AUTOMATED MODELING OF HUMAN-IN-THE-LOOP SYSTEMS

by

Noah Marquand

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Aeronautics and Astronautics

School of Aeronautics and Astronautics

West Lafayette, Indiana

December 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Karen Marais, Chair
School of Aeronautics and Astronautics

Dr. Milind Kulkarni
School of Aeronautics and Astronautics

Dr. Dengfeng Sun
School of Aeronautics and Astronautics

Approved by:
Dr. Gregory A. Blaisdell

Dedicated to the friends and family that trusted me to make it this far

ACKNOWLEDGMENTS

In acknowledgement of Purdue University's contributions of seed funding for this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Chapter 1

State: The remembered information in the system that affects how it responds to different conditions, and what it is capable of

Input: The external conditions that could alter system state
Path: The transition between states due to a provided input

Parameter: A measurable system characteristic that defines system state

Control: A measurable user action that can affect system state

Trace: A record of parameters and controls taken during system operations

Microstate: A list of parameters recorded at the same time, a specific instantiation of a state

Microinput: A list of controls recorded at the same time, a specific instantiation of an input

Reading: A list of parameters and controls taken at the same time

Factor: A system feature that defines how it can be modeled

Continuity: The number of continuous parameters and controls a system has relative to its discrete metrics

Parallelism: The number of controls a user can provide at once

Temporality: The degree to which the system expects users to provide inputs between state updates

Boundedness: The degree an analyst understands the relevant parameters and controls for the system

Chapter 2

Substate: A state that appears to exist in the path between two other states

Chapter 3

True Positive Rate: The rate at which a classifier correctly identifies data

Positive Predictive Value: The rate at which the reported classifications of a classifier are correct

ABSTRACT

Safety in human in the loop systems, systems that change behavior with human input, is difficult to achieve. This difficulty can cost lives. As desired system capability grows, so too does the requisite complexity of the system. This complexity can result in designers not accounting for every use case of the system and unintentionally designing in unsafe behavior. Furthermore, complexity of operation and control can result in operators becoming confused during use or receiving insufficient training in the first place. All these cases can result in unsafe operations. One method of improving safety is implementing the use of formal models during the design process. These formal models can be analyzed mathematically to detect dangerous conditions, but can be difficult to produce without time, money, and expertise.

This document details the study of potential methods for constructing formal models autonomously from recorded observations of system use, minimizing the need for system expertise, saving time, money, and personnel in this safety critical process. I first discuss how different system characteristics affect system modeling, isolating specific traits that most clearly affect the modeling process Then, I develop a technique for modeling a simple, digital, menu-based system based on a record of user inputs. This technique attempts to measure the availability of different inputs for the user, and then distinguishes states by comparing input availabilities. From there, I compare paths between states and check for shared behaviors. I then expand the general procedure to capture the behavior of a flight simulator. This system more closely resembles real-world safety critical systems and can therefore be used to approximate a real use case of the method outlined. I use machine learning tools for statistical analysis, comparing patterns in system behavior and user behaviors. Last, I discuss general conclusions on how the modeling approaches outlined in this document can be improved and expanded upon.

For simple systems, we find that inputs alone can produce state machines, but without corresponding system information, they are less helpful for determining relative safety of different use cases than is needed. Through machine learning, we find that records of complex system use can be decomposed into sets of nominal and anomalous states but determining the causal link between user inputs and transitions between these conditions is not simple and requires further research.

1. INTRODUCTION

Robust, complex systems that interact with humans are difficult to design (Solar-Lezama, Rabbah, Bodik, & Ebicioglu, 2005). Complex systems are typically sensitive to user inputs, and this sensitivity lends itself to more complex interactions between a system's conditions and its inputs. This behavior can result in systems becoming difficult to understand as either a designer or operator, masking how they respond to changing inputs and environmental shock, and making them less safe to use. Current methods of improving the safety of these HITL systems use model checking techniques to analyze behavior under different conditions

These techniques are often bottlenecked behind the need for a system model, which can be difficult to obtain. This research focuses on studying potential methods for autonomously constructing these system models using logical, statistical, and machine learning methods, without expert input.

1.1 Motivation

Many systems can accidentally reach failure modes without any component failures occurring. In the case of Asiana Flight 214, during final approach to the runway, unbeknownst to the pilots, the glide slope was too steep and airspeed too low. Pilots noticed the engines were set to idle, despite the auto-throttle system being in the armed position, and attempted to regain speed, but were unable to avoid a crash into the runway, during which the plane broke apart and claimed three lives.

The NTSB investigation found that the auto-throttle system did not automatically switch on as expected because it required neither or both flight director computers to be on, but only one computer was on during approach. This confusing priority system is credited in the report as being one of the major contributors to the pilots' "faulty mental model", which resulted in the crash (National Transportation Safety Board, 2013). If such a confusing aspect could be caught before the system went into production and use, it could prevent accidents.

1.2 Current Approaches

Formal methods of model checking use mathematical tools to determine whether different conditions allow the system in the model to reach an uncontrolled system state, or whether an anomalous state can be returned to user control (NASA Langley, 2016). Such methods rely on specific types of system models, called formal models. Building these models by hand requires a near exhaustive understanding of the system to achieve a level of detail that is useful for determining specific safety improvements. This expertise is time-consuming to achieve, and comes with great monetary expense, so often the models checked are of a reduced complexity, or of only a specific component of the main system, making them less useful for examining overall safety (Aalto, Husberg, & Varpaaniemi, 2003).

Much of current work focuses on techniques for autonomously identifying and labelling anomalous data (Puranik & Mavris, 2018), while other work focuses on improving autonomous model construction on digital subsystems. Emphasis is placed on mapping the decision-making space for autonomous systems as well, with some demonstration of autonomous model construction for MATLAB models of lane-change decision making systems (Selvaraj, Farooqui, Panahandeh, & Fabian, 2020). Most of this effort is dedicated to learning discrete models or distinguishing two phases of system behaviors in known systems. This work focuses on minimizing the required system knowledge and expanding the modeling process to capture system-wide phenomena.

1.3 Background and Definitions

This work predominantly uses the terminology of formal models that we expand upon to capture complex behavior. To begin our investigation, we define the most basic terms and provide some background on how they are used in industry.

1.3.1 Formal Models

Formal models are precise definitions of system operations. One of the most common and recognizable formal models is the state machine. There are many variants, but most simply, state machines are composed of three parts:

1. *States*: The remembered information in the system that affects how it responds to different conditions, and what it is capable of. For example, a light switch has states “On” and “Off”
2. *Inputs*: The external conditions that could alter system state. To continue with the previous example, a switch could be flipped to “On” or to “Off”. Note that some inputs may not always be available or may not always alter system states.
3. *Paths*: The transition between states due to a provided input. Using the same example, a light switch in “On” could be flipped to “Off”, after which it would be in “Off.”

State machines used for safety applications, like those used in formal safety checks, often label states as nominal or anomalous (Jung, et al., 2021). Nominal states are considered acceptable and part of standard operations. Anomalous states are abnormal, and perhaps hazardous. With these labels, a system designer can examine their system state machine and use formal methods to examine nominal states’ proximity and relation to anomalous states, which can be further used to compare the relative safety of each state, and so on.

The relevant states, inputs, and paths of a system may be unknown. It is not always clear when a state transition has occurred, or if a small change is relevant to system operations. Modelers then need to establish definitions for these components that capture various expected nominal behaviors, and how they might transition between themselves and anomalous behaviors; with enough depth that actionable change can be made where needed. This is difficult to do without exact knowledge of the system, and so this document focuses on how to construct such system models with only a recording of system use.

1.3.2 System Records

System records are a collection of measurements of the system and the user during operation. In this document, we will refer to each system characteristic measured as a *parameter*. For example, altitude is a parameter in a flight recording. We refer to a user characteristic measured in a recording as a *control*. One such control in a flight recording is the pilot throttle setting.

We will also use the term *trace* when referring to the system recording itself (IBM, 2017). Most simply, a trace can be represented with a matrix, where columns indicate each unique parameter/control and rows indicate simultaneous measurements.

1.3.3 Combined Definitions

Combining the two sets of terminology then provides some insight into how we may begin to examine systems. Parameters, being measurements of the system characteristics, are indicative of the current system state. For example, altitude, attitude, and velocity, are useful parameters for determining if an aircraft is in a stalled state. We can then view each reading of parameters taken at the same time as merely a specific instantiation of their state classification, which we call a *microstate*.

Similarly, controls are measurements of user characteristics that could be diagnostic of wider input categories. For example, a yoke deflection right and up with left pedal pressure might generally correspond to a bank right input. Control measurements recorded at the same time make up a specific instantiation of their input classification, which we call a *microinput*.

With these concepts in mind, each row in the trace should be useful for predicting the next. I will refer to each row of measurements taken at the same time as a *reading*.

1.4 System Factors

Systems come in a variety of forms and with variety comes different assumptions on system operations. Different assumptions affect how we must collect and extract information and need to be carefully considered. We consider four main aspects of how systems operate:

1. Continuity: Are important performance metrics discrete, continuous, or a mixture thereof?
2. Parallelism: Does the system accept multiple controls at once?
3. Temporality: Does the system continuously update its state without human intervention?
4. Boundedness: Are relevant inputs and parameters visible to the user upon use?

In this document, we refer to each of these considerations as the *system factors*. Each of these factors affect how we can effectively collect a trace and how we can detect states and paths, as discussed in further detail in each section below.

1.4.1 Factor: Continuity

Systems with discrete characteristics have clear delineations between different states, with little ambiguity between them. For example, a menu-based digital system has clear distinctions between states, and inputs are categorical or Boolean. This type of system can be analyzed using logical techniques, checking whether exact inputs are provided, and so on.

By contrast, continuous systems have no clear delineations between states or parameters, making exactitude difficult and any purely logical deductions obscured. For example, flight uses many distinct continuous parameters, so without system knowledge, it is difficult to make exact logical conclusions. Instead, we can analyze this type of system with statistical methods, examining the probability of changes occurring based on a range of values.

In general, we can assume that a system with purely discrete characteristics is simpler to analyze than a system with continuous variables. The most complex of cases being a mix of discrete and continuous variables, which would require a mix of logical and statistical methods to analyze. Most real-world, safety critical systems would be considered part of this last category.

1.4.2 Factor Parallelism

Serial systems accept a singular control as input at any given time. For example, menu-based digital systems will often only accept one input at a time. Parallel systems can accept multiple simultaneous controls. For example, each axis of the control yoke of an airplane could be considered a separate control, making flight a parallel system.

In general, serial systems are simpler to analyze than parallel systems for two main reasons. First, the added variability of possible inputs in parallel systems makes it much more difficult to use logical methods to analyze them, because complex microinputs are less likely to be exactly replicated, making it more difficult to determine when the system behaves in the same way in multiple points in the trace. Second, parallel systems can simultaneously accept discrete and continuous controls, requiring more specialized applications of each statistical/logical tool than if only one type were usable at a time. However, most real-world complex systems would be parallel systems.

1.4.3 Factor: Temporality

Atemporal systems do not change their system state without user input. For example, a menu-based digital system might not change state until the user presses a button. Temporal systems update their state without user input, and in some cases, constantly. Many safety critical systems that rely extensively on physical phenomena, like flight, would fall into this category, as vehicle physics are constantly operating on the system.

Modeling temporal systems is much more complicated than the atemporal type. Atemporal systems can record readings in the trace after every input is provided and that is enough, but temporal systems require that a modeler estimate how quickly they need to be able to detect state transitions and record readings at the corresponding frequency. Different frequencies may not capture all behavior and need to be studied to find consistent system behavior.

1.4.4 Factor: Boundedness

Bounded systems have clear and obvious boundaries for what is a relevant parameter control and what is not. For example, it is clear in a menu-based digital system that the controls used to interact with the system are the button selections made in the menu, and it is clear that the system state has changed when the display updates to a new menu screen. Unbounded systems have non-obvious boundaries. In flight, it is unclear which parameters are meaningful for determining state, and how meaningful they are. For instance, consider that while it is evidently useful to know the aircraft velocity, it is unclear what that velocity needs to be relative to when determining states.

In general, we consider bounded systems to be simpler to analyze, as they require fewer steps to determine state definitions. Unbounded systems require greater system knowledge and still require model comparisons to determine which parameters are relevant for safety.

1.5 Basic Approach

With these terms and characteristics in mind, we can outline a general process for constructing a state machine from a trace:

1. Determine system factors
2. Record a trace of system operations
3. Produce definitions for system states from the trace
4. Produce definitions for system paths from the trace

1.5.1 Collecting System Traces

Once we classify with the four factors, we can collect the trace. Each factor presents unique implications for how we need to collect and manipulate a meaningful trace. Continuous systems for instance are less likely to have exact repetitions of readings than discrete counterparts, making it less clear where state boundaries lie. For example, in flight it is unlikely than any two recorded flights will pass through the same point with the same velocity, but the distinction between the start of a stall and nominal flight is subtle. To counteract this effect, we introduced some artificial discretization into continuous data to make microstates more distinct (see Section Implications of the System3.1.2). This process requires some system knowledge, with educated guesses for what is likely to be a meaningful change in parameter and control values.

To record serial systems, we only need a single data channel for tracking performance, with an associated time channel if the system is temporal. Parallel systems by contrast require multiple channels, which can add complexity and time to the trace construction process depending on the measuring techniques used.

When we record atemporal systems, parameters and controls need to be measured after each input. Temporal systems then have multiple options for how they can be recorded, which have different behaviors. First, if the system continuously accepts user inputs, like in flight, where the user is continuously supplying a yoke input, it is efficient to record behavior at a fixed sampling rate to capture behavior. This sampling rate needs to be determined with some degree of system expertise, based on the rate at which states can change. Alternatively, if the system is designed to idle between inputs, like in a digital system such as a computer, readings should be taken when inputs are made, otherwise inputs and timings can be lost in the recording process.

Lastly, recording bounded system traces only requires recording the obvious metrics of the system, whereas unbounded systems require requires any metric that might be relevant, even indirectly. It should also be recognized that many parameters used in unbounded systems may not be used for state identification in their raw state.

