulations are anticipating how the planner might plausibly be used in closed-loop in the future. The simulation models the sensors, the aircraft dynamics, and the wire maze. The trajectory planner code is directly embedded in the simulator as an executable component.

The basic scenario begins with the UAV at a start position and computes a plan to reach the goal position. From that point, the simulation advances at 1 Hertz with the position of the aircraft following the waypoints of the plan. Gaussian noise with mean zero and standard deviation $\sigma_{pos}$ is added to the position of the vehicle simulating uncertainties in the vehicle's ability to exactly follow the given waypoints. Deciding whether to replan is governed by a *replanning condition* specified by the design of the mission. If the replanning condition is met, the planner is queried to generate a new plan given the

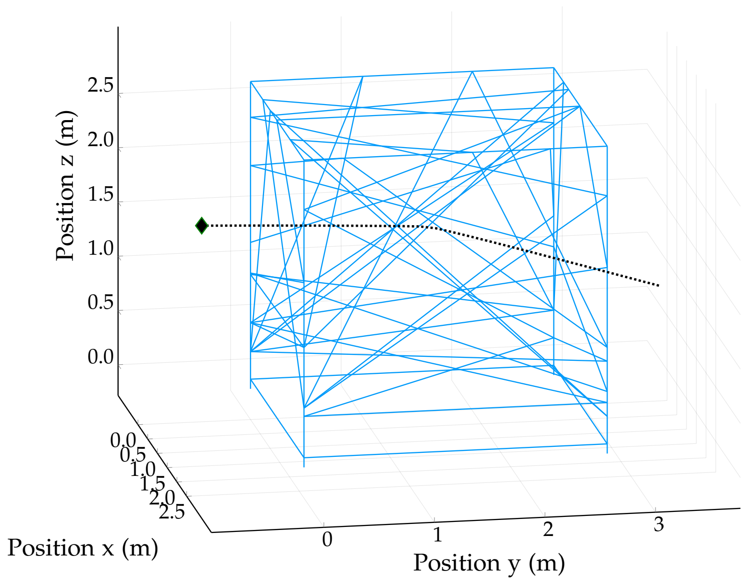


Figure 9.2: WireMaze scenario. UAV navigates from one side of maze to a the goal position on the opposite side while avoiding the wire obstacles.

current position and noisy observations of the wire obstacles. The noisy observations are generated by adding Gaussian noise with zero mean and standard deviation $\sigma_{obs}$ to the endpoints of each wire. Since the wires are straight and defined by its endpoints, adding noise to the endpoints adds variation to the entire wire. While the wire itself has no thickness, the planner considers a safety distance of $d_{safe}$ and we consider that a collision has occurred if the vehicle comes within $d_{collide}$ of the wire. The new plan becomes active after a time delay $\Delta_{plan}$, which is used to model the computation time of the planning process. The vehicle continues to follow the existing plan during the delay, i.e., while the plan is being computed. If the planner fails to produce a valid plan, then an error code is returned. The aircraft continues to follow the existing plan and another replan is attempted after $\Delta_{plan}$. The scenario ends successfully if the position of the aircraft comes within a distance $d_{goal}$ of the goal. Failures are defined as any one of the following: (1) the vehicle has collided with an obstacle defined as the vehicle's position being closer than $d_{collide}$ to an obstacle; (2) the planner has failed to produce a valid plan for three consecutive attempts; or (3) the planner has not reached the goal after $t_{max}$ seconds. A list of parameters and their values is shown in Table 9.1.

Table 9.1: AST and simulation parameters for WireMaze scenario

| **Name** | **Symbol** | **Value** |
|---|---|---|
| Maximum simulation time | $t_{max}$ | 35 s |
| Simulation time step | $\Delta_t$ | 1 s |
| Vehicle speed | $v_d$ | 0.15 m/s |
| Safety distance to obstacle | $d_{safe}$ | 0.15 m |
| Safety distance above ground | $h_{safe}$ | 1 m |
| RRT* number of samples | $N_{rrt}$ | 2000 |
| RRT* radius of steer function | $\eta_{rrt}$ | 0.10 |
| Planning delay | $\Delta_{plan}$ | 2 s (or disabled) |
| Replanning time interval | $\tau plan$ | 5 s (or disabled) |
| Wire observation noise standard deviation | $\sigma_{obs}$ | 3 cm (or disabled) |
| Vehicle Dynamics noise standard deviation | $\sigma_{pos}$ | 3 cm |
| Maximum planning tries before fail | $N_{fail}$ | 3 |
| Goal radius | $d_{goal}$ | 0.15 m |
| Collision radius | $d_{collide}$ | 0.05 m |
| Maximum number of smoothing iterations | $N_{smooth}$ | 50 |
| MCTS number of iterations | $N_{MCTS}$ | 100 |
| MCTS exploration constant | $c$ | 2 |
| MCTS action widening constant 1 | $k$ | 1 |
| MCTS action widening constant 2 | $\alpha$ | 0.25 |

### 9.4.3 Testing Dimensions

We stress tested the trajectory planning system in a number of experiments that varied several parameters of the simulation. Specifically, we experimented with all combinations of the following parameters. All experiments added Gaussian noise for vehicle motion.

**Observation Noise.** We considered scenarios with and without observation noise. In the case of no observation noise, the exact locations of the wire obstacles are always used for planning. In the case with observation noise, we added a small amount of zero-mean Gaussian noise with standard deviation $\sigma_{obs}$ to the endpoints of the wires. The noisy version of the wire obstacles is used for planning. A new noisy observation is sampled each time the planner is called.

**Replanning Condition.** We considered two alternative conditions for triggering replanning. The first triggers a replan at a fixed time interval $\tau_{plan}$. That is, the planner replans every $\tau_{plan}$ seconds regardless of its observations. The second triggers a replan whenever it is predicted that the current plan will lead to a collision in the future. The prediction is made based on the observation of the obstacles at the current time step.

**Scenario Configuration.** The scenario is defined by start and goal positions. We vary these positions by uniformly sampling start and goal positions within predefined volumes for each. These volumes are defined to be a short distance away from the entry and exit faces of the cube and at least the safety height

$h_{safe}$ above the ground, but no higher than the top of the maze.

**Planning Delay.** We considered scenarios with and without planning delay. For the case with no delay, the plans are generated instantaneously (in simulation time) and are executed immediately by the vehicle. For the case with delay, we considered a fixed planning delay $\Delta_{plan}$ of 2 seconds, which approximates the typical wall clock time of computing a plan for this implementation of the planner.

### 9.4.4 Results

We searched 10 scenario configurations under fixed-interval and collision replanning conditions, with and without observation noise, and with and without planning delay. Each search returns the 20 scenarios with the highest stress factor meaning that they obtain the highest reward in the AST search. Out of these 1600 obtained scenarios, 386 successfully reached the goal, 19 resulted in a collision, and 1195 terminated with a planning failure. Note that these failure scenarios are not purely random samples, but represent scenarios that AST found to be the most problematic. We present selected examples of failure scenarios and discuss their relevance.

**Collision with an Obstacle.** This scenario has Gaussian observation and dynamics noise, replanning based on a fixed time interval every five seconds, and instantaneous planning. Figure 9.3 shows a failure scenario where the vehicle collides with an obstacle. The vehicle plans successfully five times before a collision occurs at 24 seconds. This scenario shows that the effects of observation noise and dynamics noise can be additive to reduce the operational margin of the system and lead to a collision. Out of the 19 collisions that AST found, 18 collisions had the combination of observation and dynamics noise. Only one collision occurred under dynamics noise alone showing that the planner is quite robust to noise and only failing when multiple sources of noise conspire. Currently, the planner plans around obstacles to provide at least a safety distance $d_{safe}$, but does not try to maximize the margin away from multiple obstacles simultaneously. In the future, the planner may be improved to maximize the distance between multiple silhouettes, which would provide added margin for avoiding obstacles.

Figure 9.3 illustrates the vehicle's collision with an obstacle. The left plot shows a top down view of the WireMaze and the right plot shows a side profile of the maze as viewed from the right. Blue lines indicate the wire obstacles. The vehicle's start position is labeled *start*. The solid black line indicates the vehicle's actual traveled path and the dotted black lines indicate the trajectories returned by the planner. The green diamond markers indicate the times at which the planner computed a plan successfully and the asterisk marks the time of the collision.

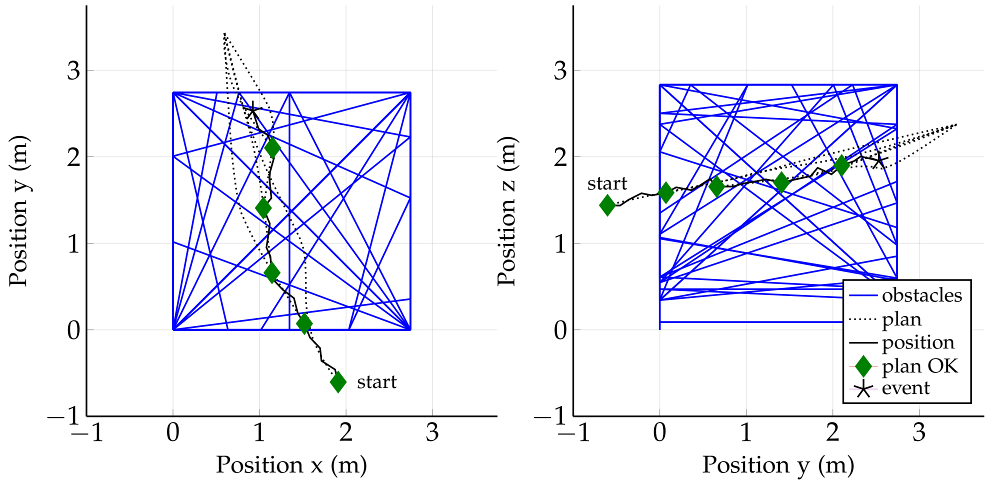


Figure 9.3: Failure scenario where the UAV collides with an obstacle under both observation and vehicle navigation noise. Collision is denoted by an asterisk.

**SIT Fails to Find a Path When Vehicle Is Inside $d_{safe}$ Region.** This scenario has Gaussian dynamics noise, no observation noise, fixed time interval replanning, and instantaneous planning. Figure 9.4 shows a failure scenario where SIT is unable to find a feasible path. The vehicle plans successfully two times before the planning failures begin. The planner attempts to compute a plan at 10 seconds but fails with error code 1 indicating that the SIT algorithm was unable to find a solution. The planner makes two more failed attempts in the next two time steps returning with the same error code. We terminate the scenario after 3 consecutive planning failures. The planning failures are indicated by red diamonds and the terminal event is indicated by a red square in the figure. On analysis, we found that the planning failure is caused by the position of the vehicle being within the planning safety distance $d_{safe}$ of the nearest wire obstacle. As a result, all the plans returned by SIT contain a collision and are not accepted. The dynamics noise has caused the vehicle to wander within $d_{safe}$ of an obstacle. It would be helpful for the planner to check for this condition, because the vehicle could still recover from this position and a valid trajectory could still be reached.