After factors are considered, we need to ensure that the trace data captured is enough to determine a system model. This means that the trace should include a variety of typical operating behaviors, capturing mostly nominal behavior with known anomalous behaviors labeled. In general, we assume that deviation from the behavior seen in most of the trace should be considered anomalous. This ideation is used in many similar works for anomaly detection (Puranik & Mavris, 2018), which can be used in conjunction with the methods specified in this document to label states generated in the state machine.

Traces also need to be as close to exhaustive as feasible, including multiple repetitions of all typical procedures that are to be considered part of the system. This reduces the likelihood that any typical procedures are considered anomalous and provides information on how slight variations in execution of procedures can affect the outcome.

1.5.2 Isolating System States

To identify states from the trace, we need to identify common trends in the behavior of microstates. Some questions we might ask are:

1. Are specific configurations of parameters distinct from other configurations? For example, a plane transitioning from climb to cruise will, relative to the time scale of the flight, quickly transition from a high pitch, to a neutral one, making the delineation from high to neutral pitch distinct.
2. Do specific configurations of parameters occur more frequently than others? For example, high throttle is most often paired with high speed because an aircraft operating at a high throttle tends to accelerate to its top speed.
3. Do specific configurations of parameters often result in known anomalous parameters? For example, a rapid descent might commonly correlate with a stall indicator.
4. How frequently are specific configurations of parameters paired with each configuration of controls? For example, in menu-based navigation, some inputs are not available always, making them potentially diagnostic of state.

We can use logical and statistical measures to attempt to answer each of these questions for each potential state. Each system may need a different tool to assess these and determine the definition of its relevant states.

1.5.3 Isolating System Paths

Once we have established definitions for states, we can begin to connect them together with paths and inputs, which are conceptually linked together. In a state machine, every path is the result of an input, including paths that return to the initial state. We can therefore extract classes of inputs by first examining the paths observed in the trace.

To identify paths then, we need to identify common trends in the transitions between microstates. As with identifying states, there are several questions we can ask:

1. How frequently do states transition between one another?
2. In any given state, do specific configurations of controls result in specific state transitions? Do they always result in the same state transitions?
3. In any given state, do specific changes in microstate result in specific state transitions? Are these changes associated with specific configurations of controls?

As with state identification, these questions can be answered with logical and statistical tools to identify possible inputs from known paths.

1.6 Probing Potential Methods

To further explore the process we have outlined, the rest of this research extrapolates on the application of theory to two systems. The first system is a simple case, an automatic coffee machine with a menu-based, digital interface. This system exhibits discrete, serial inputs, atemporal states, and bounded parameters and controls, allowing for testing of basic theory and logical analysis techniques.

The second system a complex case, a flight simulator in cruise. This case allows for the extension of theory into more “real-world” data with mixed discrete/continuous parameters and controls, temporal states, and unbounded parameters. To analyze it, we need to utilize more complex, statistical methods, making it a good comparison of methods with the logical, simple case.

1.6.1 Application of Theory to a Simple System

In the case of the coffee machine, the trace was recorded prior to our research, and only includes the controls provided. This complicates the process, requiring that states be extracted from inputs alone, but because of its menu-base architecture, this should still be possible if we distinguish states by comparing where inputs are seen in the trace relative to one another.

The system exhibits one of the simplest configurations of factors. Discrete, serial, bounded controls allow us to consider each unique control to be its own input, recorded in sequence. With some basic simplifications, we can consider the system atemporal as well, further simplifying analysis. This allows us to further consider many individual recordings as functionally identical, simplifying the logical processes and increasing our assurance that all common paths are navigated in the trace.

In general, we decompose the trace by first partitioning the trace with states based on how similar each input sequence is to other sequences in the trace. With basic state definitions, we then compare paths between them and ensure that they are mutually consistent to refine the model which concludes model construction.

Overall, this process demonstrates that a simple, menu-based system can be logically decomposed into a state machine model from an input trace using our methodology.

1.6.2 Application of Theory to Complex System

In our flight simulator case, the system factors suggest specific methods of analysis that differ from the simpler case. Mixed discrete/continuous parameters and controls cannot be simply analyzed by logical tools, so we instead reduce the system to its continuous metrics and use statistical tools, as the continuous metrics are likely to be the most informative. Additionally, parallel parameters and controls do not repeat in the trace frequently enough for microstates and microinputs to be directly considered states and inputs in a useful state machine. Here, we use statistical tools to measure similarity in behaviors for different microstate and micropath configurations.

This system is also a continuously updating temporal system, so we explore techniques for finding state and path definitions at different sampling frequencies. The parameters used during this process are also unbounded, so we demonstrate methods for producing new parameters and down-select to a useful set as well.

In general, this exploration begins with state definitions we find by comparing parameter distributions in varying sampling frequencies and parameter configurations. With these state definitions, we explore input identification techniques from the paths now visible in the trace, emphasizing statistical methods. This process demonstrates the complexities of applying statistical techniques without system knowledge. We suggest future exploration into improving methods.

2. APPLICATION OF THEORY TO SIMPLE SYSTEM

This chapter goes into detail on the methods used for constructing a state machine for a simple system, an automatic coffee machine. Discussion will begin with a system overview, where I will describe the system and its basic characteristics. Next, I will cover how we collected and organized the system trace, which will lead into how we isolated preliminary states from the trace and organized them into final state definitions. Last, we will discuss the effectiveness of our methodology for constructing a state machine for the system.

2.1 System Overview

To begin, we sought to study a test case with known, deterministic behavior to simplify analysis and check modeling results, as well as a test case that would be simple to collect trace data for. Here, we elected to test methods on an automatic coffee machine, as existing experimental trace data was available for use, and its functionality is well known.

2.1.1 Defining the System

The coffee machine used in the pre-recorded trace was a unit placed in an office lounge that could be periodically bulk loaded with drink materials, so that any user interactions were limited to loading cups and following a digital menu on the machine face, like digital soda machines. The menu options themselves were available for study in the system manual itself.

We can imagine system states for this case as the steps in the drink setup process, and any unique selections of the user. For instance, one system state might be "Empty cup in tray, coffee drink selected" while another might be "No cup in tray, hot drink selected, hot chocolate selected". System inputs would then be the menu selections from the user and any cup manipulation

2.1.2 Implications of System Factors

Because this system has a limited set of button selections for controls, we can describe the system as having discrete controls. This feature means that each control is distinct and categorical, lending system analysis towards logical methods over statistical. Controls are also input serially, as the system does not accept multiple menu inputs at once. This further simplifies any logical analysis performed by connecting state changes to a single control input instead of multiple.

This system largely operates without any regard for time between inputs, making system states atemporal. We can then assume that all system state changes are directly related to user inputs, simplifying our analysis to data directly recorded in the initial trace. Two exceptions to this rule exist however, the first being that the user manipulation of their drink cup would not affect whether the machine would pour drinks. As such, a user could potentially place a cup in the tray, and then remove it before the drink was poured, which would not be captured in the trace without timing information on when the cup was moved and when the drink poured. Drink pour times were not recorded, so to simplify analysis, I assumed that this event did not happen in any recorded trace.

The second exception to atemporal states in the system involves the system's internal "timeout condition. If a user didn't make an input in a certain amount of time, the system would reset to the start state (excluding cup positions). This feature was not engaged for most cases, so to simplify analysis, I cut and labeled the trace instance where timeouts occurred as if no future behavior was known.

Lastly, this system is clearly bounded, the only state changes occur directly from the user inputs recorded in the original trace. Most inputs could be found as programmable options in the manual, and the others used in the trace, like "move cup to tray" are obvious. This bounding simplifies analysis, ensuring that everything recorded is relevant for determining states and nothing is missed.

2.2 System Trace

The trace used for this case was originally collected as part of an unrelated study on human-device interactions and captured interactions with video. The video footage was then transcribed in a spreadsheet with the user behaviors at the video time stamp, with behaviors such as "Grab cup", "Press Coffee Drink 1", and "Think". This transcript included two camera failure incidents labeled as user actions, at which point it was inherited by the current project. Only the spreadsheet transcript was used and available for our demonstration.

To begin processing the transcript into a usable trace, I cleaned the data set of typos, duplicate labels, and "non-interactions". Duplicate labels in this case refer to different input labels for the same input. For instance, “Grab cup”, “Move cup”, and “Release cup” were used in some cases and “Place cup in Tray” in others. For this study, I used as few input labels as possible to simplify the analysis. The original trace also included “non-interaction” inputs, such as “Think”. These actions were not relevant for our study, as they do not change the system state, and I removed them from the transcript.

To normalize user interactions around performing a task, I considered a user making a single drink as a single interaction. Any user making multiple drinks in a row would then simply have multiple interactions recorded in the final trace. To finish turning the transcript into a usable trace, I inserted input labels for “Start” and “End” into the transcript to demarcate the bounds for each interaction. This process provided a total of 102 separate interactions recorded over the course of two afternoons. This dataset sufficiently maps the system space, as it is not expected that the drinks made day to significantly change, so system use will not vary largely beyond what was seen in the original recording.

With these changes made to the transcript, we now have a usable trace, seen as a series of user inputs to the system, with some inputs marking the beginning and end of a drink being made. To condense this trace, I converted each unique label into a unique numerical ID, so the entire trace can be represented as a column vector. Table 1 shows some of the sample inputs and their corresponding ID values.

Table 1: Sample Inputs, IIDS, and Number of Uses in Data Set

Inputs	IID	Number of uses in set
Start	1	102
End	3	102
Place Cup in Tray	4	102
Select Coffee 2	16	24
Select Coffee Drink	19	159
Select Large	25	35
Select Milk	29	2

2.3 Isolating System States

With this system understanding, we can assume that two systems in identical states receiving different inputs should arrive at different states. An extremely simple model of this behavior could use a flowchart that perfectly copies the recorded behavior, splitting any time a new sequence is recognized. This model would be inadequate though, because it cannot replicate behaviors that do not exactly follow a use instance in the trace. For example, if users are presented with the option to select between three drink sizes, but users in the trace only select the largest size for a specific drink variant, the simple model would not recognize that there was an option to select other sizes. If we consider all interactions not represented in our model as anomalous, this could result in otherwise nominal behavior appear to be anomalous as interactions grow longer and more complex. Even a simple use case like the “Select Cancel” would appear as a completely unique branch in the simple model, with all subsequent inputs appearing as anomalous.

To begin addressing these issues, we needed to develop a more robust method for identifying changing states rather than differences in user behavior. In a menu-based system, the clearest indicator of states differing is a difference in input availability to the user. Because this system has diverse inputs correlated the menu states, input availability can be observed with two methods or contexts: the local and global contexts. When the trace is then examined under both contexts at once, we can generate lists of available inputs for each step in the trace. When available inputs change, a new state is reached.

2.3.1 The Local Context

The local context focuses on examining what inputs in the trace immediately follow all the other inputs. This context assumes that inputs are only available if they have been seen to immediately follow the previous input in the trace. For example, if “Select Coffee Drink” is only ever followed by “Select Coffee 1”, “Select Coffee 2”, “Select Coffee 3”, and “Select Cancel”, no other inputs are considered available following a “Select Coffee Drink” input. This context works well for menu-based systems with a diverse input set that is strongly correlated with previous inputs because it can directly identify those correlations.

However, it struggles to operate on systems with low-diversity inputs because they can be highly repetitive. The frequency of each input in the trace will result in each input being seen to follow each other input and so provide no new information. For example, a menu-based system might operate off yes/no inputs and provide a targeted question following each input. Analysis would suggest that yes/no is always available, but this provides no information on whether yes and no responses were both in the trace for the specific question asked. This can be avoided by carefully labelling inputs to increase input diversity: in the sample case, convert “Yes/No” into “Yes for Q1/ No for Q1”. Such conversions would need to be done with some degree of system knowledge and may not be possible in all cases. Luckily, such low-diversity input systems do not frequently occur in safety critical systems because fewer input options necessitate longer input sequences to transfer the same amount of information to a system. This inefficiency extends the time required to complete any task, making them less capable of resolving time-critical hazards.

The local context will also struggle on systems that do not have a strong correlation between their inputs and their precursor inputs. It would instead favor exactly copying the trace behavior. For example, a menu-based questionnaire system emulating a multiple-choice quiz may not have relations between individual responses, regardless of how diverse the potential inputs are.

Lastly, this context is limited in that it does not meaningfully constrain the number of available inputs for inputs that have diverse following inputs. For example, "Select Cancel" reverts the system to the previous state and can be input to the system in a variety of states. As such, many different inputs immediately follow it, none of which are only available after "Select Cancel" is input to the system.

2.3.2 The Global Context

The global context focuses on examining what inputs always occur before other inputs in the trace. This context assumes that inputs that always occur before other inputs are mandatory for the second input to occur. For example, "Select Cappuccino" is only seen in traces where "Select Gourmet Drink" has already been input to the system, so "Select Cappuccino" is never listed as an available input until at least "Select Gourmet Drink" has been input by the user. This global context complements the local context weakness for inputs with diverse following inputs. In the same example given before, when "Select Cancel" is input to the system, only inputs that have had their mandatory precursors are considered available in the global context, so the field of available inputs in narrowed.

However, this strength makes this context useful only for high-diversity, high-correlation systems. If no inputs have mandatory precursors, this context does not help narrow the inputs available. Additionally, if the system allows itself to return to previous states, the global context becomes less useful for identifying input availability. For instance, a use instance where a user inputs "Select Cancel" after every input until each menu-option is exhausted, would see every mandatory precursor having been input, so the global context would not be useful for narrowing available inputs.

2.3.3 The Combination Model

These contexts focus primarily on input availability, which can be indications that the system is in a different state but is not the sole determining factor for state differentiation. For example, the menu display for selecting drink size presents the same options regardless of what drink is being made. Input availability alone would suggest that all instances of this menu are the same, even though “Select Cancel” would direct to different menus depending on prior inputs. To maintain consistency in paths, we define initial states iteratively using the following process:

1. Starting at the Start state for each instance, we can navigate through the trace input by input.
2. If our dual context method suggests that multiple inputs are available, a state has been reached.
3. If the same input sequence is used to reach a state as in a prior instance, the same state is reached.
4. Once a state is reached, navigate to the next instance, until all instances have been examined.
5. This process is repeated, navigating from each of the new states as if they were the start state, until the full trace has been examined and no new states can be detected.

This process creates an initial branching tree model, with all instances represented as a sequence of state-to-state paths, reconvening in the end state as shown in Figure 1: Simplified branching tree model.

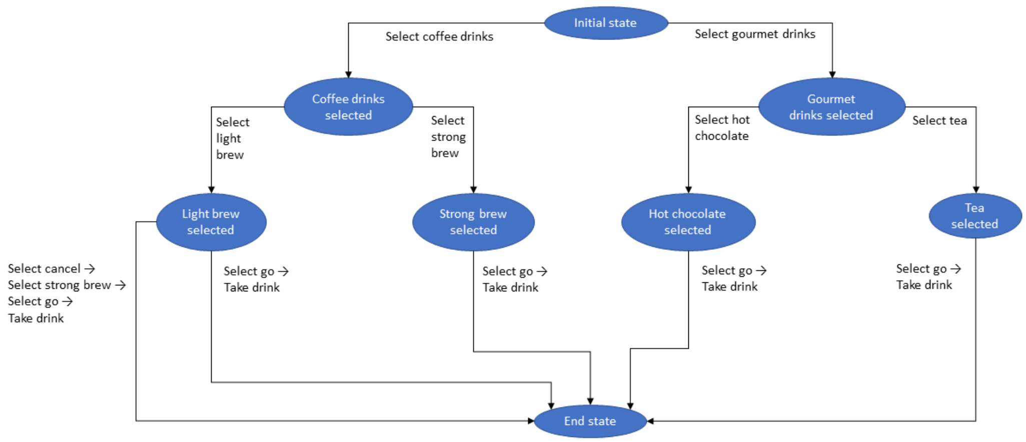


Figure 1: Simplified branching tree model

2.4 Refining Known States

The state definitions generated in this manner are incomplete, often being redundant or inconsistent. Some states are functionally identical, sharing paths, which suggests that some preliminary state definitions are duplicated. Some paths are inconsistent and share some behavior with other behavior that suggests that some states are not being detected in the initial identification pass. Additionally, some true states are likely missing from the model. As shown in Figure 2: Instance dilution with inputs, the total number of people on each path decreases with each input. If paths are selected randomly from a current state, the probability that all paths out of a given system state are seen in the trace is dependent on the number of users that arrive at said state.

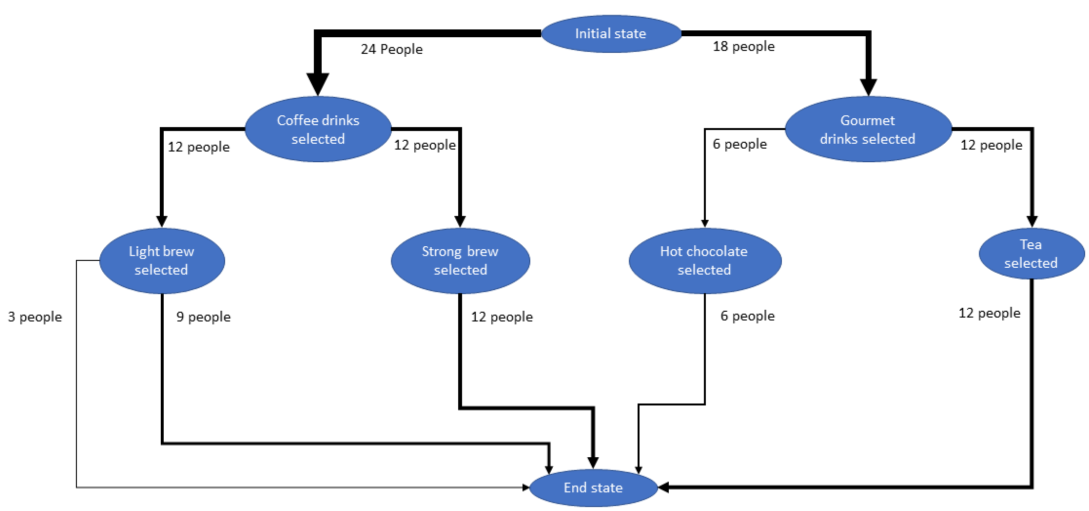


Figure 2: Instance dilution with inputs

These issues can be addressed in three ways:

1. Create a second model operating in the reverse order to the trace. This concentrates instances at the end state instead of the start state, making it more likely to catch missing states than the forward order model.
2. Create logical rules for identifying when states are functionally similar. This would allow for some state definitions to be combined, simplifying the model.
3. Create logical rules for identifying when paths can be broken into segments passing through additional states. This would ensure that inputs and paths are consistent.

2.4.1 Parallel Models

To construct a reverse order model, we ran the same model construction process with the entire trace inverted, starting with the end states, and running to the initial states. This concentrated instances into the end paths as intended. Figure 3: Simplified reverse branching tree model, shown below, demonstrates how a reverse model using the same simplified trace might appear. Now, both a forward and reverse model can be run together to identify states and paths.

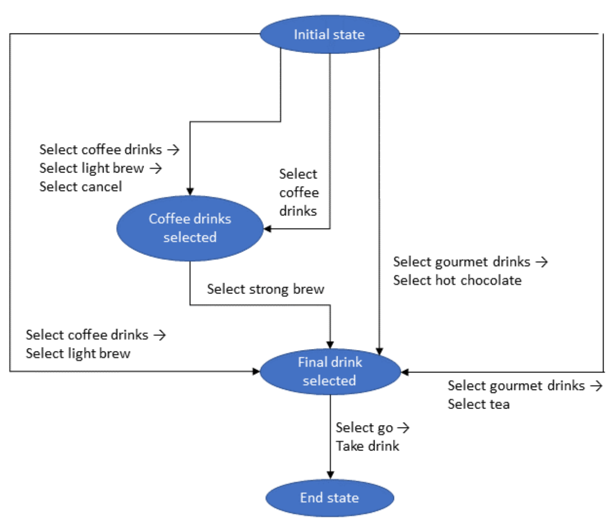


Figure 3: Simplified reverse branching tree model

2.4.2 Correcting State Redundancy

When preliminary states share behavior to operate logically similarly, they can be describing the same state. As such, they should be combined, considering that if we combine states that are not actually identical, we will add false information to the model. To avoid doing this, we acknowledge that states are largely determined by their paths in this method, so any preliminary states that share paths are functionally identical.

In forward iteration, two preliminary states sharing all their outward paths are functionally identical. Combining said states would not any additional information, making them safe to combine and simplify the model. For example, **Error! Reference source not found.** shows how t he states "Strong brew selected", "Hot chocolate selected", and "Tea selected" can be combined to form the new state "Final drink (Not light brew) selected". Note that "Light brew selected" is not considered functionally identical because it has an additional path that is not seen in the other states.

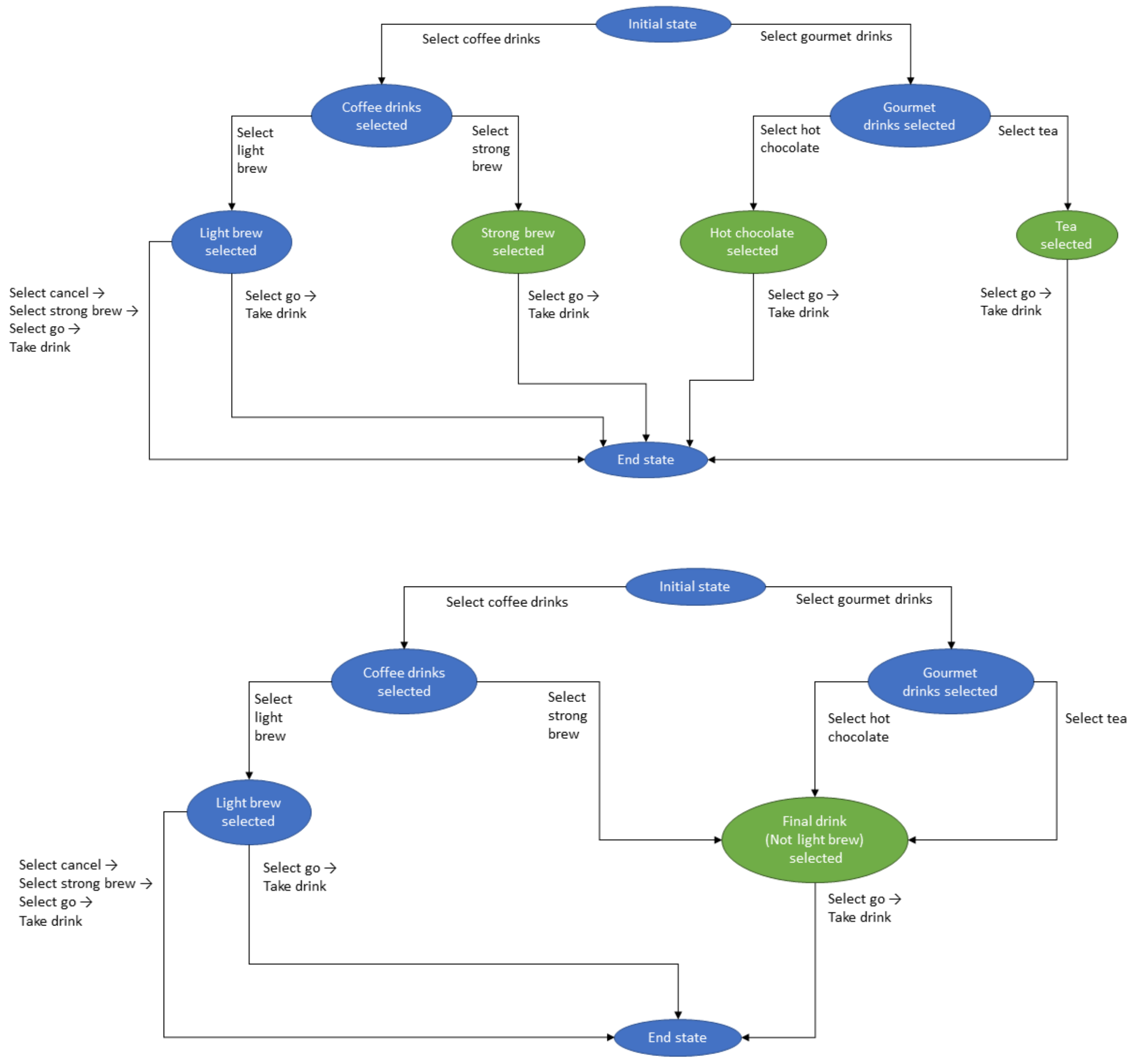


Figure 4: Forward model with redundant behavior merged. Green states in the first diagram are merged into the single green state in the second

Reverse iteration operates in the reverse direction from forward, and so states that can be considered functionally identical are instead those that share inward paths. This direction makes sense, as the same sequence of inputs should always lead to the same state. This fact also implies that states only need to share one inward to be considered the same, whereas forward iteration requires the sharing of all outwards paths to be considered identical. Figure 5 shows how this concept can be applied to a simple case.

Figure 5: Reverse model with redundant behavior merged. Green states in the first diagram are merged into the single green state in the second

2.4.3 Path Intersection

Some paths may pass through known states without being labeled as such, making the paths and state definitions inconsistent and not match the true performance. For example:

Path 1) Coffee ready to brew → Input: Go → Input: Remove cup → Ready for new drink

Path 2) Coffee in cup → Input: Remove cup → Ready for new drink

Both paths appear to share behavior. From only this information, we would conclude that path 2 is a subset of path 1, such that it would be more efficient and potentially more accurate to rewrite these paths as follows:

Path 1) Coffee ready to brew → Input: Go → Coffee in cup

Path 2) Coffee in cup → Input: Remove cup → Ready for new drink

We formalize this process of intersecting paths with two different methods, one for forward iteration and one for the reverse. Each direction places different logical demands on how we can conclude that states are passed through. Both iteration directions however share the concept of a *sub-state* to indicate the state inserted into a path between an initial and end state.

In forward iteration, we use the following set of rules to determine the presence of sub-states:

Rule 1.1) All the potential end states of a sub-state must also be potential end states of the initial state. This rule ensures that no additional connections are made beyond initial to sub.

Rule 1.2) All the sub-state to end state paths must be included exactly as part of the existing initial to end state paths. This rule ensures that the component paths from the initial and the sub-states into the end states are shared.

Figure 6 shows how these rules operate. There is a path from “Light brew selected” decomposed to pass through the sub-state “Final drink selected (Not light brew)”. For Rule 1.1, both states share end states, namely the “End state” state. Rule 1.2 is then satisfied when the sub-state path “Select go” → “Take drink” is included exactly in an initial state path. It is included in both paths, satisfying the rule.