**SIT Fails to Find a Path After Exhausting Sampling Budget.** This scenario has Gaussian dynamics, no observation noise, five-second interval replanning, and instantaneous planning. Figure 9.5 shows a failure scenario where the tree search algorithm is unable to find a feasible path, but the vehicle position is outside of the $d_{safe}$ region. The vehicle plans successfully two times before the planning failures begin. We focus on the final planning failure indicated by a red square and error code 1, which indicates that the SIT was not able to find a feasible solution. The vehicle position is not within $d_{safe}$ as the previous example. The probability of the SIT algorithm finding a feasible solution tends towards 1 as the number of samples increases to infinity. However, for finite samples, the SIT algorithm may not find a solution

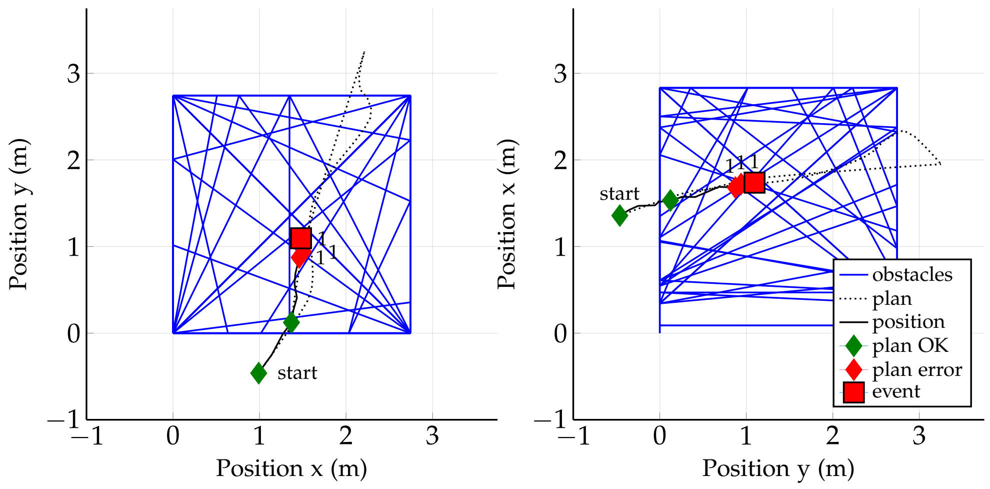


Figure 9.4: Failure scenario where the vehicle begins planning too close to an obstacle. All plans violate safety constraint and planning fails. Planner fails at three consecutive time steps indicated by the red markers.

as demonstrated in Figure 9.5. While it is unclear how many samples are required to guarantee a certain level of search performance, the probability of finding a solution decreases with fewer samples. Since the algorithm is being used online, the number of samples is constrained, so occasionally there may be cases where a solution is not found. In this example, the planner was not able to find a solution within the sampling budget. Interestingly, we see that a solution does exist in this case, since the existing plan can still be used.

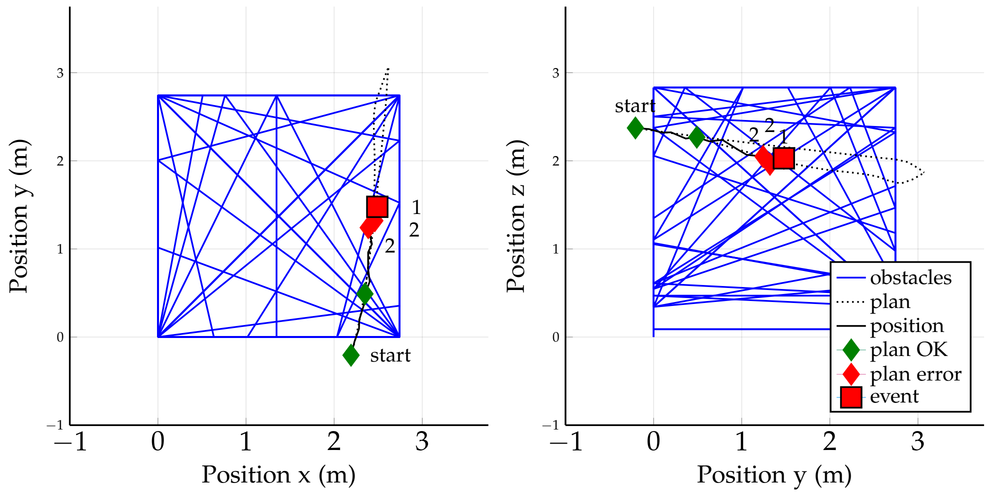


Figure 9.5: Failure scenario where SIT exhausts the sampling budget. SIT is a stochastic search algorithm that has non-zero probability of failing to find a solution within a finite sampling budget.

**Excessive Replanning.** This scenario has Gaussian dynamics noise, collision replanning, and instantaneous planning. Figure 9.6 shows a scenario where the vehicle replans if the current trajectory is predicted to collide with an obstacle given the current observation. The figure compares the scenarios with and without observation noise and illustrates the emergent behavior resulting from the interactions between the collision replanning condition and the observation noise. The top row of Figure 9.6 shows the case where there is no observation noise. The vehicle plans only once at the start of the scenario. Because the vehicle can observe the obstacles exactly, the condition for replanning is never activated and the vehicle follows the plan until reaching the goal. The bottom row of Figure 9.6 shows the case where there is Gaussian observation noise. In this case, the observation noise at each time step makes it seem like a collision will occur at some point in the future. As a result, the vehicle replans excessively, almost at every opportunity. This scenario also terminates with a planning failure where the SIT algorithm was not able to find a feasible solution. The reason is because the vehicle is within the $d_{safe}$ region for the three final time steps. This issue is more of a potential problem that can occur if the planner is to be combined with a replanning rule based on collision prediction since the current version of the planner is meant to be used in open-loop. Enhancing the planner to increase the margin around obstacles to account for the uncertainty may help reduce the number of replans as the increased margin can help offset the operational margin lost to observation noise.

**Failure of the Smoothing Stage to Find a Path.** This scenario has Gaussian observation and dynamics noise, fixed time interval replanning, and instantaneous planning. Figure 9.7 shows a failure scenario where the smoothing algorithm is unable to find a valid solution. The planner experiences three consecutive planning errors beginning at five seconds. The scenario terminates after the three planning failures. The error code 2 indicates that the failure occurred in the smoothing stage of the planner. That is, the algorithm found a piecewise linear solution. However, the planner was not able to find a $\mathcal{C}^2$ continuous solution that approximates it. Internally, the error is an infinite loop in the smoothing stage that is caught and returned with an error code. This type of planning error was the most prevalent in our testing.

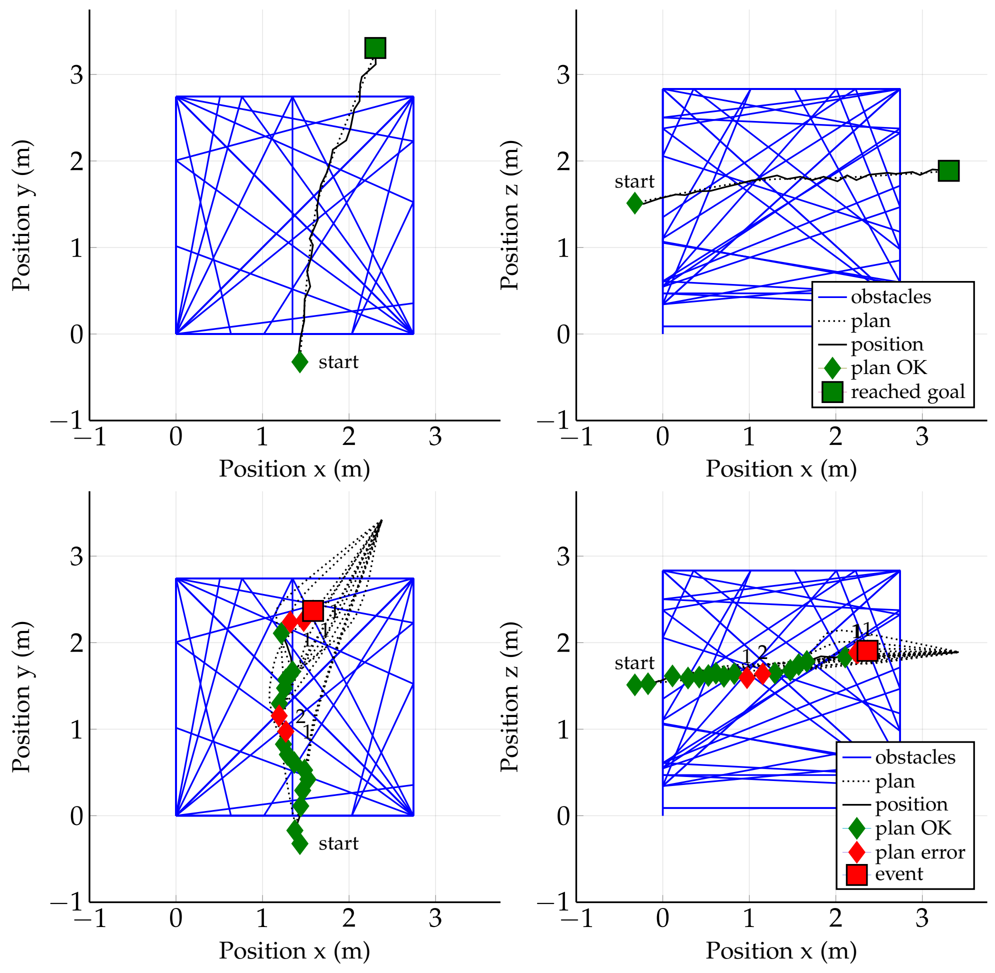


Figure 9.6: Planning without (top) and with (bottom) observation noise. Observation noise causes replanning condition to the triggered excessively resulting in excessive replanning.

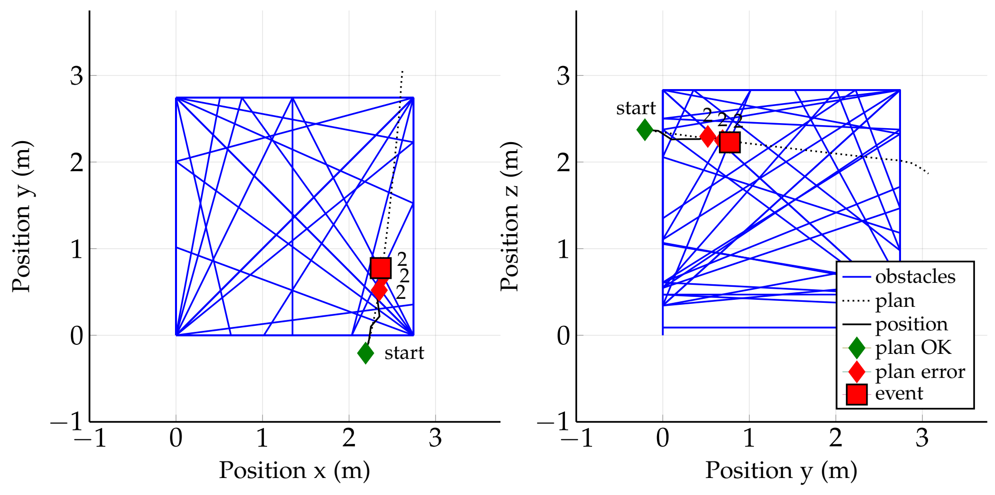


Figure 9.7: Failure scenario where the planner fails at the smoothing stage indicated by the error code 2. The smoothing stage is unable to find a $\mathcal{C}^2$ continuous path that approximates the piecewise solution returned by the tree search stage.