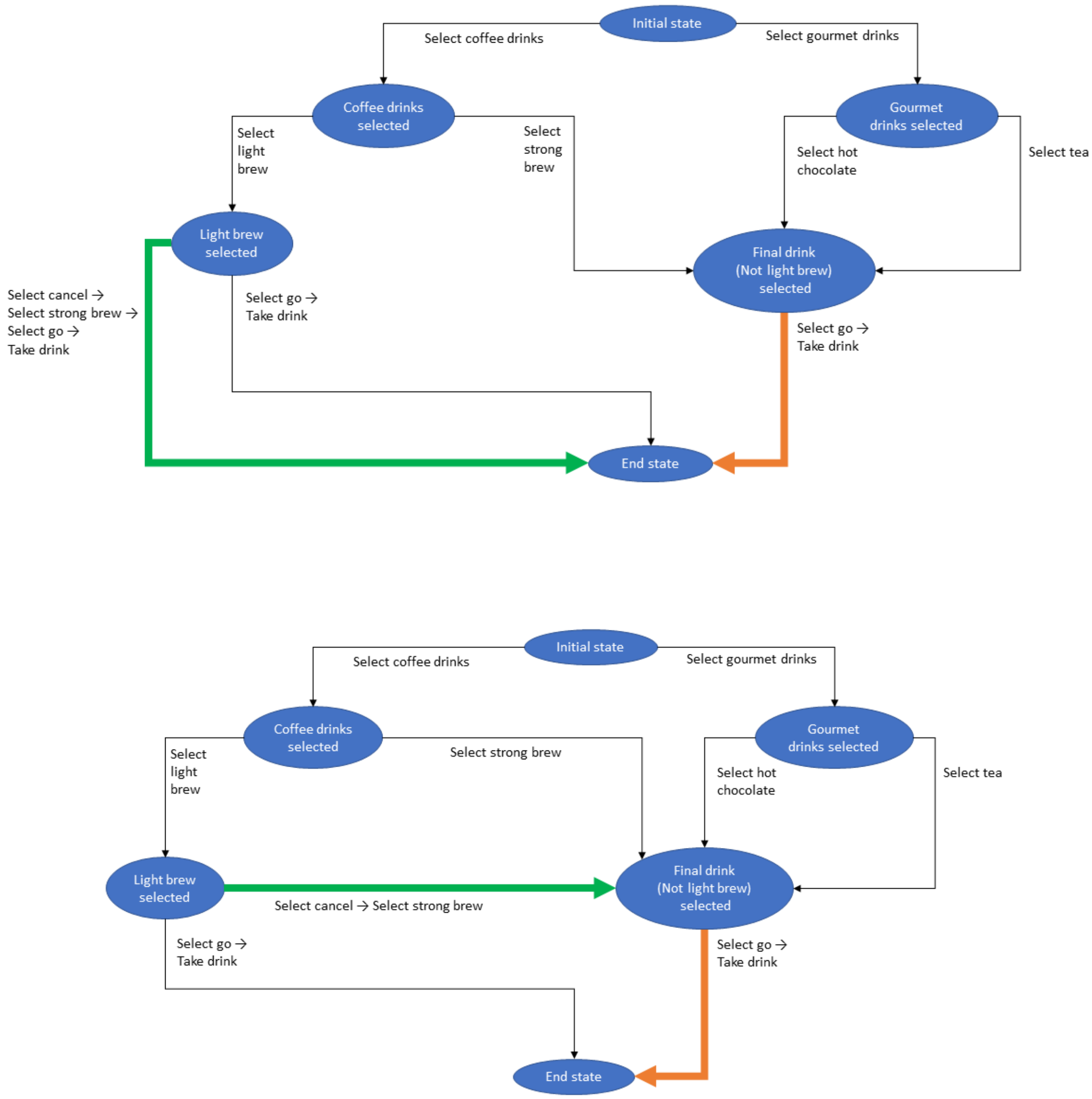


Figure 6: Forward model with path intersection. The orange path is subset to the green in the first diagram, so the green is redirected in the second diagram

This process is modified for reverse iteration:

Rule 2.1) The sub-state must be an existing end state of the initial state. This rule prevents excess connections being made.

Rule 2.2) The existing path from initial to sub-state is the beginning of another path from initial to end state. This enforces path determinism.

Figure 7 and Figure 8 show how these rules can be applied twice over, simplifying the model twice. First, “Coffee drinks selected” is made subset to a path from “Initial state” to “Final drink selected”. Rule 2.1 is observed, as a path from “Initial state” to “Coffee drinks selected” exists, and this path is also the beginning of the larger path from “Initial state” to “Final drink selected”, satisfying Rule 2.2 for path intersection. This process is then repeated, making “Coffee drinks selected” subset to its own path to “Final drink selected”. Both applications introduce recursion into the model, which would not be possible without path intersection. While the final reverse iteration model alone does not represent reality completely (Selecting cancel after ordering tea would not allow the user to selected coffee drinks) the model is more accurate in cases where recursion does occur. Additionally, this model is not to be used in isolation, and can be paired with the forward iteration model to better understand the system.

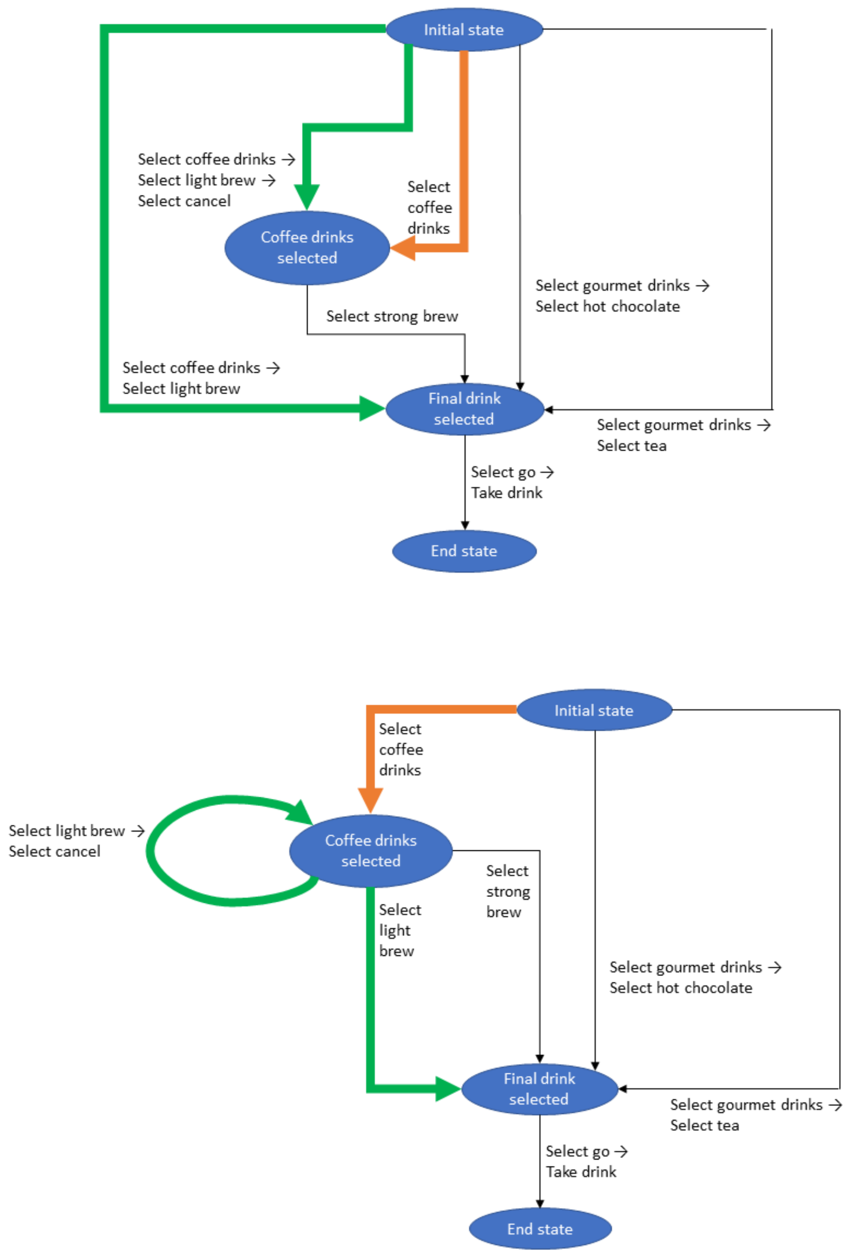
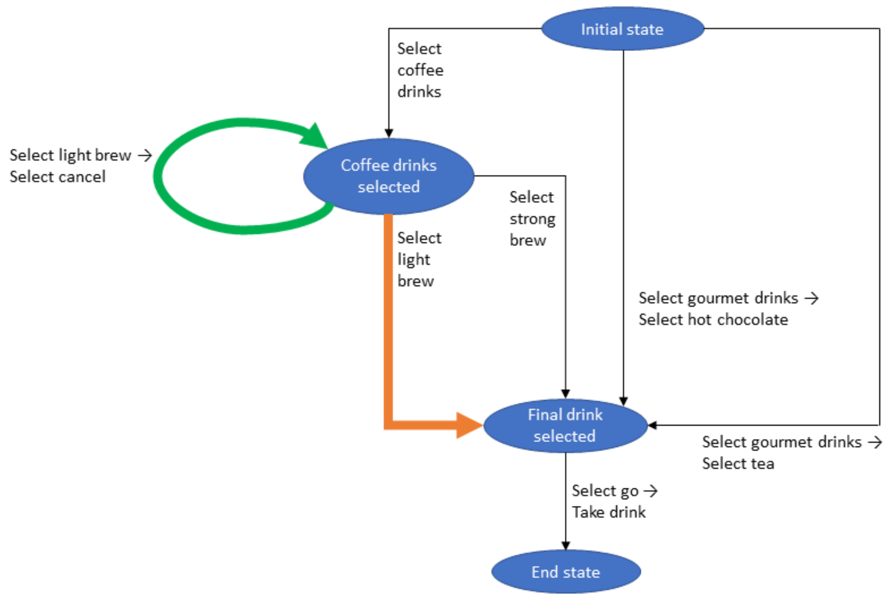


Figure 7: Reverse iteration model with path intersection. Orange is subset to green paths in the first case, so green paths are redirected



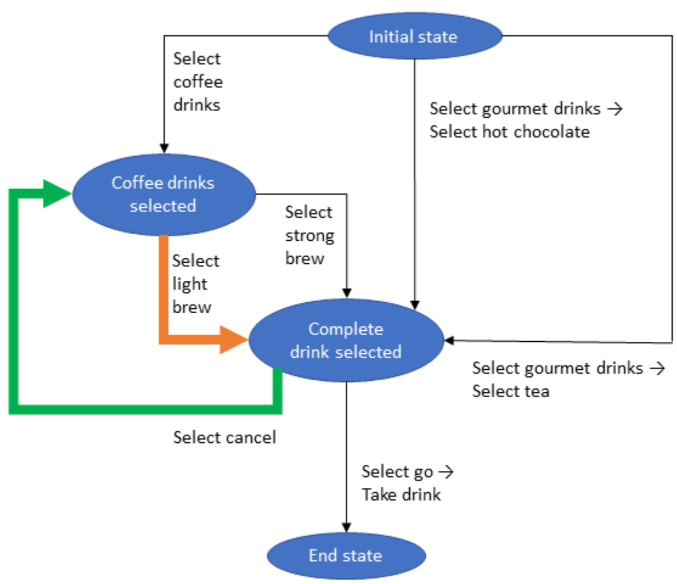


Figure 8: Reverse iteration model with path intersection applied once more. Orange is subset to green in the first case, so green is redirected

2.5 Final State Machines

The steps of state combination and path decomposition can be run iteratively, alternating between steps. Once no further simplifications can be made, the two state machines are complete. Figure 9 and Figure 11Figure 10 show the final state machines we constructed for the coffee maker, shown in forward and reverse iteration respectively. The forward iteration model exactly replicates trace behaviors, such that all paths shown are valid paths through the menu, but it is not exhaustive of all possible paths. The reverse iteration model, by contrast, includes both valid and invalid paths not seen in the trace. For example, one valid path included allowed a user to place a cup in the machine, remove the cup, and then end the interaction. Other paths included allow for recursion, which is a possibility the forward model is not capable of replicating.

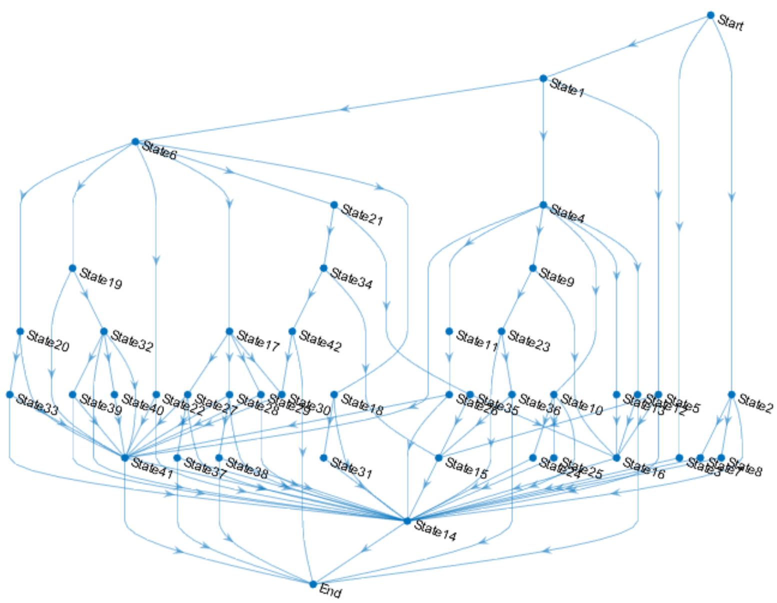


Figure 9: Final cofee maker state machine in forward iteration

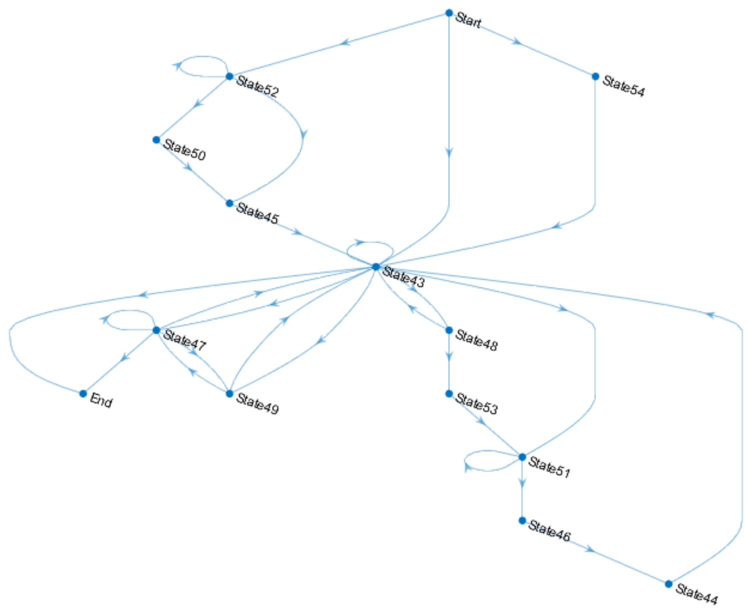


Figure 10: Final cofee maker state machine in reverse iteration

The models can also be analyzed in parallel. Note that in other simple cases, the state machines generated with the method may be very similar, but they will diverge with increased system complexity. While it may be possible to combine these machines together, it may not be efficient to do so. Parallel state machines are used for simplicity in many cases, particularly those with two disjointed tasks being performed at once. For example, we could imagine that it might be efficient to decompose the cup/coffee machine system into two subsystems (cup and coffee machine) with their own state machines, allowing us to avoid considering how the cup might affect the machine in ways beyond catching the drink at the end.

In general, we can see how the basic methodology outlined here can apply to a real system and produce a functional state machine that replicates trace behavior. Further demonstrations could also work to capture more behavior, expanding beyond simply replicating the trace, despite the lack of parameter information.

3. APPLICATION OF THEORY TO COMPLEX SYSTEM

This chapter goes into detail on the methods explored for constructing a state machine for a complex system: a flight simulator. We elected to use such a system for a demonstration of complex systems for its similarity to "real-world" safety critical systems, and for its convenience. Flight simulators provide a safe way to explore a variety of scenarios and are relatively simple to learn to operate, making them excellent candidates for study. Discussion begins with an overview of the simulator's characteristics, and then moves into how we collected and organized our trace of use. The next section centers on how we developed state detection methods with machine learning techniques, and the final section covers how these techniques fare with path and input classification.

3.1 System Overview

3.1.1 Defining the System

To begin, I selected the flight simulator YSFlight for study (Yamakawa, 2021). YSFlight presents a wide variety of benefits, first being that it is free and has low CPU and GPU requirements, making it simple to collect on any computers used, including home computers. Much of this recording needed to be performed at home due to social distancing guidelines, so this was a strong quality to have. The simulator also offers a built-in recording tool for collecting a trace. The recording tool was originally developed for replaying gameplay footage, and outputs a selection of flight data to a text file at the end of each flight. Given that this file is specialized for video replay, it unfortunately features some compression in the form of an irregular recording frequency of approximately 20 Hz that varies depending on accelerations to make video playback smooth. Nevertheless, having a built-in recorder made trace collection convenient. Table 2: YS-Flight recorded flight parametersshows the flight parameters the tool records.

Table 2: YS-Flight recorded flight parameters

Parameter	Description	Format
Time	The time after simulation start when parameters are recorded	Continuous, recorded in [s]
Inertial position (X/Y/Z)	Aircraft position vector in the simulation map, with the y parameter corresponding to altitude	Continuous vector, recorded in [m]
Attitude	Compass heading, pitch, and yaw	Continuous vector, recorded in [rad]
G-load	Unknown loading parameter	Unknown
Flight status	Categorical parameter indicating if aircraft is in flight, rolling, stalled, on fire, broken, etc.	Discrete, ranging from 1–6
Variable wing geometry	Deflection of any variable wing geometry features	Discrete, ranging from 0–255
Airbrake status	Deflection of any airbrakes	Discrete, ranging from 0–255
Landing gear position	Deflection of landing gear	Discrete, ranging from 0–255
Flap position	Deflection of flaps	Discrete, ranging from 0–255
Brake strength	Strength of brake application	Discrete, ranging from 0–255
Smoke trail status	Features of controllable smoke trail	Unknown
Vapor trail status	Features of controllable vapor trail	Unknown
Vehicle strength	Vehicle health (Used for military simulations)	Discrete, ranging from 0–4
Throttle strength	Throttle setting	Discrete, ranging from 0–99
Control surface deflections	Deflection vector of three main control surfaces	Discrete, ranging from –255–255
Thrust vector deflection	Deflection vector for thrust vectoring systems	Discrete, ranging from –255–255
Thrust reverser deflection	Deflection of thrust reverser surfaces	Discrete, ranging from 0–255
Bomb bay deflection	Deflection of bomb bay doors	Discrete, ranging from 0–255
Turret positions	Rotation position of aircraft mounted turrets	Discrete, ranging from 0–255

YS-Flight also has many features that allow it to better simulate real-world behaviors, first being a built-in air traffic control system that can provide live directions to a target airport and runway, following typical flight paths. Such directions included target bearing, altitude, and airspeed, making instructions intuitive to follow, and updates the pilot regularly as legs of the flight change or if major deviations occur. This system generally helps pilots fly more consistently and closer to realistic scenarios for safe flight.

This realism is further enabled with the YS-Flight selection aircraft and airports available. Real GA, commercial, and military vehicles are all available for flight in the simulation, as well as many airports, most of which are in Japan, the home of the game developers. Real airports allow our simulated system to mimic real flight paths, making our pilot behavior more like a real, safety-critical system operator’s than hypothetical cases. The environment can also be controlled as weather, time, and visibility conditions are all controllable in the simulation, giving additional options for testing in hazardous conditions.

Once we had settled on YS-Flight, we needed to ensure that any trace we recorded would explore the use of a single system in multiple ways, and span as much of the system space as feasible. To span the space, we decided to fix as much of the physical system as possible, beginning with the aircraft itself. I elected to record traces with a Cessna 172-R exclusively, because it is a common GA aircraft, is intuitive to learn, and has little variable geometry that might affect flight behavior, making the trace more consistent. We also elected to fix our system study to cruising flight, allowing our study to avoid the use of flaps and taxiing entirely, simplifying the space we needed to model.

The physical space we navigated could also be constrained to reduce variability in system behavior and make flights shorter and therefore easier to record. I selected two runways near to each other: Misawa airport and Hachinohe airbase in the Aomori prefecture, Japan, with a total flight time of about 15 minutes between the two. This short flight time also had the additional benefit of keeping the vehicle mass from changing significantly during the flight, which would result in aerodynamic forces acting differently on the aircraft with time, making it more difficult to approximate behavior. We reduced this issue further by fixing the initial vehicle fuel load to 75%, making each flight more consistent than a varying starting mass. To constrain the flight path between the two airports to a safe approach, we engaged the automatic ATC system and followed its directions as best as possible during trace recording. These flights were always conducted in full sun and with no wind to ensure that air velocities and inertial velocities matched, simplifying any needed approximations of physics.

To then ensure that our trace captured significant variation in system use, we recorded flights to and from each airport, using both runways in either direction. This method of capturing the system space provided numerous flight paths to examine how behavior in each path differs, as well as how total system behavior operates. We also decided that some abnormal behavior was necessary in the trace to include such behaviors in a final model. Such behavior could be included by recording the trace with an inexperienced pilot, or by altering the visibility conditions or disabling flight instruments. I had little prior piloting experience, so my unmodified flights seemed sufficient for adding variation. Our attempts to record with disabled instruments resulted in flight conditions that were so abnormal that they were difficult to interpret into a distinct cruise phase, making them too difficult to integrate into the trace to be usable in analysis.

3.1.2 Implications of the System

Flight is defined by a variety of parameters and controls, both discrete and continuous. For instance, flap deflections are set positions, making them discrete, but altitude is a continuous parameter. Controls like the brake toggle are discrete, but yoke deflections are continuous. While the simulator itself is a digital tool, all the measurements taken for the trace are technically discrete but are taken with enough precision to be treated as continuous variables. Truly discrete variables are in the minority here, and we assumed that they had a minimal effect on flight, as we restricted the system to cruising flight only, where brakes and flaps are not in use. The stall indicator remains as a discrete parameter for study, but it is exclusively used for signaling to the pilot and does not affect the system otherwise, so we excluded it from analysis to simplify the study to continuous variables only and reserved it for validation tests. Continuous variables however result in measurements that are difficult to distinguish, and thus should be rounded to a minimum precision that is expected to be relevant. This rounding requires some knowledge from the analyst, and results in a system where purely logical definitions of states and inputs like that seen in Chapter 2 are non-achievable.

Flight is defined by many parameters and controls at once, making it a parallel system. With many continuous variables in use at once, each microstate and microinput will likely see little to exact replication in the trace, removing the possibility of simple logical model. Instead, statistical definitions for states and inputs need to be generated to classify behavior.

These states are also time-dependent, as the system state constantly updates based on physics. This means that an effective system frequency needs to be determined and used as a sampling rate for trace. Different sampling rates might result in different classifications because the difference between consecutive readings decreases at higher sampling rates. So, model construction will require sampling at multiple frequencies and selecting the most effective state definitions that are consistently identify similar states despite varying sampling frequencies and parameters provided.

Lastly, it is unclear what parameters are meaningful for determining the system state. For example, is proximity to a runway meaningful? This question cannot be answered without analysis, so we can describe the system as being unbounded. Flight is bound by physics, so we can surmise that parameter vectors like position and velocity will be relevant, but reference frames and attitudes are also important to consider. Multiple variations of parameters in different references frames need to be studied and optimized to select the most effective classifiers.

3.2 System Trace

With a system defined, the system space needs to be traversed and recorded into a trace. This process begins with data collection and synchronizing microstates and microinputs. Once they are synced, we can add parameters that could not be initially recorded in the trace and perform any reference frame manipulations we might need to explore to expand the trace. Once an initial trace is completed, we can compute alternate traces with differing sampling frequencies to examine system frequency and begin analyzing the trace.

3.2.1 Data Collection

The system space can be spanned by recording multiple instances of each flight path to explore variations in execution of each path. As previously mentioned, the airports selected (Misawa airport and Hachinohe airbase) each have a single runway, which could be taken off from and landed on in either direction. Table 3: Flight paths recorded in trace shows the configurations of flights used in the final trace, and Figure 11 shows the complete record of flight from engine start to shut down.

Table 3: Flight paths recorded in trace

Takeoff runway	**Landing runway**	**Instances recorded**
Misawa RW28	Hachinohe RW25	4
Misawa RW28	Hachinohe RW07	2
Misawa RW10	Hachinohe RW25	3
Misawa RW10	Hachinohe RW07	3
Hachinohe RW25	Misawa RW10	3
Hachinohe RW25	Misawa RW28	3

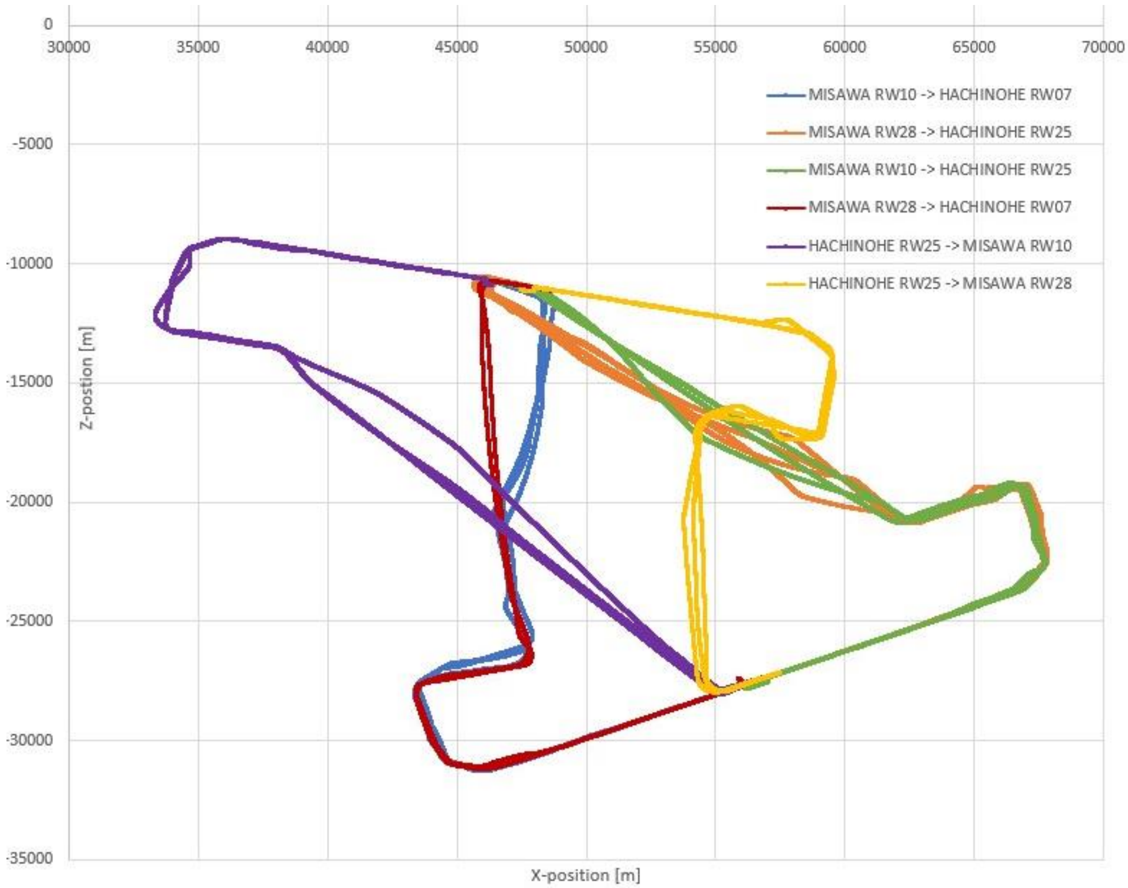


Figure 11: Top down view of flight paths recorded

I piloted the trace myself, using a Logitech Extreme 3D Pro Joystick controller, which I connected to a Simulink recording tool outputting inputs at 40 Hz. We assumed this frequency to be sufficient to capture all but the most aggressive stick inputs, which were not performed in this experiment. Given the sampling frequencies and duration of each flight, many thousands of microstate/microinput conditions were recorded, and while there are only a few instances of each flight path and no exact repetitions of microstates, similarities between each reading are likely to be high, making further repetitions likely to reinforce existing similarities. This behavior also implies that the trace approximately spans the nominal system space. Any additional flights do not seem as if they would add more information on the validity of different flights, beyond the missing flight paths.