**Planning Failure when Near the Goal.** This scenario has Gaussian dynamics, no observation noise, interval replanning, and instantaneous planning. Figure 9.8 shows a failure scenario where the planner is unable to find a solution when the vehicle is very close to the goal position. The vehicle has traveled through the maze and has almost reached the goal. At 25 seconds, the vehicle is 0.94 meters from the goal and the planner is unable to find a solution twice with error code 1. At the final attempt, the vehicle is 0.65 meters from the goal. The SIT algorithm is able to find a solution, but the smoothing algorithm is not able to find a smooth approximation. The failure of the smoothing stage is not related to the obstacles as it has already cleared the maze. The problem seems to be that the implementation of the smoothing stage cannot handle trajectories that contain only a single path segment.

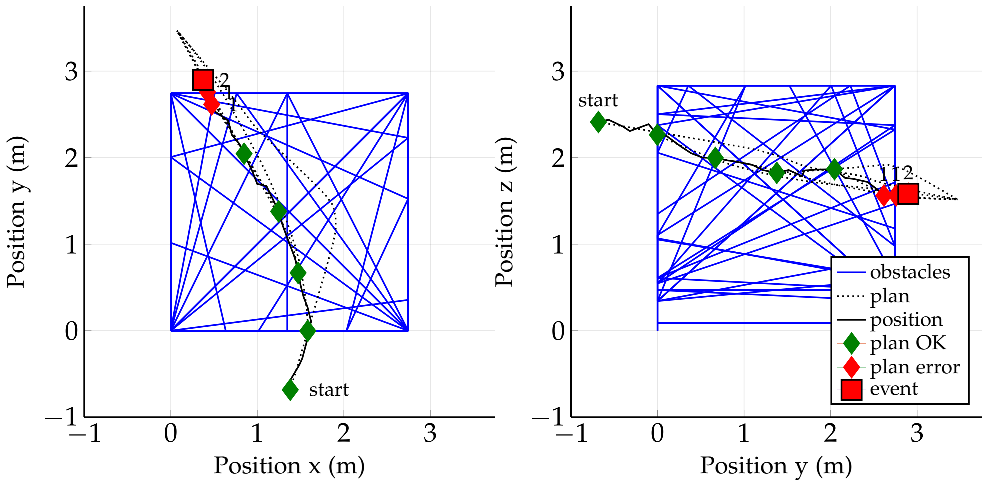


Figure 9.8: Planning fails when vehicle is too close to the goal. The implementation of the smoothing stage does not properly handle trajectories that contain only a single path segment.

**Planning Delays and Disjunct Paths.** This scenario has Gaussian obstacle and dynamics noise, and the vehicle replans only when a collision is predicted. Planning requires two seconds and the vehicle follows the existing plan during the computation. The vehicle starts following the new plan after the two-second delay. Replanning attempts following a planning failure also must wait for two seconds. Figure 9.9 shows a failure scenario where the vehicle travels a very disjunct trajectory due to the planning delays. We can see that the planner can produce very different paths on replan. Planning does not take into account the degree of deviation from the existing plan. Because of the two-second delay, the new plan may have already diverged from the existing path causing the vehicle's traveled path to jump as it switches to follow the new path. Control logic to blend the two plans and smoothen the transition would be helpful. Furthermore, as we discussed previously, the collision-based replanning strategy can lead to excessive planning under noisy observations of the obstacles. When combined with the delay, the result is a very disjunct vehicle path. This issue is another one that anticipates possible closed-loop operation of the planner, which is currently meant for open-loop use. It highlights a potential issue in the interaction between collision replan and computational delays, and motivates the addition of path blending logic. Enhancing the planner to increase margin between obstacles to account for added noise as discussed earlier may also help.

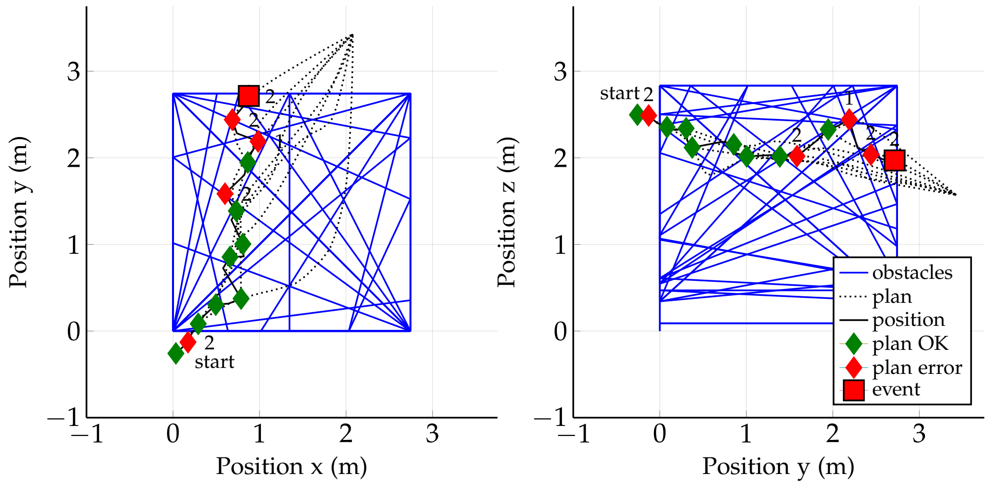


Figure 9.9: Disjunct behavior results from interactions between planning delay and collision replanning. Planner does not take into account amount of deviation from current plan and thus can produce significantly different plans.

**Failure in Evaluating Points.** This scenario has Gaussian dynamics noise, no observation noise, and replans every five seconds. Plans take two seconds to compute and take effect. Figure 9.10 shows a failure scenario involving the final stage of the planner where temporal specifications are incorporated and the final waypoints are evaluated along the continuous trajectory. In the figure, the planning failure is shown as error code 3 at seven seconds. The planning failure of the final stage seems to be related to an implementation issue where the end time of the trajectory is returned as an invalid number.

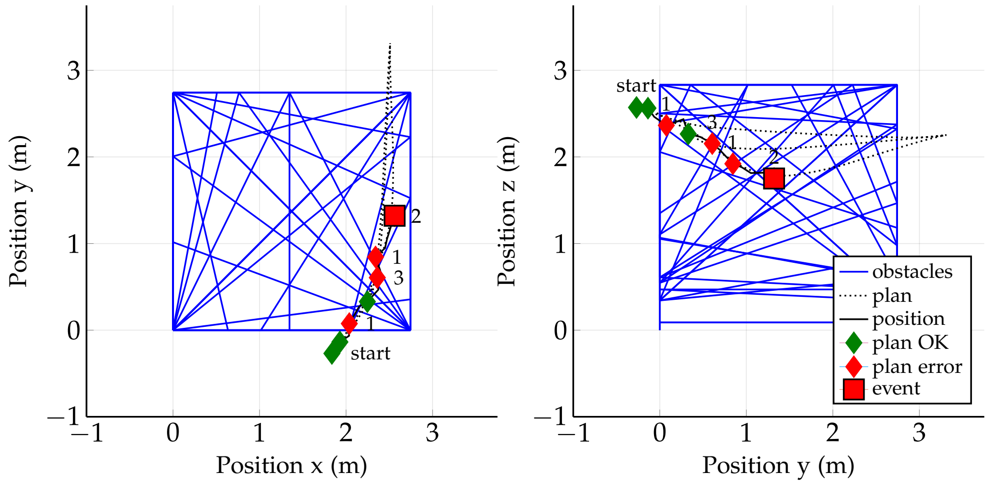


Figure 9.10: Failure scenario where the planner fails in the final point evaluation stage (indicated by error code 3) due to a implementation issue.

## 9.5 Summary

This chapter discussed how the AST framework can be applied to test the runtime behavior of trajectory planning systems. We presented a generic system architecture and discussed sources of uncertainty, which include imperfect sensors, changing environment, vehicle motion, and computation time of the planner. We also discussed possible failure modes of trajectory planning systems, which include vehicle failed states, replanning issues, unable to find a plan, and implementation issues. Subsequently, we presented a study that applies AST to analyze failure scenarios in a prototype trajectory planning system navigating through a three-dimensional maze of wires. Using AST, we found 1214 failure scenarios out of 1600 tested, and identified 8 categories of issues, including collisions with obstacles, failures when originating a search from within a safe region, planner exhausting the sampling budget, excessive replanning, unable to find a smooth approximation, failure when planning too close to the goal, disjunct paths due to planning delays, and failures at the point evaluation stage. Table 9.2 summarizes the failure categories identified in our analysis of the ATTRACTOR trajectory planning system.

These issues were reported to the ATTRACTOR team and future versions of the trajectory planning system will incorporate changes to address these issues. AST continues to support the ATTRACTOR project and AST will be applied to stress test future versions of the trajectory planner as they become available.

| ID | Failure Category | Observation | Cause |
|---|---|---|---|
| 1 | Collision with an obstacle | Vehicle closer than $d_{collide}$ from nearest wire | Sensor and vehicle navigation noises combine to reduce margin |
| 2 | SIT fails to find a path when vehicle is inside $d_{safe}$ region | Planning failure at stage 1 (SIT) | Vehicle state is within $d_{safe}$ of obstacle (but not $d_{collide}$), all plans fail safety check |
| 3 | SIT fails to find a path after exhausting sampling | Planning failure at stage 1 (SIT) | SIT is a stochastic search algorithm that is not guaranteed to find a solution within finite samples |
| 4 | Excessive replanning | Planner replans at every time step | Replanning condition is triggered excessively due to sensor noise |
| 5 | Failure of the smoothing stage to find a path | Planning failure at stage 2 (smoothing) | Smoothing algorithm can fail to find a viable $\mathcal{C}^2$ approximation of SIT solution |
| 6 | Planning failure when near the goal | Planning failure in stage 2 (smoothing) when vehicle is near the goal | Smoothing algorithm implementation does not properly handle single segment paths |
| 7 | Planning delays and disjunct paths | Vehicle path is disjunct and unsmooth | Interaction between planning delay and collision-based replanning condition |
| 8 | Failure to evaluate points | Planning failure in stage 3 (evaluation) | Software bug |

Table 9.2: Summary of failure categories discovered in ATTRACTOR trajectory planner using AST

# Chapter 10

# Summary and Further Work

In this dissertation, we presented AdaStress, a collection of novel algorithms for analyzing failure events in cyber-physical systems. We identified several key challenges in the failure analysis of safety-critical systems, including large and complex state spaces, interactions with large environments, operation over time, black boxes and hidden states, rare failures, heterogeneous multivariate time series data, and explaining failures from large datasets. We proposed solutions to address these challenges and demonstrated them on an aircraft collision avoidance system and a trajectory planning system. In this chapter, we conclude the thesis, highlight the contributions made, and outline several areas of further work.

## 10.1 Summary

In Chapter 5, we proposed adaptive stress testing (AST), a novel framework for stress testing safety-critical systems. AST poses the problem of finding the most likely failure scenario as a sequential decision-making problem, and then uses reinforcement learning algorithms to optimize it. For systems where the full system state is available, AST can be applied with existing reinforcement learning algorithms. For systems with hidden state, we propose the seed-action simulator abstraction where the pseudorandom seed of the simulator is used to control the stochastic evolution of the simulator. We present an optimization algorithm for searching seed-action simulators, called Monte Carlo tree search for seed-action simulators (MCTS-SA), that is based on Monte Carlo tree search with double progressive widening (MCTS-DPW). The seed-action abstraction requires very few internal details to be exposed from the components, which makes it very easy to apply to new problems.