To further expand on this trace collection, we could begin by recording more instances of each flight path, which would provide more information about behavior exclusive to the flight path but was not deemed necessary for the system wide study for the aforenoted reasons. We could also capture all runway-to-runway configurations instead of the six in the set used here. Doing so would provide an exhaustive set of flight paths, which would more likely capture the total breadth of anticipated behaviors, but was not expected to significantly improve models, as the flight paths themselves would overlap with much of the existing set. Lastly, the order in which flights were recorded could be better managed, as is, each flight path was recorded with each of its repetitions all in a row, such that the first paths recorded saw the least experience with the system. This could have biased analysis towards finding more hazardous conditions in the vicinity of the early recorded flight paths, but I did not see this bias as a significant enough factor for further study. While the initial flight recorded were slightly more anomalous than the others, the total proportion of stalls decreased only gradually with more practice. Additionally, flight paths often passed through the same flight corridors, demonstrating less that any anomalous flight was due to the physical location.

3.2.2 Synchronizing Microstates with Microinputs

Time between readings needs to be constant and consistent to effectively compare changes in microstate and state. In other words, parameter and control sampling rates should be equivalent to measure the effect of inputs on states. Unfortunately, YS-Flight records at a non-constant sampling frequency close to 20 Hz, which needs to be matched to a constant 40 Hz control sampling frequency. YS-Flight records parameters to approximate the real behavior in as few frames as possible while maintaining smooth transitions between frames. This trait implies that the true behavior can be approximated with linear interpolations between frames, further implying that the final data set can then be up sampled to a fixed 40 Hz frequency without overly distorting system behavior.

We began synchronization by finding an initial reading for both parameters and controls. YS-Flight sims start by selecting conditions and then loads to a starting window that begins recording with any further inputs. As such, recording can then proceed through the following steps:

1. Prepare flight weather/vehicle/starting location conditions
2. Start control recording
3. Set YS-Flight to the simulation start screen
4. Input a single, Boolean command via the control stick
5. Set up ATC

6. Fly normally

The step four Boolean command was issued through an unused control input on the stick, the trigger. Thus, the first trigger reading in the control recording is simultaneous with the first recorded parameter reading. This establishes a uniform time zero. Once the two data sets had a synchronized start, we linearly interpolated each parameter value to match the control reading time, producing a synchronized 40 Hz record of parameters and controls.

3.2.3 Additional Parameters and Transformations

Each system state may be defined by parameters that are not directly recordable, suggesting that known parameters that fit this description should be added to the trace when possible. In this case, YS-Flight does not record velocities, which presumably affect state, so velocities need to be calculated from the existing trace if possible. Failing to include all the system relevant parameters will result in the system state being undetermined.

In this case, velocity can be extracted from the existing record using position and sampling frequency to calculate a distance moved per unit time. For this analysis, we conceptualized velocity as being determined from the current position and position in the immediately preceding frame. This concept matches with the rest of data presented in the trace as being an instantaneous measurement.

However, frame-by-frame analysis can result in "jitter" due to precision loss in the parameters. For example, a slow-moving aircraft in our system could see zero velocity for several readings and then a sudden spike in velocity for one reading, followed by zero velocity. This cannot be completely corrected, as the true behavior is not recorded in YS-Flight. The best option then to reduce jitter is to smooth velocity, using the average calculated velocity for a given time frame. In this case, we opted for a 0.125 s smoothing window, centered on the reading being updated, that averaged the five readings within the window. Longer smoothing windows would result in a loss of high-frequency velocity changes, which are largely correlated with anomalous behavior because most of the nominal flight is conducted through with low accelerations for safety. Shorter smoothing windows do not meaningfully reduce velocity jitter, making this window size effective for this application. Additionally, because system is in cruise, high speeds will balance out most precision issues in position data, making smoothing less necessary that low-speed applications, but still required to improve performance.

Other parameters may need to be added to the set with reference frame transformations. When applied well, reference frames can be used to produce alternative parameter sets with less variation between flight paths, making them more effective for characterizing overall system behavior. In our existing trace, inertial position alone provides an incomplete view of the system, mostly emphasizing information on system-wide valid flight paths, but it does not provide information on valid flight paths for specific runway configurations, and obscures information on the mechanics of flight itself.

To combat this issue, we transformed the inertial coordinates to runway-relative reference frames. This added two reference frames to the trace, one for takeoff and one for landing, providing a total of five positional parameters. Each frame was centered on the runway of interest, ran one axis in the direction of use, preserved the vertical altitude axis, and ran the third axis in the transverse direction of the runway.

We transformed velocities similarly to match the positional frame, with the additional information that the physics of flight are largely defined by aircraft relative velocities, suggesting that a third frame be used. We then placed this third frame in the aircraft relative orientation, with one axis point along the forward axis of the vehicle, one on the vertical, and one on the horizontal axis. With the three velocity reference frames, two sharing a vertical axis, we brought the total number of velocity parameters up to eight.

Other parameters, like compass heading, similarly lack consistent meaning from one flight path to another. North is held consistent with inertial coordinates and could help with identifying valid flight paths, but in the runway-relative frames, North is not consistent. Instead, I used a target-relative heading, using the direction to center of the landing runway as “North”. This alternative helps enforce cruising generally towards the landing runway.

Attitude in general presents some issues for analysis because it is a vector of angular parameters. Angular parameters that can rotate fully skip from 359 degrees to zero degrees which statistical modeling techniques will have difficulty modeling. Instead, we took the sine and cosine of angular parameters and split compass heading into two parameters, removing the discontinuity.

Some system behaviors may have time-delayed effects, which could require additional parameters to capture. For example, aircraft flaps have several set positions in YS-Flight and can be controlled by pressing a corresponding button to initiate extension or retraction. This change in position does not occur immediately, so a parameter and control set that only captures the current flap position and current flap control input would be unable to determine what the next flap position would be. This issue would not be corrected with the inclusion of a flap velocity parameter either, because no information on how many times the extend/retract command has been input is stored. Potential fixes would be to use a dedicated parameter for tracking commanded flap position, or in the case of system behaviors that execute after a passage of time, a "time since input *x*" parameter. Because we are operating exclusively in cruise, where flaps are not in use, such time-delayed effects are not a concern, but the issue could be relevant in other systems.

Once we had selected all the parameters, I encoded each parameter to condense discussion Table 4 shows the final parameter set, including the parameter encodings. Note that the stall indicator parameter is a Boolean variable, and therefore cannot be used in conjunction with the other continuous variables using statistical methods. As previously discussed, its inclusion in the trace was useful for validation of analysis.

When discussing the parameters in this analysis, it is useful to also develop a concept of parameter frequency. Here, I will use this term to qualitatively refer to how quickly a parameter is likely to change its value in a meaningful way. For example, we could consider most position parameters as being low-frequency parameters, because they change their value very little between readings in most cases. On the other end, control surface deflections and throttle strength could be considered high frequency, because even in non-hazardous conditions, they may change their value significantly relative to the recording frequency.

Table 4: Total parameter set used in trace

Parameter	Code	Description	Freq.	Unit
SIN(Compass heading or Target heading)	SINCH/ SINTH	Sine of the corresponding bearing variable	High	
COS(Compass heading or Target heading)	COSCH/SI NTH	Cosine of the corresponding bearing variable	High	
Pitch angle	PA	Angle between the aircraft longitudinal axis and level flight	High	rad
Bank angle	BA	Angle between the aircraft wing and level flight	High	rad
X-position (Takeoff)	XPT	Distance between the aircraft and the center of the takeoff runway in the direction of takeoff	Low	m
Y-position (Inertial)	YPI	Aircraft altitude relative to sea-level	Low	m
Z-position (Takeoff)	ZPT	Distance between the aircraft and the center of the takeoff runway in the direction of the runway transverse	Low	
X-position (Landing)	XPL	Distance between the aircraft and the center of the takeoff landing in the direction of landing	Low	m
Z-position (Landing)	ZPL	Distance between the aircraft and the center of the landing runway in the direction of the runway transverse	Low	m
X-velocity (Takeoff)	XVT	Aircraft velocity in the direction of takeoff	Low	m/s
Y-velocity (Inertial)	YVI	Aircraft climb velocity	Low	m/s
Z-velocity (Takeoff)	ZVT	Aircraft velocity in the direction of takeoff transverse	Low	m/s
X-velocity (Landing)	XVL	Aircraft velocity in the direction of landing	Low	m/s
Z-velocity (Landing)	ZVL	Aircraft velocity in the direction of landing transverse	Low	m/s
Forward velocity (Plane)	FVP	Aircraft forward velocity	Low	m/s
Vertical velocity (Plane)	VVP	Aircraft vertical velocity	Low	m/s
Horizontal velocity (Plane)	HVP	Aircraft horizontal velocity	Low	m/s
Throttle strength	T	Throttle setting on a scale of 0-100	High	
Elevator deflection	CSE	Elevator deflection from -256-256	High	
Aileron deflection	CSA	Aileron deflection from -256-256	High	
Rudder deflection	CSR	Rudder deflection from -256-256	High	
Stall indicator	S	Truncation of the original “Flight status” parameter, showing one if stall has occurred and zero otherwise	N/A	

3.2.4 Examining System State Frequency

The rate at which the system changes state is unknown and needs to be explored. Low sampling rates will not capture fast changes in state and will instead overemphasize the effect of high frequency parameters. For example, instantaneous control surface deflections would generally do little to affect the system state, but sustained deflection would. A trace with a low sampling rate recording a pilot rapidly oscillating the elevators would not be effective for matching the elevator deflection to aircraft motion. However, this effect would have the positive outcome of making microstates behave like random, independent samples, which is necessary for performing statistical analysis.

In the opposite case, high sampling rates can bias state classification methods towards low frequency parameters. At these rates, parameters that change with low frequency will have values that are closer together than those that change at high frequencies because they change value slowly. This makes the trace readings more visibly dependent on each other and less useful for statistical methods of analysis. Additionally, this effect biases classification methods based on group densities, which are common and generally effective tools, towards using the high density and low-frequency parameters, obscuring the true system behavior. Microstates that are close in value are more difficult to separate into mutually exclusive states, making the final classifications less meaningful.

Overall, these effects suggest that multiple sampling frequency traces need to be constructed and examined to determine whether there is consistent behavior across multiple frequencies. This consistent behavior would then indicate a true system frequency. To do this, we elected to test 40 Hz and 4 Hz traces. 40 Hz was the highest possible frequency we could reliably capture data with the recording controls, and 4 Hz is much lower, but not so low as to not catch high-frequency transitions like a dive due to stall. No recovery attempt post stall would result in normal flight parameters at this sampling frequency, so it would still be able to catch basic behavior.

There are two basic methods for converting the baseline 40 Hz to 4 Hz. The first option is to down-sample the set by picking every tenth point of the 40 Hz set, which would result in measurements as if the system had been originally sampled at 4 Hz. The second is to arithmetically average each 10-reading segment of the trace into a single reading. This process would lower the apparent frequency of all parameters by averaging values but would affect high frequency parameters the most. Overall, this would bring parameter frequencies closer together, reducing the high-frequency bias, and making it the preferable choice for downsampling.

3.3 State Detection through Unsupervised Machine Learning

With a complete trace, we can analyze microstates to produce state descriptions. Several methods already exist to classify vector data into classes, which we can use to approximate state descriptions. Methods that seek to produce the ground truth we are seeking are called "unsupervised" methods.

Most simple classifiers treat the microstate as spatial coordinates for points in the state space (Boonchoo, et al., 2019). Such coordinates would have *n* dimensions, with *n* being the number of parameters. The raw parameters in the trace have dramatically different magnitudes and need to be normalized to ensure that each parameter is weighted evenly in our statistical methods. With a normalized trace, the classifier can then examine the distribution of points and use different methods to partition the space into states.

In general, classifiers require some tuning to produce meaningful results, so they often require several attempts at classification before a final model is achieved (IBM, 2020). For this case, we will not only need to tune the classifier variables to our system, but we also need to determine which parameters are useful for describing states. I constructed an optimizer tool described in the next section to tune these settings and produce the most reasonable state descriptions possible. Once we had state descriptions, I validated them against known conditions like stall and other clear transitions in behavior, variations in trace reference frames, and variations in sampling frequency.

3.3.1 Classifier Selection

We considered three classifiers that are commonly used for generating states: DBSCAN, K-means, and Gaussian-Mixed-Models (IBM, 2020).

DBSCAN (Density-based spatial clustering of applications with noise) operates by clustering microstates by proximity, using two metrics: minPts and search radius (Mathworks, 2021). We can describe it as following this set of rules to determine states:

1. Each parameter is normalized to weight changes in magnitude as equivalently as possible
2. All microstates within the search radius of each other are neighbors
3. Each microstate counts the number of neighbors
4. Microstates with at least minPts neighbors become core points, starting a new state
5. If a microstate is neighbors with a microstate in a state, it can also be said to be in a state
6. Microstates with no neighbors are considered to be outliers

Figure 12 shows these rules graphically, where red indicates core points, yellow indicates non-core points in the same state as red, and blue indicates outlier points that not classified.