In Chapter 6, we proposed differential adaptive stress testing (DAST), a novel framework for stress testing relative to a baseline system. The goal of DAST is to find failure scenarios that occur with the system under test, but not in the baseline system under the same circumstances. DAST runs two parallel instances of the simulator, one running the system under test and the other running the baseline system, and uses AST to drive them to different outcomes. Differential analysis can be useful for comparing two candidate solutions or regression testing where a newer version is compared to a previous version to see whether new failures have been introduced.

In Chapter 7, we proposed grammar-based decision tree (GBDT), a novel interpretable machine model for classification and categorization of multivariate heterogeneous time series data. GBDT generalizes a decision tree to support Boolean expressions derived from a context-free grammar (CFG). By choosing a CFG that produces temporal logic expressions, GBDT can model multivariate heterogeneous time series data. We proposed a learning algorithm for GBDT that uses expression optimization algorithms as a subroutine and demonstrated that GBDT can not only produce good classification performance, but also intuitive explanations in analyzing the Australian Sign Language dataset.

In Chapter 8, we applied the algorithms in AdaStress to analyze near mid-air collisions (NMACs) in the next-generation Airborne Collision Avoidance System (ACAS X). We created a seed-action simulator modeling aircraft mid-air encounters and searched for the most likely NMAC scenarios using AST. We found and analyzed various categories of NMAC for single-threat (two-aircraft) and multi-threat (three-aircraft) encounters. Subsequently, we applied DAST to compare the NMAC behavior of ACAS X relative to the existing Traffic Alert and Collision Avoidance System (TCAS). With DAST, we considered both (1) ACAS X relative to TCAS and (2) TCAS relative to ACAS X and discussed the most likely relative NMAC scenarios for each system. Overall, we found more NMAC scenarios with TCAS as the system under test, which gives confidence that ACAS X provides a safety benefit over TCAS. We applied GBDT to analyze a larger dataset of encounters to automatically discover categories of NMACs. For this encounter dataset, GBDT found and explained six natural categories of NMACs. Lastly, we evaluated the performance of AST using MCTS-SA on single-threat encounters and compared the performance to direct Monte Carlo algorithm. We found that MCTS-SA was significantly more effective at finding NMACs than direct Monte Carlo.

In Chapter 9, we showed how AST can be applied to stress test trajectory planning systems and we demonstrated the approach on a prototype trajectory planning system in a wire maze environment from the Autonomy Teaming & TRAjectories for Complex Trusted Operational Reliability (ATTRACTOR) project. By considering such factors as observation noise, replanning condition, scenario configuration, planning delay, and vehicle navigation uncertainty, AST discovered a variety of collisions and planning failures.

## 10.2 Contributions

The first part of this thesis provides a broad introduction to testing and validation approaches (Chapter 2), sequential decision processes and reinforcement learning (Chapter 3), and interpretable and explainable machine learning (Chapter 4). These background chapters do not advance any new ideas, but they do provide a valuable review of existing work from different research communities.

The remainder of this thesis makes the following contributions:

- **A novel approach to finding failure scenarios using sequential decision-making.** We introduce adaptive stress testing (AST), a framework that formulates finding the most likely failure scenario as a sequential decision-making problem, which allows reinforcement learning algorithms to be applied to optimize it (Section 5.2). A key component of this framework is the AST reward function (Eq. 5.1), which finds failure events and maximizes the path probability. We propose a novel simulation abstraction based on controlling the pseudorandom seed, called seed-action simulator (Section 5.4.1) and an optimization algorithm for it, called MCTS-SA (Section 5.4.3), which enables effective stress testing of simulators with hidden state.
- **A novel approach to differential analysis of failure behaviors between two systems.** Some applications are interested in analyzing failure behavior relative to a another system. Chapter 6 introduces DAST, a stress testing approach that finds the most likely failure scenarios that occur in one system but not in another baseline system. A key component of this framework is the DAST reward function (Eq. 6.1), which compares the output of two simulators to find differences in failure behavior while maximizing the path probability.
- **A novel interpretable machine learning approach for multivariate heterogeneous time series data.** Chapter 7 discusses GBDT, a novel interpretable machine learning model for multivariate heterogeneous time series data using expressions from a context-free grammar. In a GBDT model, the splitting criteria in the decision tree nodes can be more complex, which allows a more abstract and compact representation. The model can be used for classification as well as categorization and generating explanations (Section 7.2.2). We propose a novel learning algorithm for the GBDT model based on expression optimization (Section 7.3).
- **A characterization of ACAS X NMAC scenarios in support of a Federal Aviation Administration (FAA) program.** Our analyses of ACAS X have contributed to the verification and validation and safety case for the certification of ACAS X. The RTCA accepted ACAS X as the new standard for airborne collision avoidance on September $20^{th}$, 2018, and the system is expected to be widely deployed in the near future. We identified NMAC scenarios in single-threat encounters (Sec-

tion 8.3.2) and multi-threat encounters (Section 8.3.3). We also identified the most likely scenarios where ACAS X fails but TCAS does not (Section 8.4.1), and vice versa (Section 8.4.2). We also automatically identified the key failure categories in a large encounter dataset and explained their most relevant properities (Section 8.5).

- **A framework for finding failures in trajectory planning systems.** Chapter 9 describes a generic framework for adaptive stress testing of trajectory planning systems (Section 9.3). The successful adaptation of AST to the trajectory planning setting demonstrates the generality and broad applicability of AST framework. Our characterization of the failure scenarios in the silhouette-informed tree (SIT) trajectory planning system is important to the ATTRACTOR project and its final demonstration taking place in 2020 (Section 9.4).
- **An implemented system for testing safety-critical systems and categorizing failure scenarios.** We provide open source implementations of the algorithms described in this thesis for stress testing and automatic categorization. The software is available as an open source Julia package AdaStress.jl available at `https://github.com/rcnlee/AdaStress.jl`.

## 10.3 Further Work

In this final section of the thesis, we discuss a few promising areas of work that can further advance this research and the state-of-the-art.

**Apply AdaStress to Other Domains.** In this thesis, we have used AST to analyze two applications in aviation. However, the AST framework is very general and in no way limited to the aviation domain. Indeed, concurrent work has applied AST to analyze autonomous cars at pedestrian crossings [76]. We believe there are many opportunities for AST to contribute in other application domains as well, including autonomous cars, ships, spacecraft, interplanetary rovers, and smart infrastructure.

**Scalable Algorithms with Formal Guarantees.** We have shown that sequential decision-making algorithms can be used to scale the search for failure scenarios in large and complex cyber-physical systems. While formal verification methods have difficulty scaling to large systems, they can provide formal guarantees that simulation testing cannot. A promising research direction would be to integrate ideas from formal verification and adaptive stress testing to create scalable validation algorithms with formal guarantees.

**Integrating Stress Testing and Categorization.** At present, AST is used to find failure scenarios and then GBDT is applied to categorize them. A promising research direction would be to close the loop by using the results of GBDT categorization as feedback to inform and improve the AST search. AST could use the information to search more efficiently or to find a more diverse set of NMACs.

**Safe Autonomy that is Easy to Validate.** AdaStress, like other existing approaches to validation, assumes that the system to be validated is fixed. However, development and validation are not isolated activities. How a system is designed greatly impacts how difficult it is to validate. A promising research direction would be to improve validation through exploration of system designs that are both safer and easier to validate. A further goal would also add interpretability, so that the systems could also be easily checked and understood by a human expert.

# Appendix A

# ACAS X Grammar

$$
\begin{aligned}
b &::= \mathit{always} \mid \mathit{eventually} \mid \mathit{implies} \mid \mathit{count}\\
\mathit{always} &::= \mathrm{G}(\vec{b})\\
\mathit{eventually} &::= \mathrm{F}(\vec{b})\\
\mathit{implies} &::= \mathrm{G}(\vec{b} \implies \vec{b})\\
\mathit{count} &::= (\textsc{Count}(\vec{b}) < \mathit{timestep}) \mid (\textsc{Count}(\vec{b}) \leq \mathit{timestep}) \mid (\textsc{Count}(\vec{b}) > \mathit{timestep})\\
\mathit{count} &::= \textsc{Count}(\vec{b}) \geq \mathit{timestep}) \mid (\textsc{Count}(\vec{b}) = \mathit{timestep})\\
\vec{b} &::= \mathit{bin\_feat} \mid \mathit{and} \mid \mathit{or} \mid \mathit{not} \mid \mathit{eq} \mid \mathit{lt} \mid \mathit{lte} \mid \mathit{abseq} \mid \mathit{abslt} \mid \mathit{abslte} \mid \mathit{sign}\\
\vec{b} &::= \mathit{absdiff\_eq} \mid \mathit{absdiff\_lt} \mid \mathit{absdiff\_lte}\\
\mathit{and} &::= \vec{b} \wedge \vec{b}\\
\mathit{or} &::= \vec{b} \vee \vec{b}\\
\mathit{not} &::= \neg\vec{b}\\
\mathit{eq} &::= \mathit{vrate\_eq} \mid \mathit{chi\_angle\_eq} \mid \mathit{psi\_angle\_eq} \mid \mathit{sr\_eq} \mid \mathit{psid\_eq} \mid \mathit{v\_eq} \mid \mathit{alt\_eq} \mid \mathit{abs\_altdiff\_eq}\\
\mathit{vrate\_eq} &::= (\mathit{vrate\_feat} = \mathit{vrate\_val}) \mid (\mathit{vrate\_feat} = \mathit{vrate\_feat})\\
\mathit{chi\_angle\_eq} &::= (\mathit{chi\_angle\_feat} = \mathit{angle\_val}) \mid (\mathit{chi\_angle\_feat} = \mathit{chi\_angle\_feat})\\
\mathit{psi\_angle\_eq} &::= (\mathit{psi\_angle\_feat} = \mathit{psi\_angle\_feat})\\
\mathit{sr\_eq} &::= (\mathit{sr\_feat} = \mathit{sr\_val})\\
\mathit{psid\_eq} &::= (\mathit{psid\_feat} = \mathit{psid\_val}) \mid (\mathit{psid\_feat} = \mathit{psid\_feat})\\
\mathit{v\_eq} &::= (\mathit{v\_feat} = \mathit{v\_val}) \mid (\mathit{v\_feat} = \mathit{v\_feat})\\
\mathit{alt\_eq} &::= (\mathit{alt\_feat} = \mathit{alt\_val}) \mid (\mathit{alt\_feat} = \mathit{alt\_feat})\\
\mathit{abs\_altdiff\_eq} &::= (\mathit{abs\_altdiff\_feat} = \mathit{abs\_altdiff\_val})\\
\mathit{abseq} &::= \mathit{vrate\_abseq} \mid \mathit{altdiff\_abseq} \mid \mathit{chi\_angle\_abseq} \mid \mathit{psi\_angle\_abseq} \mid \mathit{sr\_abseq} \mid \mathit{psid\_abseq}\\
\mathit{abseq} &::= \mathit{v\_abseq} \mid \mathit{alt\_abseq}\\
\mathit{vrate\_abseq} &::= (|\mathit{vrate\_feat}| = |\mathit{vrate\_val}|) \mid (|\mathit{vrate\_feat}| = |\mathit{vrate\_feat}|)\\
\mathit{altdiff\_abseq} &::= (|\mathit{altdiff\_feat}| = |\mathit{altdiff\_val}|)
\end{aligned}
$$

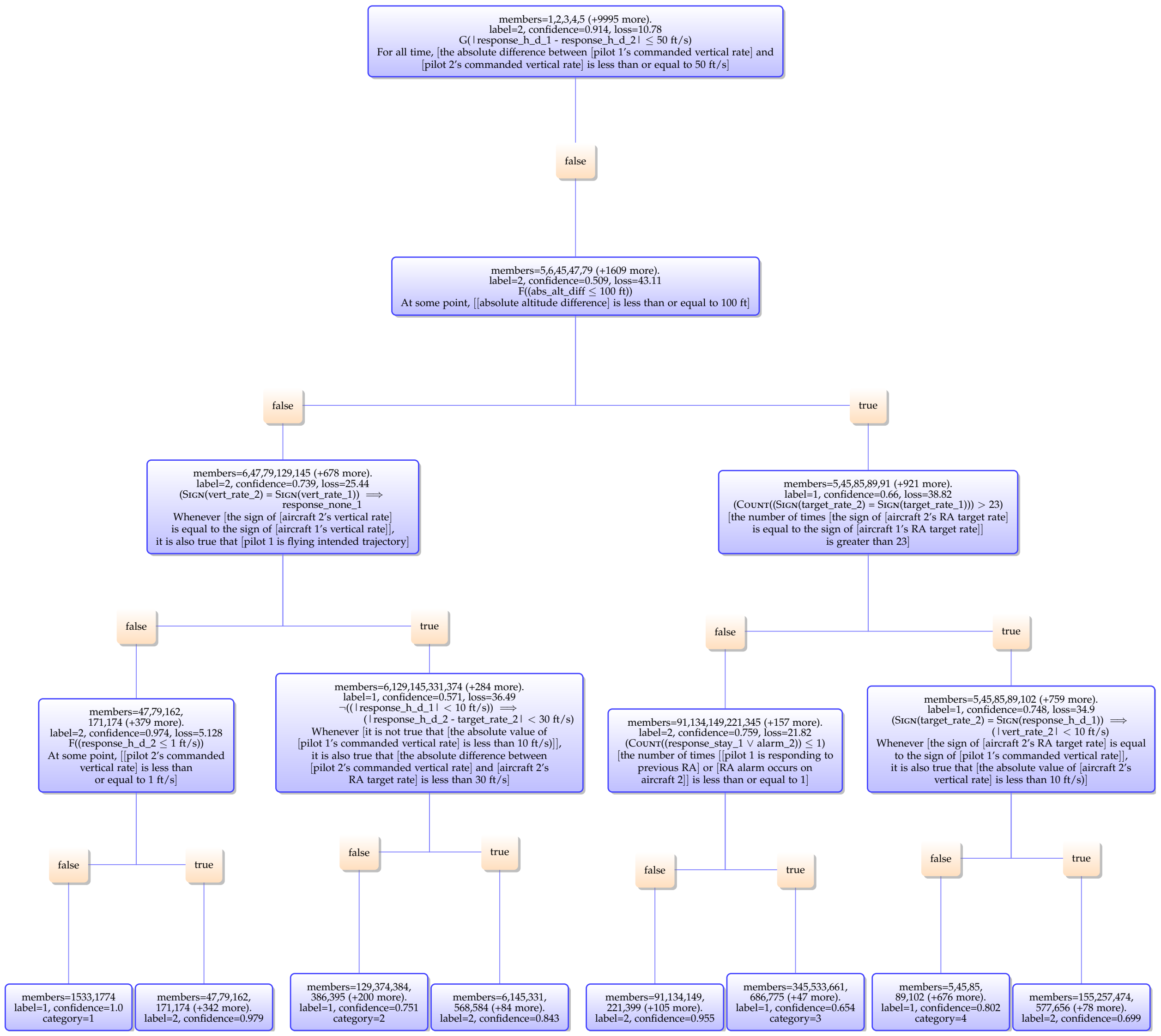
$$
\begin{aligned}
chi\_angle\_abseq &::= (|chi\_angle\_feat| = |angle\_val|) \mid (|chi\_angle\_feat| = |chi\_angle\_feat|)\\
psi\_angle\_abseq &::= (|psi\_angle\_feat| = |psi\_angle\_feat|)\\
sr\_abseq &::= (|sr\_feat| = |sr\_val|)\\
psid\_abseq &::= (|psid\_feat| = |psid\_val|) \mid (|psid\_feat| = |psid\_feat|)\\
v\_abseq &::= (|v\_feat, | = |v\_val|) \mid (|v\_feat| = |v\_feat|)\\
alt\_abseq &::= (|alt\_feat| = |alt\_val|) \mid (|alt\_feat| = |alt\_feat|)\\
lt &::= vrate\_lt \mid chi\_angle\_lt \mid psi\_angle\_lt \mid sr\_lt \mid psid\_lt \mid v\_lt \mid alt\_lt \mid abs\_altdiff\_lt\\
vrate_lt &::= (vrate\_feat < vrate\_val) \mid (vrate\_feat < vrate\_feat)\\
chi\_angle\_lt &::= (chi\_angle\_feat < angle\_val) \mid (chi\_angle\_feat < chi\_angle\_feat)\\
psi\_angle\_lt &::= (psi\_angle\_feat < psi\_angle\_feat)\\
sr\_lt &::= (sr\_feat < sr\_val)\\
psid\_lt &::= (psid\_feat < psid\_val) \mid (psid\_feat < psid\_feat)\\
v\_lt &::= (v\_feat < v\_val) \mid (v\_feat < v\_feat)\\
alt\_lt &::= (alt\_feat < alt\_val) \mid (alt\_feat < alt\_feat)\\
abs\_altdiff\_lt &::= (abs\_altdiff\_feat < abs\_altdiff\_val)\\
abslt &::= vrate\_abslt \mid altdiff\_abslt \mid chi\_angle\_abslt \mid psi\_angle\_abslt \mid sr\_abslt \mid psid\_abslt\\
abslt &::= v\_abslt \mid alt\_abslt\\
vrate\_abslt &::= (|vrate\_feat| < vrate\_val) \mid (|vrate\_feat| < vrate\_feat)\\
altdiff\_abslt &::= (|altdiff\_feat| < altdiff\_val)\\
chi\_angle\_abslt &::= (|chi\_angle\_feat| < angle\_val) \mid (|chi\_angle\_feat| < chi\_angle\_feat)\\
psi\_angle\_abslt &::= (|psi\_angle\_feat| < psi\_angle\_feat)\\
sr\_abslt &::= (|sr\_feat| < sr\_val)\\
psid\_abslt &::= (|psid\_feat| < psid\_val) \mid (|psid\_feat| < psid\_feat)\\
v\_abslt &::= (|v\_feat| < v\_val) \mid (|v\_feat| < v\_feat)\\
alt\_abslt &::= (|alt\_feat| < alt\_val) \mid (|alt\_feat| < alt\_feat)\\
lte &::= vrate\_lte \mid chi\_angle\_lte \mid psi\_angle\_lte \mid sr\_lte \mid psid\_lte \mid v\_lte \mid alt\_lte \mid abs\_altdiff\_lte\\
vrate\_lte &::= (vrate\_feat \leq vrate\_val) \mid (vrate\_feat \leq vrate\_feat)\\
chi\_angle\_lte &::= (chi\_angle\_feat \leq angle\_val) \mid (chi\_angle\_feat \leq chi\_angle\_feat)\\
psi\_angle\_lte &::= (psi\_angle\_feat \leq psi\_angle\_feat)\\
sr\_lte &::= (sr\_feat \leq sr\_val)\\
psid\_lte &::= (psid\_feat \leq psid\_val) \mid (psid\_feat \leq psid\_feat)\\
v\_lte &::= (v\_feat \leq v\_val) \mid (v\_feat \leq v\_feat)\\
alt\_lte &::= (alt\_feat \leq alt\_val) \mid (alt\_feat \leq alt\_feat)\\
abs\_altdiff\_lte &::= (abs\_altdiff\_feat \leq abs\_altdiff\_val)\\
abslte &::= vrate\_abslte \mid altdiff\_abslte \mid chi\_angle\_abslte \mid psi\_angle\_abslte \mid sr\_abslte \mid psid\_abslte\\
abslte &::= v\_abslte \mid alt\_abslte\\
vrate\_abslte &::= (|vrate\_feat| \leq vrate\_val) \mid (|vrate\_feat| \leq vrate\_feat)\\
altdiff\_abslte &::= (|altdiff\_feat| \leq altdiff\_val)\\
chi\_angle\_abslte &::= (|chi\_angle\_feat| \leq angle\_val) \mid (|chi\_angle\_feat| \leq chi\_angle\_feat)\\
psi\_angle\_abslte &::= (|psi\_angle\_feat| \leq psi\_angle\_feat)\\
sr\_abslte &::= (|sr\_feat| \leq sr\_val)\\
psid\_abslte &::= (|psid\_feat| \leq psid\_val) \mid (|psid\_feat| \leq psid\_feat)\\
v\_abslte &::= (|v\_feat| \leq v\_val) \mid (|v\_feat| \leq v\_feat)\\
alt\_abslte &::= (|alt\_feat| \leq alt\_val) \mid (|alt\_feat| \leq alt\_feat)
\end{aligned}
$$