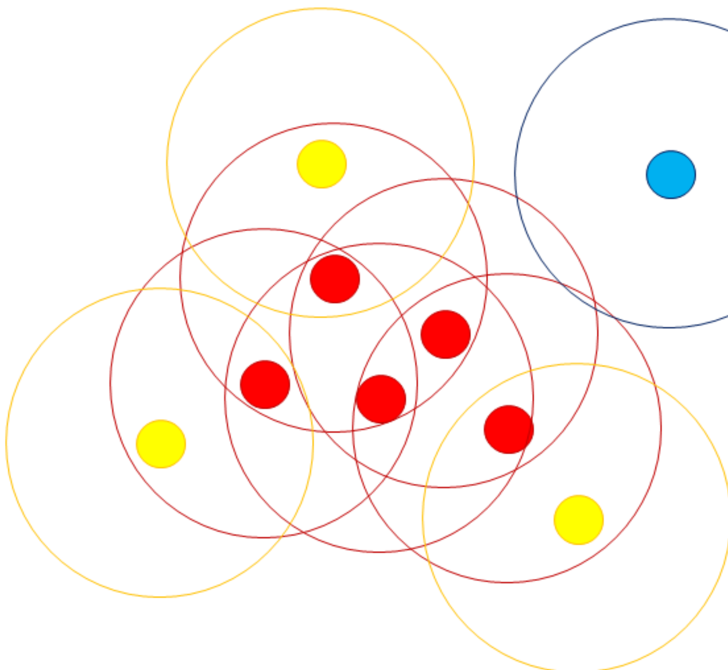


Figure 12: DBSCAN visualization

DBSCAN offers two main advantages (Boonchoo, et al., 2019). First, unlike other classifiers, DBSCAN does not require an analyst to specify the number of states to classify data into. This reduces the amount of system knowledge required to optimize, and the two tuning variables have established techniques for estimation. Second, DBSCAN can classify states that have arbitrarily shaped perimeters. Many classification methods struggle with state definitions that create concave shapes, especially when they leave the centroid of the structure outside the perimeter. Figure 13 shows a potential real-world case of this concavity in data sets, where coordinates are being using to classify flight into a safe zone and an unsafe mountainous zone. If a classification method that cannot handle concavity is used, it might generate state boundaries like the ellipsoids marking the map. Such definitions can lead to ambiguity over whether the intersection is a safe flight zone, and in this case, the centroid of the safe flight state definition is completely outside the true border.

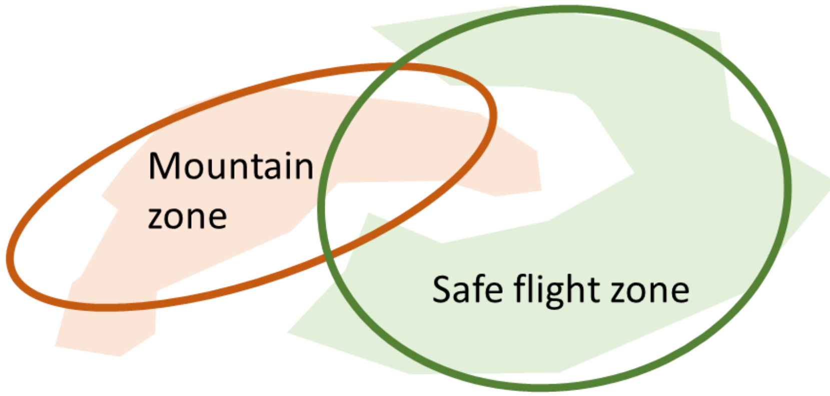


Figure 13: Demonstration of how concavity can affect classification. Here, the green and orange filled areas are being defined using a technique based around ellipsoids (GMM)

On the other hand, DBSCAN is less effective than other methods with higher-dimensional data sets, finding more arbitrary definitions than other methods in high dimensions (Boonchoo, et al., 2019). This is due to an effect referred to as the curse of dimensionality, where adding a new dimension to data exponentially increases the hyperspace of the set (Köppen, 2000). This increases the random odds that any point is within a given distance of any other, increasing the likelihood of random "order" appearing in data defined by proximity. Some alternative measures of distance, like city-block, can help with the issue, but in general DBSCAN handles this perceived order worse than other classifiers. Additionally, each parameter is weighted the same for each point in each state, which has the side effect of making each state tend towards having the same minimum density. In general, despite the established techniques for tuning, finding optimized values for the tuning metrics can be difficult to approach, given that they affect each state uniformly.

For our specific application, DBSCAN is also unsuited for our task because it requires all points to be independent samples. Any dependence leads to separate readings being very geometrically close, and hence difficult to distinguish. Random sampling readings from the trace can combat this but given that we expect traces to sample states unevenly, this can lead to complications. Lastly, DBSCAN only classifies specific sets of points together into states, and requires further analysis to produce state definitions, while other classifiers do not.

The K-means classifier works to define *k* states, with new microstates being classified into states based on which states they are closest to the mean value of all its constituent points. This is done by assuming that microstates can be separated partitioning the hyperspace with hyperplanes, drawing a clear boundary between states (IBM, 2020). To place these planes, first the analyst must provide one tuning metric: the number of states to establish. Then, they place hyperplanes in the state space, attempting to maximize the density of each state, by minimizing the total variance of the microstates in each state. Figure 14: K-means visualizationFigure 14 shows how a K-means classifier might separate 12 microstates into three states, with each color indicating the final state of classification.

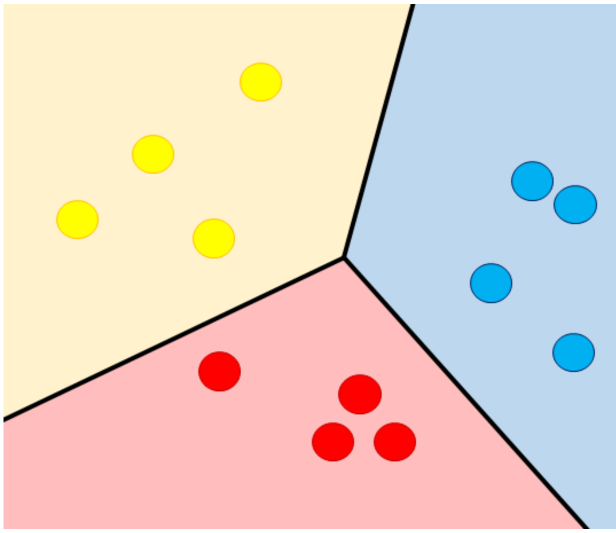


Figure 14: K-means visualization

This classifier offers several advantages over DBSCAN, first and most obviously, that only a single tuning metric is needed. K-means can also interpret dependent data better than DBSCAN, because it does not classify all similar states as identical, reducing the effect of the curse of dimensionality. This method also establishes clear, geometric definitions of states, allowing new readings to be classified easily without additional interpretation.

However, DBSCAN is slightly preferable in some regards. Placing the hyper-planes is computationally difficult and requires many iterations. K-means state definitions tend to trend towards spherical states that are roughly equal in size, as hyperspheres generally have the lowest variance. This geometry is not necessarily how states are structured and distributed however, we can expect that a trace will exhibit states at different rates and shapes.

The last method we considered was the Gaussian-Mixed-Model (GMM) classifier. It is often considered as a direct improvement on K-means classification, as they both used iterative methods to minimize variance in classifications, but they approach the problem differently (McGonagle, Pilling, & Dobre, 2021). As with K-means, GMM uses state count as its sole tuning variable. GMM then follows the following set of rules:

1. Each parameter is normalized to weight changes in magnitude as equivalently as possible.
2. Within each state, each parameter is assumed to follow a normal distribution.
3. Each state can then be described with a set of mean, variance, and covariance values for each parameter.
4. Statistical tools can estimate reasonable values for each metric from the trace.
5. Microstates can then be assumed to be randomly produced by each state model, with a probability of generation provided.
6. Each microstate in the trace can be classified to the highest probability state.

Figure 15 then shows how we could visualize probability curves for different states and how those curves could plausibly generate corresponding microstates.

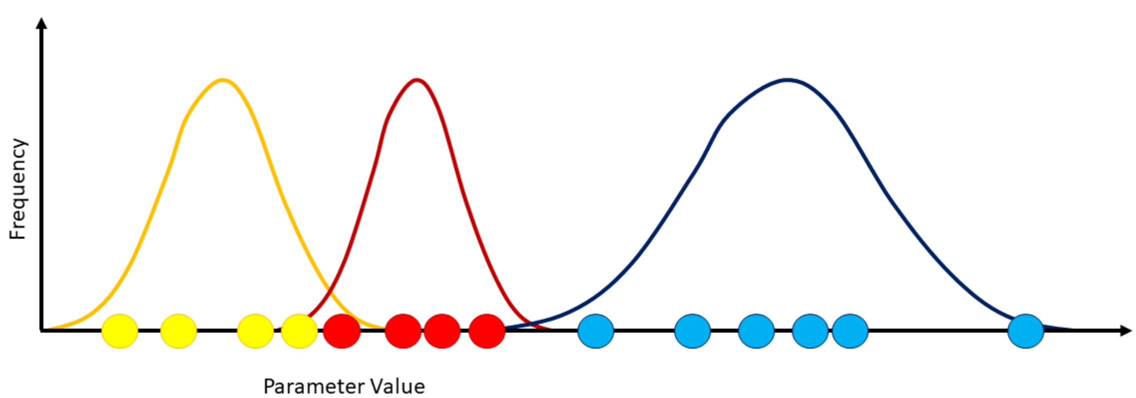


Figure 15: Gaussian Mixed Model visualization

This final method is preferable for our application, because it can manipulate dependent data sets well, like K-means, but has more flexible geometric constraints for states. While it still cannot produce concave state perimeters, it lacks some of the major downsides of K-means, namely its restrictions on size and shape. Additionally, GMM provides confidence ratings for each classification, providing another value for an analyst to compare states and microstates on the periphery of each state.

3.3.2 Classifier Optimization

While Gaussian Mixed Modeling can classify microstates into states, it still requires tuning, with the additional need to compare importance for determining states with different parameters. I considered tuning the GMM to be an optimization problem, where differently tuned models could be compared to maximize the quality of classifications produced.

We can think of classification quality as being composed primarily of two metrics: distance between states, and density of states. Classifications that produce states that are generally well-separated show that their states are well-defined and distinct. Classifications that then produce states that are dense, show that their states show many readings with similar behavior, and are well represented in the data set.

Classification methods then use at least one of three criteria to measure the combined effects of distance and density: silhouette, Davies-Bouldin, and Calinski-Harabasz (MathWorks, 2021). The silhouette criterion is defined with mean distances between microstates in a state compared to the mean distances of microstates in the next closest state. This method produces a metric that is bounded, making it easy to interpret, but it is computationally complex, because it requires distance calculations between each point in the set.

The Davies-Bouldin index is akin to the average similarity between states, primarily relying on the state centroids. This process is less computationally expensive than silhouette but restricts analysis to Euclidean space. This can be an issue if classifiers use non-Euclidean distances to determine classifications. This is not the case here, but it can be restrictive for some classifiers. The Calinski-Harabasz index by comparison, uses matrix comparisons of microstate dispersions inside states and between states, avoiding using the centroid and distance, making it preferable to both other options.

To then optimize models compared with the Calinski-Harabasz index, I used genetic optimization. In general, we can think of this method as using different optimization variables as genes for many separate model tunings. In this case, these variables would be our state count, since we are using GMM, and Booleans indicating which parameters should be considered relevant for determining state, each encoded as a gray binary number. The optimizer then generates many random configurations of genes and evaluates them with a fitness criterion (in our case, the Calinski-Harabasz index). Individual configurations that performed worse in the bottom 50% are culled from the population, and the surviving configurations are randomly paired to swap gene values, with some degree of random mutation. Each pairing produces four "child" configurations, establishing a new population for a second iteration. In this way, we produce an artificially evolving system, with genes for configurations that perform well being retained in the population. Once 90% of the genes in the population are identical, the optimization finishes, and the best performing configuration is output as the optimized solution.

This optimization scheme works well in our case, as many optimization techniques are restricted to continuous variables, whereas our variables are all discrete or categorical. It also works well if even if there are multiple local minimums in the optimization space. We can assume multiple local minimums exist in this set, as it is plausible that certain parameter inclusions will have different optimized state counts, especially with the high dimensionality of the trace, so this is a trait that is desirable for our optimization.

To set up this optimization, each variable used must be bounded and encoded into binary. I bounded the state count from 3 to 18, deeming that fewer than three states would provide no more information than existing anomalous state identification techniques, and more than 18 states would result in definitions so fine-grained that they may not be intuitive to distinguish, and therefore difficult to validate. This range then holds 16 possible values, keeping the number of binary digits required to represent the data as small as possible to reduce the volume of data needed to optimize, improving turn-around rates for diagnostics.

Parameter inclusion was represented as a gene by assigning each parameter a gene of value zero or one, indicating its inclusion in the GMM model. However, allowing all parameters to be enabled and disabled in the optimization could result in well-defined states that had little to do with the safety critical performance. For example, we would imagine that a state classifier that does not consider the rate of climb/descent for the aircraft would not be meaningfully considering the system safety and could instead be classifying irrelevant parameters. This issue is a direct result of the system being unbounded.

To adapt to this problem, we assumed that some parameters are relevant for determining states, so they are not included as genes in the optimization but are always used in the classifier.

These parameters, Y-velocity (Inertial), plane relative velocity, and the throttle setting, are strongly correlated with the physics of flight and flight safety.

With these pieces in place, we began optimization. Our initial test case used 4 Hz sampled data and compass heading as the bearing variable and provided three coherent states. Table 5 lists the means for each normalized parameter included in the optimization. Most parameters show a clear separation of means for one state, providing some simple information on what distinguishes each state from the others. For example, none of the mean parameter values of the first state are outliers, suggesting that this is close to a baseline, and can be primarily defined by its contrast with the other states. The second state has high SINCH, ZVL, FVP, VVP, and T means, and a low ZPL mean. With many high velocity parameter means as well as a high throttle setting, we can assume that this state will largely be characterized by its high speed. The third and final state can then be contrasted with low YVI and HVP means, suggesting that this is a dive state, and likely an uncontrolled dive given the HVP value. Overall, we can now view each state as the low-speed, high-speed, and hazard state respectively.

Table 5: Compass heading 4 Hz normalized parameter means

Name	**SINCH**	**COSCH**	**ZPL**	**YVI**	**ZVT**	**ZVL**	**FVP**	**VVP**	**HVP**	**T**
Low-speed	–0.10	0.14	0.13	0.03	–0.01	–0.12	–0.38	–0.37	0.02	–0.36
High-speed	0.28	–0.35	–0.40	0.02	0.03	0.34	1.04	1.05	0.02	1.02
Hazard	–0.17	–0.15	0.31	–0.74	0.07	–0.23	–0.16	–0.40	–0.41	–0.38

3.3.3 State Validation against Known Behaviors

With a state model in place, we can classify microstates in the trace and examine their behavior to see if the state definitions result in coherent behavior. Figure 16 shows the top-down view of the recorded flights in the landing-runway relative frame. There is a clear delineation of the low-speed and high-speed flight states as the aircraft transitions from flying towards the runway to lining up for approach. Figure 17 then shows the recorded altitudes in order of instance, with the hazard state showing up disproportionately in areas of rapid descent. Both inspections suggest that our definitions are coherent.