$$\begin{aligned}
\mathit{sign} &::= \mathit{vrate\_sign} \mid \mathit{chi\_angle\_sign} \mid \mathit{psid\_sign} \\
\mathit{vrate\_sign} &::= (\textsc{Sign}(\mathit{vrate\_feat}) = \textsc{Sign}(\mathit{vrate\_feat})) \\
\mathit{chi\_angle\_sign} &::= (\textsc{Sign}(\mathit{chi\_angle\_feat}) = \textsc{Sign}(\mathit{chi\_angle\_feat})) \\
\mathit{psid\_sign} &::= (\textsc{Sign}(\mathit{psid\_feat}) = \textsc{Sign}(\mathit{psid\_feat})) \\
\mathit{absdiff\_eq} &::= \mathit{vrate\_absdiff\_eq} \mid \mathit{psi\_angle\_absdiff\_eq} \mid \mathit{psid\_absdiff\_eq} \mid \mathit{v\_absdiff\_eq} \\
\mathit{vrate\_absdiff\_eq} &::= (|\mathit{vrate\_feat} - \mathit{vrate\_feat}| = \mathit{vrate\_val}) \\
\mathit{psi\_angle\_absdiff\_eq} &::= (|\mathit{psi\_angle\_feat} - \mathit{psi\_angle\_feat}| = \mathit{angle\_val}) \\
\mathit{psid\_absdiff\_eq} &::= (|\mathit{psid\_feat} - \mathit{psid\_feat}| = \mathit{psid\_val}) \\
\mathit{v\_absdiff\_eq} &::= (|\mathit{v\_feat} - \mathit{v\_feat}| = \mathit{v\_val}) \\
\mathit{absdiff\_lt} &::= \mathit{vrate\_absdiff\_lt} \mid \mathit{psi\_angle\_absdiff\_lt} \mid \mathit{psid\_absdiff\_lt} \mid \mathit{v\_absdiff\_lt} \\
\mathit{vrate\_absdiff\_lt} &::= (|\mathit{vrate\_feat} - \mathit{vrate\_feat}| < \mathit{vrate\_val}) \\
\mathit{psi\_angle\_absdiff\_lt} &::= (|\mathit{psi\_angle\_feat} - \mathit{psi\_angle\_feat}| < \mathit{angle\_val}) \\
\mathit{psid\_absdiff\_lt} &::= (|\mathit{psid\_feat} - \mathit{psid\_feat}| < \mathit{psid\_val}) \\
\mathit{v\_absdiff\_lt} &::= (|\mathit{v\_feat} - \mathit{v\_feat}| < \mathit{v\_val}) \\
\mathit{absdiff\_lte} &::= \mathit{vrate\_absdiff\_lte} \mid \mathit{psi\_angle\_absdiff\_lte} \mid \mathit{psid\_absdiff\_lte} \mid \mathit{v\_absdiff\_lte} \\
\mathit{vrate\_absdiff\_lte} &::= (|\mathit{vrate\_feat} - \mathit{vrate\_feat}| \leq \mathit{vrate\_val}) \\
\mathit{psi\_angle\_absdiff\_lte} &::= (|\mathit{psi\_angle\_feat} - \mathit{psi\_angle\_feat}| \leq \mathit{angle\_val}) \\
\mathit{psid\_absdiff\_lte} &::= (|\mathit{psid\_feat} - \mathit{psid\_feat}| \leq \mathit{psid\_val}) \\
\mathit{v\_absdiff\_lte} &::= (|\mathit{v\_feat} - \mathit{v\_feat}| \leq \mathit{v\_val}) \\
\mathit{bin\_feat} &::= X[\mathit{bin\_feat\_id}] \\
\mathit{vrate\_feat} &::= X[\mathit{vrate\_feat\_id}] \\
\mathit{altdiff\_feat} &::= X[\mathit{altdiff\_feat\_id}] \\
\mathit{abs\_altdiff\_feat} &::= X[\mathit{abs\_altdiff\_feat\_id}] \\
\mathit{angle\_feat} &::= X[\mathit{angle\_feat\_id}] \\
\mathit{psi\_angle\_feat} &::= X[\mathit{psi\_angle\_feat\_id}] \\
\mathit{chi\_angle\_feat} &::= X[\mathit{chi\_angle\_feat\_id}] \\
\mathit{sr\_feat} &::= X[\mathit{sr\_feat\_id}] \\
\mathit{psid\_feat} &::= X[\mathit{psid\_feat\_id}] \\
\mathit{v\_feat} &::= X[\mathit{v\_feat\_id}] \\
\mathit{alt\_feat} &::= X[\mathit{alt\_feat\_id}] \\
\mathit{bin\_feat\_id} &::= 1 \mid 23 \mid 24 \mid 25 \mid 30 \mid 31 \mid 32 \mid 38 \mid 60 \mid 61 \\
\mathit{bin\_feat\_id} &::= 62 \mid 67 \mid 68 \mid 69 \mid 75 \\
\mathit{vrate\_feat\_id} &::= 2 \mid 22 \mid 34 \mid 39 \mid 59 \mid 71 \\
\mathit{altdiff\_feat\_id} &::= 3 \mid 40 \\
\mathit{abs\_altdiff\_feat\_id} &::= 76 \\
\mathit{angle\_feat\_id} &::= \mathit{psi\_angle\_feat\_id} \mid \mathit{chi\_angle\_feat\_id} \\
\mathit{psi\_angle\_feat\_id} &::= 4 \mid 41 \\
\mathit{chi\_angle\_feat\_id} &::= 6 \mid 43 \\
\mathit{sr\_feat\_id} &::= 5 \mid 42 \mid 77 \\
\mathit{timer\_feat\_id} &::= 33 \mid 70 \\
\mathit{psid\_feat\_id} &::= 35 \mid 72 \\
\mathit{v\_feat\_id} &::= 36 \mid 73 \\
\mathit{alt\_feat\_id} &::= 37 \mid 74
\end{aligned}$$

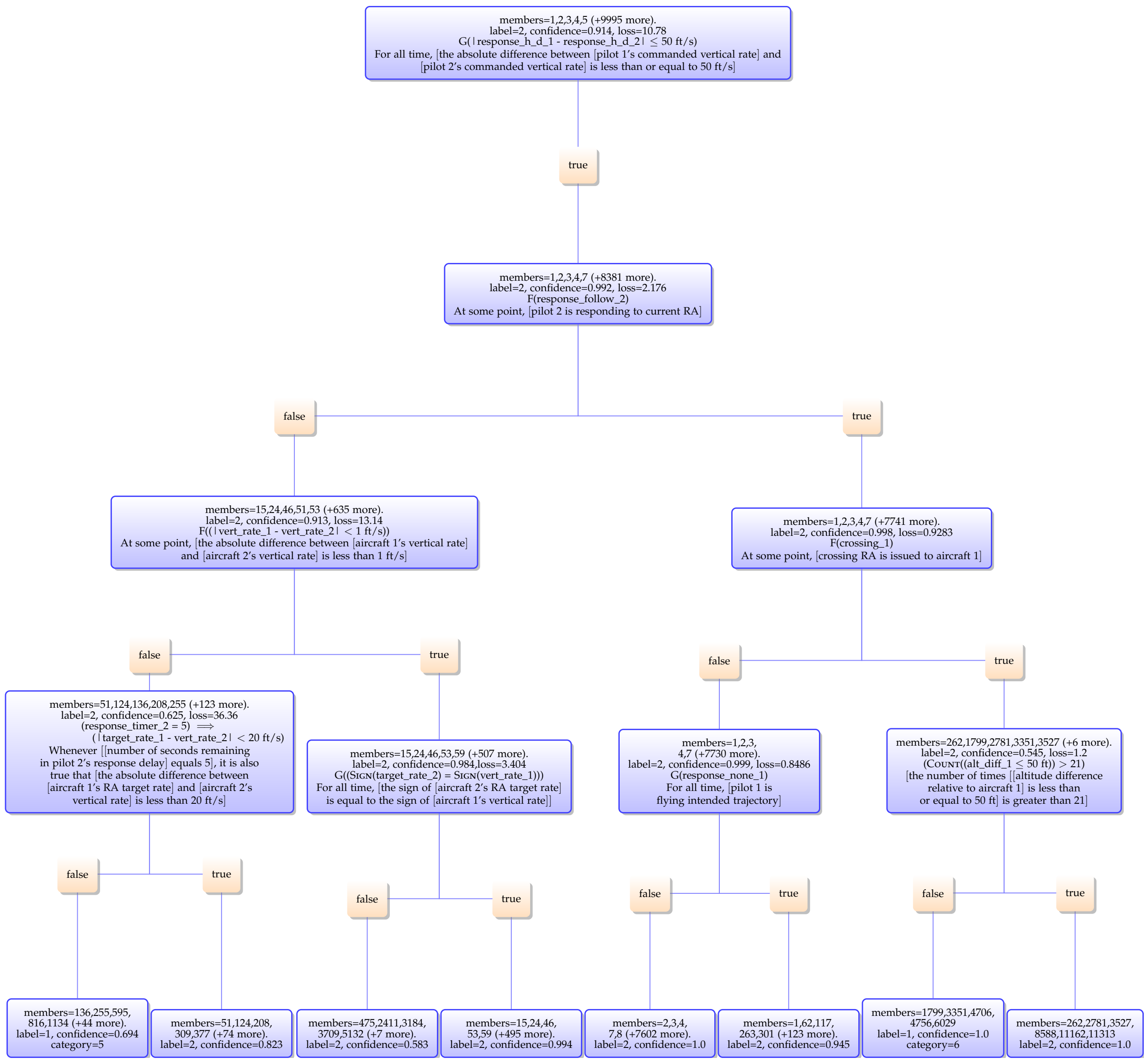
$$
\begin{aligned}
vrate\_val &::= -50 \mid -45 \mid -40 \mid -35 \mid -30 \mid -25 \mid -20 \mid -15 \mid -10 \mid -5 \mid -1 \mid 0 \mid 1 \mid 5 \mid 10 \mid 15 \\
vrate\_val &::= 20 \mid 25 \mid 30 \mid 35 \mid 40 \mid 45 \mid 50 \\
altdiff\_val &::= -2000 \mid -1500 \mid -1000 \mid -500 \mid -250 \mid -200 \mid -150 \mid -100 \mid -50 \mid -25 \mid -10 \\
altdiff\_val &::= -5 \mid 0 \mid 5 \mid 10 \mid 25 \mid 50 \mid 100 \mid 150 \mid 200 \mid 250 \mid 500 \mid 1000 \mid 1500 \mid 2000 \\
abs\_altdiff\_val &::= 0 \mid 1 \mid 5 \mid 10 \mid 25 \mid 50 \mid 100 \mid 150 \mid 200 \mid 250 \mid 300 \mid 350 \mid 400 \mid 450 \mid 500 \mid 750 \mid 1000 \\
abs\_altdiff\_val &::= 1250 \mid 1500 \mid 1750 \mid 2000 \\
angle\_val &::= -180 \mid -135 \mid -90 \mid -45 \mid 0 \mid 45 \mid 90 \mid 135 \mid 180 \\
sr\_val &::= 30000 \mid 25000 \mid 20000 \mid 15000 \mid 10000 \mid 7500 \mid 6000 \mid 5000 \mid 4500 \mid 4000 \mid 3500 \\
sr\_val &::= 3000 \mid 2500 \mid 2000 \mid 1500 \mid 1000 \mid 500 \mid 250 \mid 100 \mid 50 \mid 25 \mid 10 \mid 1 \mid 0 \\
psid\_val &::= -4.5 \mid -4.0 \mid -3.5 \mid -3.0 \mid -2.5 \mid -2.0 \mid -1.5 \mid -1.0 \mid -0.5 \mid 0.0 \mid 0.5 \mid 1.0 \mid 1.5 \\
psid\_val &::= 2.0 \mid 2.5 \mid 3.0 \mid 3.5 \mid 4.0 \mid 4.5 \\
v\_val &::= 25 \mid 50 \mid 75 \mid 100 \mid 125 \mid 150 \mid 175 \mid 200 \mid 225 \mid 250 \mid 275 \mid 300 \mid 325 \mid 350 \mid 375 \\
v\_val &::= 400 \mid 425 \mid 450 \mid 475 \mid 500 \mid 525 \mid 550 \\
alt\_val &::= 500 \mid 1000 \mid 2000 \mid 3000 \mid 4000 \mid 5000 \mid 7500 \mid 10000 \mid 15000 \mid 17500 \mid 20000 \mid 25000 \\
alt\_val &::= 25000 \mid 30000 \mid 35000 \mid 40000
\end{aligned}
$$

# Appendix B

# ACAS X GBDT



members=1,2,3,4,5 (+9995 more).
label=2, confidence=0.914, loss=10.78
G(|response_h_d_1 - response_h_d_2| ≤ 50 ft/s)
For all time, [the absolute difference between [pilot 1's commanded vertical rate] and [pilot 2's commanded vertical rate] is less than or equal to 50 ft/s]
false
members=5,6,45,47,79 (+1609 more).
label=2, confidence=0.509, loss=43.11
F((abs_alt_diff ≤ 100 ft))
At some point, [[absolute altitude difference] is less than or equal to 100 ft]
false
true
members=6,47,79,129,145 (+678 more).
label=2, confidence=0.739, loss=25.44
(SIGN(vert_rate_2) = SIGN(vert_rate_1)) ⟹ response_none_1
Whenever [the sign of [aircraft 2's vertical rate] is equal to the sign of [aircraft 1's vertical rate]], it is also true that [pilot 1 is flying intended trajectory]
members=5,45,85,89,91 (+921 more).
label=1, confidence=0.66, loss=38.82
(COUNT((SIGN(target_rate_2) = SIGN(target_rate_1))) > 23)
[the number of times [the sign of [aircraft 2's RA target rate] is equal to the sign of [aircraft 1's RA target rate]] is greater than 23]
false
true
false
true
members=47,79,162, 171,174 (+379 more).
label=2, confidence=0.974, loss=5.128
F((response_h_d_2 ≤ 1 ft/s))
At some point, [[pilot 2's commanded vertical rate] is less than or equal to 1 ft/s]
members=6,129,145,331,374 (+284 more).
label=1, confidence=0.571, loss=36.49
¬((|response_h_d_1| < 10 ft/s)) ⟹ (|response_h_d_2 - target_rate_2| < 30 ft/s)
Whenever [it is not true that [the absolute value of [pilot 1's commanded vertical rate] is less than 10 ft/s)]], it is also true that [the absolute difference between [pilot 2's commanded vertical rate] and [aircraft 2's RA target rate] is less than 30 ft/s]
members=91,134,149,221,345 (+157 more).
label=2, confidence=0.759, loss=21.82
(COUNT((response_stay_1 ∨ alarm_2)) ≤ 1)
[the number of times [[pilot 1 is responding to previous RA] or [RA alarm occurs on aircraft 2]] is less than or equal to 1]
members=5,45,85,89,102 (+759 more).
label=1, confidence=0.748, loss=34.9
(SIGN(target_rate_2) = SIGN(response_h_d_1)) ⟹ (|vert_rate_2| < 10 ft/s)
Whenever [the sign of [aircraft 2's RA target rate] is equal to the sign of [pilot 1's commanded vertical rate]], it is also true that [the absolute value of [aircraft 2's vertical rate] is less than 10 ft/s)]
false
true
false
true
false
true
false
true
members=1533,1774 label=1, confidence=1.0 category=1
members=47,79,162, 171,174 (+342 more). label=2, confidence=0.979
members=129,374,384, 386,395 (+200 more). label=1, confidence=0.751 category=2
members=6,145,331, 568,584 (+84 more). label=2, confidence=0.843
members=91,134,149, 221,399 (+105 more). label=2, confidence=0.955
members=345,533,661, 686,775 (+47 more). label=1, confidence=0.654 category=3
members=5,45,85, 89,102 (+676 more). label=1, confidence=0.802 category=4
members=155,257,474, 577,656 (+78 more). label=2, confidence=0.699


members=1,2,3,4,5 (+9995 more).
label=2, confidence=0.914, loss=10.78
G(|response_h_d_1 - response_h_d_2| ≤ 50 ft/s)
For all time, [the absolute difference between [pilot 1's commanded vertical rate] and [pilot 2's commanded vertical rate] is less than or equal to 50 ft/s]
true
members=1,2,3,4,7 (+8381 more).
label=2, confidence=0.992, loss=2.176
F(response_follow_2)
At some point, [pilot 2 is responding to current RA]
false
true
members=15,24,46,51,53 (+635 more).
label=2, confidence=0.913, loss=13.14
F((|vert_rate_1 - vert_rate_2| < 1 ft/s))
At some point, [the absolute difference between [aircraft 1's vertical rate] and [aircraft 2's vertical rate] is less than 1 ft/s]
members=1,2,3,4,7 (+7741 more).
label=2, confidence=0.998, loss=0.9283
F(crossing_1)
At some point, [crossing RA is issued to aircraft 1]
false
true
false
true
members=51,124,136,208,255 (+123 more).
label=2, confidence=0.625, loss=36.36
(response_timer_2 = 5) ⟹ (|target_rate_1 - vert_rate_2| < 20 ft/s)
Whenever [[number of seconds remaining in pilot 2's response delay] equals 5], it is also true that [the absolute difference between [aircraft 1's RA target rate] and [aircraft 2's vertical rate] is less than 20 ft/s]
members=15,24,46,53,59 (+507 more).
label=2, confidence=0.984,loss=3.404
G((Sign(target_rate_2) = Sign(vert_rate_1)))
For all time, [the sign of [aircraft 2's RA target rate] is equal to the sign of [aircraft 1's vertical rate]]
members=1,2,3,4,7 (+7730 more).
label=2, confidence=0.999, loss=0.8486
G(response_none_1)
For all time, [pilot 1 is flying intended trajectory]
members=262,1799,2781,3351,3527 (+6 more).
label=2, confidence=0.545, loss=1.2
(Count((alt_diff_1 ≤ 50 ft)) > 21)
[the number of times [[altitude difference relative to aircraft 1] is less than or equal to 50 ft] is greater than 21]
false
true
false
true
false
true
false
true
members=136,255,595,816,1134 (+44 more).
label=1, confidence=0.694
category=5
members=51,124,208,309,377 (+74 more).
label=2, confidence=0.823
members=475,2411,3184,3709,5132 (+7 more).
label=2, confidence=0.583
members=15,24,46,53,59 (+495 more).
label=2, confidence=0.994
members=2,3,4,7,8 (+7602 more).
label=2, confidence=1.0
members=1,62,117,263,301 (+123 more).
label=2, confidence=0.945
members=1799,3351,4706,4756,6029
label=1, confidence=1.0
category=6
members=262,2781,3527,8588,11162,11313
label=2, confidence=1.0

# Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016. 3, 24

[2] W. Royce, “Managing the development of large software systems: Concepts and techniques,” in *IEEE WESCON*. IEEE Computer Society Press, 1970. 6

[3] K. Petersen, C. Wohlin, and D. Baca, “The waterfall model in large-scale development,” in *International Conference on Product-Focused Software Process Improvement*. Springer, 2009, pp. 386–400. 6

[4] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, no. 5, pp. 61–72, 1988. 6

[5] V. D’Silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008. 10

[6] C. Kern and M. R. Greenstreet, “Formal verification in hardware design: A survey,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999. 10

[7] A. Pnueli, “The temporal logic of programs,” in *Foundations of Computer Science, 1977*. IEEE, 1977, pp. 46–57. 10, 27, 28, 53, 56

[8] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018. 10

[9] C. von Essen and D. Giannakopoulou, “Probabilistic verification and synthesis of the next generation airborne collision avoidance system,” *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 2, pp. 227–243, 2016. 10, 11

[10] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer, “A formally verified hybrid system for the next-generation airborne collision avoidance system,” in

*International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015. 10, 11

[11] E. A. Emerson and J. Y. Halpern, “Sometimes and Not Never revisited: On branching versus linear time temporal logic,” *Journal of the ACM (JACM)*, vol. 33, no. 1, pp. 151–178, 1986. 10

[12] E. M. Clarke, O. Grumberg, and D. E. Long, “Model checking and abstraction,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994. 10

[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003. 10

[14] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, “Learning assumptions for compositional verification,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2003, pp. 331–346. 10

[15] J.-P. Katoen, “The probabilistic model checking landscape,” in *ACM/IEEE Symposium on Logic in Computer Science*. ACM, 2016, pp. 31–45. 11

[16] R. W. Gardner, D. Genin, R. McDowell, C. Rouff, A. Saksena, and A. Schmidt, “Probabilistic model checking of the next-generation airborne collision avoidance system,” in *Digital Avionics Systems Conference (DASC)*. AIAA/IEEE, 2016. 11

[17] R. Segala and N. Lynch, “Probabilistic simulations for probabilistic processes,” *Nordic Journal of Computing*, vol. 2, no. 2, pp. 250–273, 1995. 11

[18] J. Sproston, “Decidable model checking of probabilistic hybrid automata,” in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 2000, pp. 31–45. 11

[19] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Courier Dover Publications, 2015. 11

[20] Y. Kouskoulas, D. Genin, A. Schmidt, and J. Jeannin, “Formally verified safe vertical maneuvers for non-deterministic, accelerating aircraft dynamics,” in *International Conference on Interactive Theorem Proving*, 2017, pp. 336–353. 11

[21] B. J. Chludzinski, “Evaluation of TCAS II version 7.1 using the FAA fast-time encounter generator model,” Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-346, 2009. 11

[22] M. J. Kochenderfer and J. P. Chryssanthacopoulos, "A decision-theoretic approach to developing robust collision avoidance logic," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2010, pp. 1837–1842. 12

[23] J. E. Holland, M. J. Kochenderfer, and W. A. Olson, "Optimizing the next generation collision avoidance system for safe, suitable, and acceptable operational performance," *Air Traffic Control Quarterly*, vol. 21, no. 3, pp. 275–297, 2013. 12, 79

[24] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999. 12

[25] P. R. Srivastava and T.-h. Kim, "Application of genetic algorithm in software testing," *International Journal of software Engineering and its Applications*, vol. 3, no. 4, pp. 87–96, 2009. 12

[26] Y. Kim and M. J. Kochenderfer, "Improving aircraft collision risk estimation using the cross-entropy method," *Journal of Air Transportation*, vol. 24, no. 2, pp. 55–62, 2016. 12

[27] J. Kapinski, J. Deshmukh, X. Jin, H. Ito, and K. Butts, "Simulation-guided approaches for verification of automotive powertrain control systems," in *American Control Conference (ACC)*. IEEE, 2015, pp. 4086–4095. 12

[28] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 254–257. 12

[29] J. Deshmukh, X. Jin, J. Kapinski, and O. Maler, "Stochastic local search for falsification of hybrid ystems," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2015, pp. 500–517. 12

[30] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh, "Efficient guiding strategies for testing of temporal properties of hybrid systems," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 127–142. 12

[31] M. J. Kochenderfer, *Decision Making under Uncertainty: Theory and Application*. MIT Press, 2015. 14, 16, 97

[32] R. Bellman, "On the Theory of Dynamic Programming," *Proc. of the National Academy of Sciences of the United States of America*, vol. 38, no. 8, p. 716, 1952. 15

[33] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998. 16, 17

[34] M. Wiering and M. van Otterlo, Eds., *Reinforcement Learning: State of the Art*. New York: Springer, 2012. 16

[35] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, University of Cambridge, 1989. 17, 37

[36] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European Conference on Machine Learning (ECML)*, 2006, pp. 282–293. 18, 20, 37, 42

[37] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012. 18, 19

[38] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256. 18

[39] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016. 19

[40] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in Monte Carlo tree search," in *Computational Intelligence and Games*. IEEE, 2008, pp. 389–395. 19

[41] D. Bertsimas, J. D. Griffith, V. Gupta, M. J. Kochenderfer, and V. V. Mišić, "A comparison of monte carlo tree search and rolling horizon optimization for large-scale dynamic resource allocation problems," *European Journal of Operational Research*, vol. 263, no. 2, pp. 664–678, 2017. 19

[42] D. Hennes and D. Izzo, "Interplanetary trajectory planning with Monte Carlo tree search," in *IJCAI*, 2015, pp. 769–775. 19, 97

[43] T. Cazenave, "Monte-carlo expression discovery," *International Journal on Artificial Intelligence Tools*, vol. 22, no. 01, pp. 1–21, 2013. 19, 26

[44] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline Monte-Carlo tree search planning," in *Advances in Neural Information Processing Systems (NIPS)*, 2014, pp. 3338–3346. 19

[45] T. Pepels, M. H. Winands, and M. Lanctot, "Real-time Monte Carlo tree search in Ms. Pac-Man," *Computational Intelligence and AI in Games*, vol. 6, no. 3, pp. 245–257, 2014. 19

[46] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous upper confidence trees," in *Learning and Intelligent Optimization (LION)*, 2011, pp. 433–445. 19, 41

[47] A. Basak, O. J. Mengshoel, K. Schmidt, and C. Kulkarni, "Wetting and drying of soil: From data to understandable models for prediction," in *International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2018, pp. 303–312. 25, 52

[48] H. Schielzeth, "Simple means to improve the interpretability of regression coefficients," *Methods in Ecology and Evolution*, vol. 1, no. 2, pp. 103–113, 2010. 25

[49] Y. Lou, R. Caruana, and J. Gehrke, "Intelligible models for classification and regression," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012, pp. 150–158. 25

[50] B. Kim, C. Rudin, and J. A. Shah, "The Bayesian case model: A generative approach for case-based reasoning and prototype classification," in *Advances in Neural Information Processing Systems*, 2014, pp. 1952–1960. 25

[51] A. Basak, O. Mengshoel, S. Hosein, and R. Martin, "Scalable causal learning for predicting adverse events in smart buildings," in *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016. 25

[52] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. CRC Press, 1984. 25, 26, 27, 52, 58

[53] R. L. Rivest, "Learning decision lists," *Machine Learning*, vol. 2, no. 3, pp. 229–246, 1987. 25, 52

[54] H. Lakkaraju, S. Bach, and J. Leskovec, "Interpretable decision sets: A joint framework for description and prediction," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2016, pp. 1675–1684. 25

[55] L. Ye and E. Keogh, "Time series shapelets: A novel technique that allows accurate, interpretable and fast classification," *Data Mining and Knowledge Discovery*, vol. 22, no. 1, pp. 149–182, 2011. 25, 52

[56] P. Senin and S. Malinchik, "SAX-VSM: Interpretable time series classification using sax and vector space model," in *IEEE International Conference on Data Mining (ICDM)*, 2013, pp. 1175–1180. 25, 52

[57] M. Shokoohi-Yekta, Y. Chen, B. Campana, B. Hu, J. Zakaria, and E. Keogh, "Discovery of meaningful rules in time series," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2015, pp. 1085–1094. 26

[58] A. Mueen, E. Keogh, and N. Young, "Logical-shapelets: An expressive primitive for time series classification," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2011, pp. 1154–1162. 26

[59] M. W. Kadous, "Temporal classification: Extending the classification paradigm to multivariate time series," Ph.D. dissertation, The University of New South Wales, 2002. 26, 59

[60] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, "The great time series classification bake off: A review and experimental evaluation of recent algorithmic advances," *Data Mining and Knowledge Discovery*, vol. 31, no. 3, pp. 606–660, 2017. 26

[61] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019. 26, 27, 28, 29

[62] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *International Conference on Software Engineering*. ACM, 2010, pp. 215–224. 26

[63] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, 2015. 26

[64] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Automatic programming via iterated local search for dynamic job shop scheduling," *IEEE Transactions on Cybernetics*, vol. 45, no. 1, pp. 1–14, 2015. 26

[65] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, "A machine learning framework for programming by example," in *International Conference on Machine Learning*, 2013, pp. 187–195. 26

[66] S. Muggleton, L. De Raedt, D. Poole, I. Bratko, P. Flach, K. Inoue, and A. Srinivasan, "ILP turns 20," *Machine Learning*, vol. 86, no. 1, pp. 3–23, 2012. 26

[67] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986. 26, 27

[68] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill, "Grammar-based genetic programming: A survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, 2010. 27

[69] "Python full grammar specification," https://docs.python.org/3/reference/grammar.html, accessed: 2019-03-15. 27

[70] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A Field Guide to Genetic Programming*, 2008. 28, 30, 32

[71] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992. 29, 31, 56

[72] M. O'Neil and C. Ryan, "Grammatical evolution," in *Grammatical Evolution*. Springer, 2003, pp. 33–47. 29, 32

[73] P. A. Whigham *et al.*, "Grammatically-based genetic programming," in *Workshop on Genetic Programming: From Theory to Real-World Applications*, 1995, pp. 33–41. 30

[74] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT press, 1998. 32

[75] R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat, and M. P. Owen, "Adaptive stress testing of airborne collision avoidance systems," in *Digital Avionics Systems Conference (DASC)*. AIAA/IEEE, 2015. 35, 83

[76] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer, "Adaptive stress testing for autonomous vehicles," in *Intelligent Vehicles Symposium (IV)*. IEEE, 2018. 38, 118

[77] R. Lee, O. J. Mengshoel, A. Saksena, R. Gardner, D. Genin, J. Brush, and M. J. Kochenderfer, "Differential adaptive stress testing of airborne collision avoidance systems," in *SciTech, Modeling and Simulation Technologies Conference (MST)*. AIAA, 2018. 47

[78] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000. 52

[79] H. Blockeel and L. De Raedt, "Top-down induction of first-order logical decision trees," *Artificial Intelligence*, vol. 101, no. 1-2, pp. 285–297, 1998. 52

[80] J. Schumann, K. Y. Rozier, T. Reinbacher, O. J. Mengshoel, T. Mbaya, and C. Ippolito, "Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems," in *Annual Conference of the Prognostics and Health Management Society*, 2013. 52

[81] M. L. Wong and K. S. Leung, *Data Mining using Grammar Based Genetic Programming and Applications*. Springer, 2006. 53

[82] A. A. Motsinger-Reif, S. Deodhar, S. J. Winham, and N. E. Hardison, "Grammatical evolution decision trees for detecting gene-gene interactions," *BioData Mining*, vol. 3, no. 1, p. 8, 2010. 53

[83] R. Koymans, “Specifying real-time properties with metric temporal logic,” *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990. 56

[84] M. Lichman, “UCI machine learning repository,” 2013. [Online]. Available: http://archive.ics.uci.edu/ml 59

[85] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-Learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. 61

[86] J. K. Kuchar and A. C. Drumm, “The traffic alert and collision avoidance system,” *Lincoln Laboratory Journal*, vol. 16, no. 2, pp. 277–296, 2007. 64

[87] M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos, “Next-generation airborne collision avoidance system,” *Lincoln Laboratory Journal*, vol. 19, no. 1, pp. 17–33, 2012. 64, 65

[88] Federal Aviation Administration, “Next-generation airborne collision avoidance system (ACAS X) requirements matrix,” 2018, RTCA (Unpublished). 65

[89] M. J. Kochenderfer, L. P. Espindle, J. K. Kuchar, and J. D. Griffith, “Correlated encounter model for cooperative aircraft in the national airspace system,” Massachusetts Institute of Technology, Lincoln Laboratory, Project Report ATC-344, 2008. 68, 69

[90] M. J. Kochenderfer, M. W. M. Edwards, L. P. Espindle, J. K. Kuchar, and J. D. Griffith, “Airspace encounter models for estimating collision risk,” *AIAA Journal on Guidance, Control, and Dynamics*, vol. 33, no. 2, pp. 487–499, 2010. 68

[91] International Civil Aviation Organization, “Surveillance, radar and collision avoidance,” in *International Standards and Recommended Practices*, 4th ed., Jul. 2007, vol. IV, annex 10. 69

[92] D. M. Asmar, M. J. Kochenderfer, and J. P. Chryssanthacopoulos, “Vertical state estimation for aircraft collision avoidance with quantized measurements,” *AIAA Journal on Guidance, Control, and Dynamics*, vol. 36, no. 6, pp. 1797–1802, 2013. 82

[93] C. A. Ippolito, S. Campbell, and Y.-H. Yeh, “A trajectory generation approach for payload directed flight,” in *AIAA Aerospace Sciences Meeting*. AIAA, 2009. 95, 97

[94] J. Tisdale, Z. Kim, and J. K. Hedrick, “Autonomous UAV path planning and estimation,” *IEEE Robotics & Automation Magazine*, vol. 16, no. 2, 2009. 95

[95] Z. Zeng, A. Lammas, K. Sammut, F. He, and Y. Tang, "Shell space decomposition based path planning for AUVs operating in a variable environment," *Ocean Engineering*, vol. 91, pp. 181–195, 2014. 95

[96] I. K. Nikolos, K. P. Valavanis, N. C. Tsourveloudis, and A. N. Kostaras, "Evolutionary algorithm based offline/online path planner for UAV navigation," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 33, no. 6, pp. 898–912, 2003. 95, 97

[97] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, 2009. 95, 97

[98] A. Gasparetto, A. Lanzutti, R. Vidoni, and V. Zanotto, "Experimental validation and comparative analysis of optimal time-jerk algorithms for trajectory planning," *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 2, pp. 164–181, 2012. 95

[99] A. Trevisani, "Experimental validation of a trajectory planning approach avoiding cable slackness and excessive tension in underconstrained translational planar cable-driven robots," in *Cable-Driven Parallel Robots*. Springer, 2013, pp. 23–39. 95

[100] A. Narkawicz and G. E. Hagen, "Algorithms for collision detection between a point and a moving polygon, with applications to aircraft weather avoidance," in *AIAA Aviation Technology, Integration, and Operations Conference*. AIAA, 2016. 95

[101] S. Bensalem, K. Havelund, and A. Orlandini, "Verification and validation meet planning and scheduling," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 1, pp. 1–12, Feb 2014. 96

[102] D. Giannakopoulou, C. S. Pasareanu, M. Lowry, and R. Washington, "Lifecycle verification of the NASA Ames K9 rover executive," in *ICAPS Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, 2005. 96

[103] F. Raimondi, C. Pecheur, and G. Brat, "PDVer, a tool to verify PDDL planning domains," in *ICAPS Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*, 2009. 96

[104] J. Frank, "Reflecting on planning models: A challenge for self-modeling systems," in *IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2015, pp. 255–260. 96

[105] J. Puig-Navarro, N. Hovakimyan, N. M. Alexandrov, and B. D. Allen, "Silhouette-informed trajectory generation through a wire maze for UAS," in *AIAA Aviation Technology, Integration, and Operations Conference (ATIO)*. AIAA, Jun 2018. 96, 97, 101, 102, 103

[106] N. M. Alexandrov and B. D. Allen, "Autonomy Teaming & TRAjectories for Complex Trusted Operational Reliability (ATTRACTOR)," NASA ARMD TACP/Convergent Aeronautics Solutions Project Showcase, Oral Presentation, Mountain View, CA, Sep 2018. 96, 101

[107] H. Erzberger, T. Lauderdale, and Y. Chu, "Automated conflict resolution, arrival management, and weather avoidance for air traffic management," *Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering*, vol. 226, no. 8, pp. 930–949, 2012. 97

[108] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001. 97

[109] V. G. Boltyanskii, R. V. Gamkrelidze, and L. S. Pontryagin, "The theory of optimal processes. I. the maximum principle," TRW Space Technology Labs, Los Angeles, CA, Tech. Rep., 1960. 97

[110] R. E. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 189–211, 1990. 97

[111] C. Ippolito, K. Krishnakumar, S. Hening, and S. Sankararaman, "A modeling, simulation and control framework for small unmanned multicopter in urban environments," in *AIAA SciTech, Intelligent Systems Conference (IS)*. AIAA, 2019. 99

[112] R. Lee and C. Ippolito, "A perception and mapping approach for plume detection in payload directed flight," in *AIAA Infotech@Aerospace Conference*. AIAA, April 2009. 99

[113] R. Lee and D. H. Wolpert, "Game theoretic modeling of pilot behavior during mid-air encounters," in *Decision Making with Imperfect Decision Makers*, ser. Intelligent Systems Reference Library. Springer, 2012, vol. 28, ch. 4, pp. 75–111. 99

[114] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Iowa State University, Computer Science Department, Tech. Rep. TR 98-11, 1998. 102

[115] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011. 102