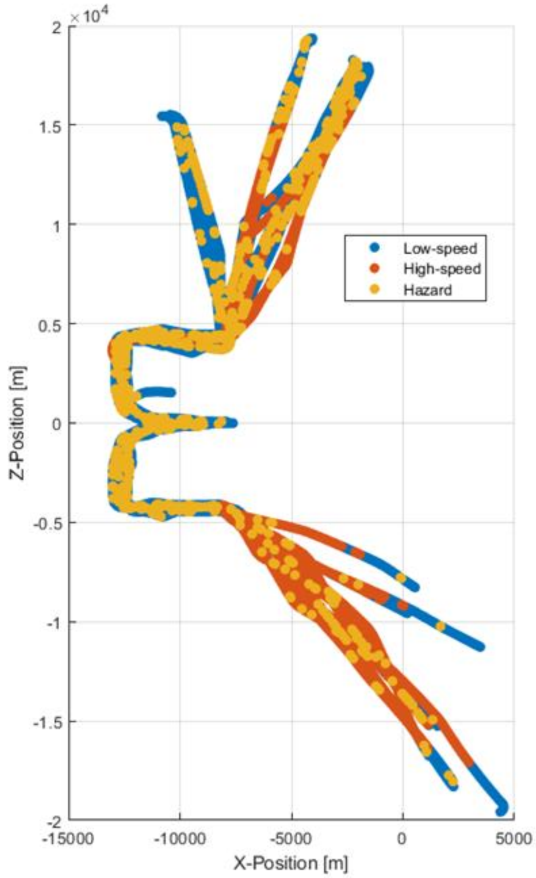


Figure 16: Top-down view of initial classification using landing runway relative coordinates

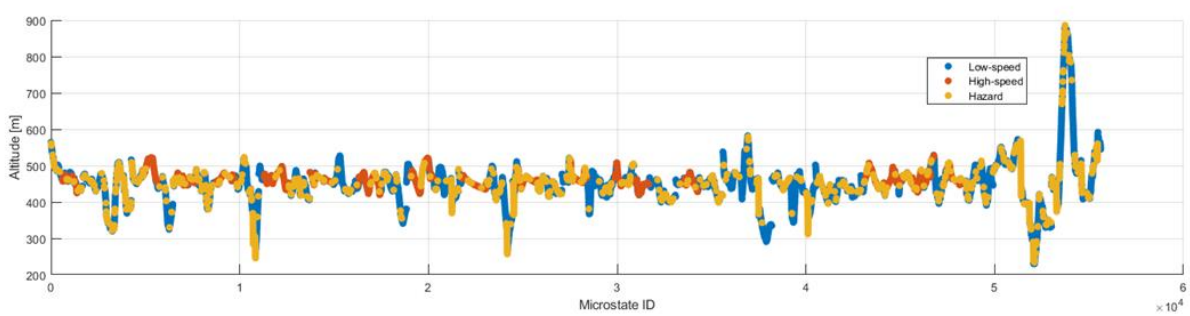


Figure 17: Altitude readings of initial classification in order of appearance in trace with state indication

As an additional check, we compared the distribution of hazard states to the distribution of stalled microstates. 75% of stalled microstates appeared in the hazard state, with the remaining 25% appearing exclusively immediately before state transitions into the hazard state, with about a half second lag time. This makes sense, as short periods of stall will not dramatically affect flight parameters, but sustained stall will. Overall, checking the initial data set against the perceived behavior and the stall characteristics supports that our states have some grounding in reality.

3.3.4 State Validation through Parameter Variation

To provide further grounding, we assume that true system states will have definitions that can be found with this optimization process even when the initial parameters available are changed. To then test if our initial definitions exhibit this property, we altered the initial parameter set and reoptimized the system in two separate ways.

First, we altered the reference frame of the compass heading. As is, the compass heading variable is somewhat arbitrary outside the inertial reference frame. In a runway relative frame, North is inconsistent from flight to flight, so instead, we rotated the compass reference frame to always point North towards the center of the target landing runway, producing the target heading parameter, which is converted with sine and cosine as before into SINTH and COSTH respectively.

With compass heading replaced, I optimized the model again, and produced a second set of state definitions. When comparing the states generated, we can look at the parameters shared in both definitions, and their extreme means. Table 6 shows the normalized parameter means as before, and we can see similar trends in behavior, with the shared extrema highlighted, green corresponding to shared high values and red corresponding to shared low values. No extrema disagree, suggesting that these definitions are defining the same states, like we would expect of a true system state.

Table 6: Target heading 4 Hz normalized parameter means

Name	**SINTH**	**XPT**	**XPL**	**YVI**	**XVL**	**ZVL**	**FVP**	**VVP**	**HVP**	**T**
Low-speed	–0.15	0.01	–0.58	0.06	0.21	–0.05	–0.61	–0.58	0.02	–0.59
High-speed	0.25	0.01	0.80	–0.01	–0.31	0.09	0.87	0.85	0.02	0.85
Hazard	–0.16	0.17	0.00	–0.68	0.21	–0.15	–0.13	–0.38	–0.68	–0.32

When we examine the states by comparing them to known behaviors as before, we see similar performance to the compass heading case. Figure 18 shows the top-down view, where we can see a similar transition from a mix of all three states to exclusively low-speed and hazard once we transition into approach. Interestingly, some flight paths appear to have been completely reclassified from low-speed to high-speed, while others have been reclassified in the opposite manner. In Figure 19, target heading shows similar behavior to the compass heading for altitude plots, and when we compared stall inclusion, we saw the same 75% in hazardous split, suggesting that these state definitions are describing similar phenomena in the system.

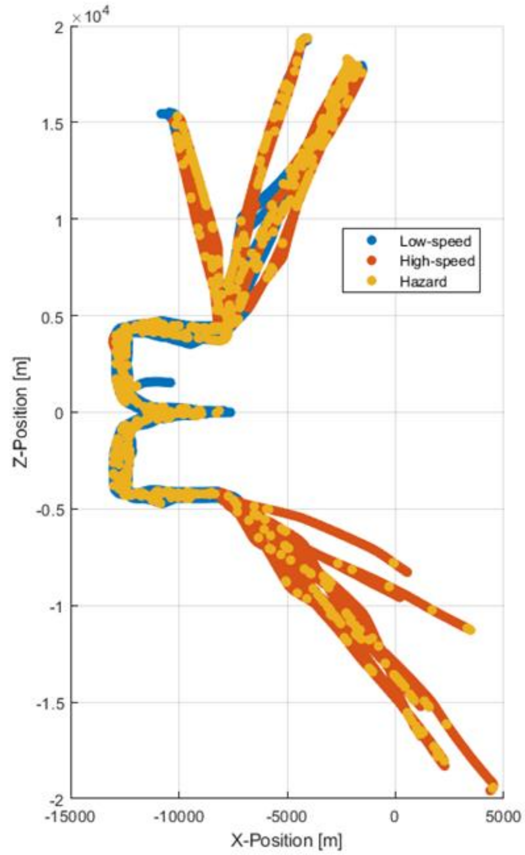


Figure 18: Top-down view of target heading optimization using landing runway relative coordinates

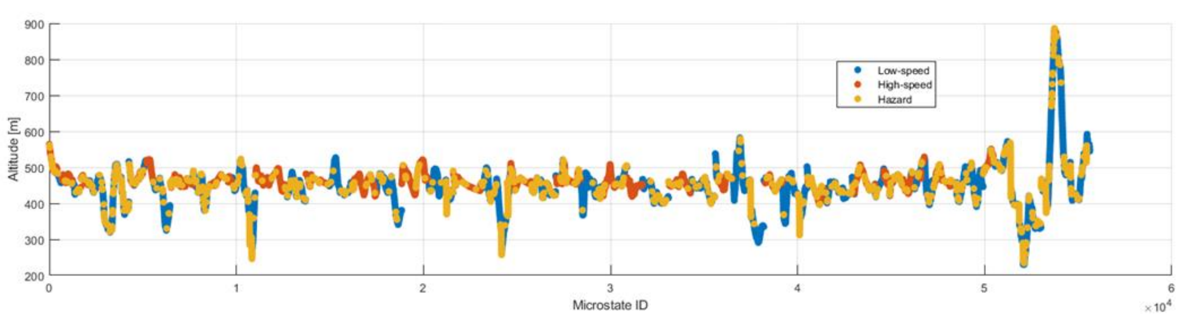


Figure 19: Altitude readings of target heading optimization in order of appearance in trace with state indication

The evidence in both cases suggests however that the parameters included may be masking the true system behavior. Both optimizations included parameters in their definitions that have no clear connections to behavior. For instance, XPL and ZPL are both position parameters indicating proximity to the landing runway. Proximity might affect when a pilot might alter their speed or course, but it would not physically alter the vehicle state, suggesting correlation with state, but not causation. This masking issue is further reinforced when entire flight paths appear to change state from compass to target heading optimizations, but the actual speeds flown remain the same. To then develop a more consistent definition, we excluded position and runway-relative velocities from the optimization and produced a third set of definitions.

As before, the normalized mean values retain their extrema in the parameters used in all three definitions, as seen in Table 7, where green and red once again indicate conserved extrema. Interestingly, without the position parameters, the optimization instead includes pitch angle, bank angle, and elevator deflection to define states. This is technically a less optimized definition, as the Calinski-Harabasz index of the positionless optimization is the lowest of the three performed so far, but these new parameters have much more obvious causal connections to states. For example, the low-speed state has a high pitch angle mean relative to the other states, suggesting that flying at low speeds requires flying at a higher angle of attack to stay in level flight, as we would expect.

Table 7: Positionless 4 Hz normalized parameter means

Name	**PA**	**BA**	**YVI**	**FVP**	**VVP**	**HVP**	**T**	**CSE**
Low-speed	0.31	0.00	0.04	-0.41	-0.39	0.02	-0.39	0.36
High-speed	-0.74	0.02	0.01	1.06	1.05	0.02	1.03	-0.96
Hazard	-0.09	-0.19	-0.64	-0.29	-0.51	-0.33	-0.52	0.52

The top-down plot in Figure 20 shows that like the other optimizations, the same general regions each state occupies are preserved, with slightly more transitions from low to high-speed states. Overall, we can conclude that despite the new parameters, the state definitions we have reached are consistent in roughly which microstates belong to which state, and how those states look and behave.

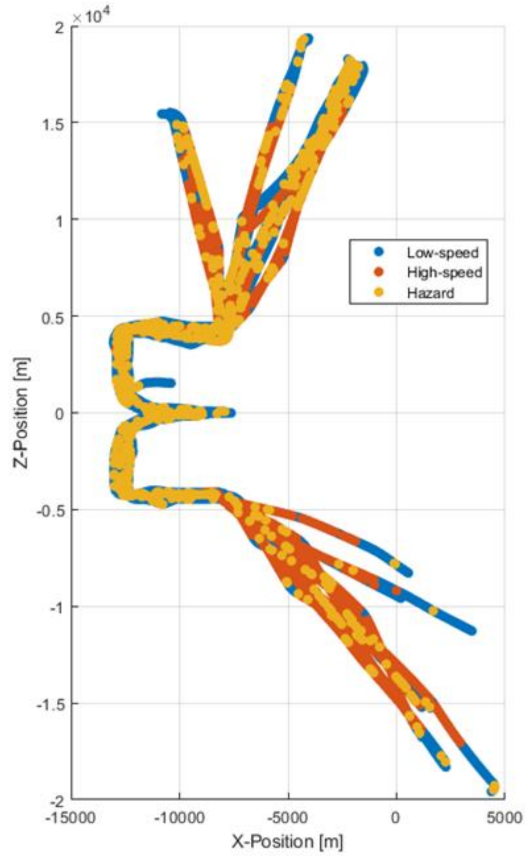


Figure 20: Top-down view of 4 Hz positionless optimization using landing runway relative coordinates

3.3.5 State Validation through Sampling Frequency Variation

We can also assume that true system states will share behavior in multiple sampling frequencies. State definitions that do not have this property are more likely to be artifacts of the recording process or of the specific parameters provided. As such, we performed the same optimization process on a 40 Hz trace to compare results to the 4 Hz optimizations performed previously.

To begin this comparison, we first consider that while the 4 Hz trace has parameter smoothing implemented to reduce the effect of high-frequency parameters, the individual microstates still resemble the original microstates taken from the 40 Hz. As such, a 4 Hz classifier should be able to classify 40 Hz microstates, and vice-versa, but the more visibly dependent microstates in the 40 Hz trace may affect the tuning of a 40 Hz classifier. When optimizing a GMM classifier to the 40 Hz trace as before, we see varying state definitions because of this dependency.

Compass and target heading state definitions appear like each other, but nothing like their 4 Hz counterparts, as shown in Table 8 and Table 9. Completely different extrema are shared, highlighted in red and green as usual. This issue is likely due to the inclusion of low-frequency, position parameters, as the higher 40 Hz frequency is more biased to low-frequency parameters. We can see this bias by examining the top-down plot in Figure 21, where each state appears to be constrained to specific regions of the map. Overall, it is difficult to extract meaning from these state definitions beyond the local state, which contains all points of stall in the trace, but is also so present in the rest of the trace that it is unhelpful to label it as exclusively a hazard state.

Table 8: Compass heading 40 Hz normalized parameter means

Name	**SINCH**	**COSCH**	**XPT**	**YVI**	**ZVT**	**ZVL**	**FVP**	**VVP**	**HVP**	**T**
North cruise	–0.63	0.12	–0.49	0.00	–1.01	–0.69	0.67	0.66	–0.01	0.67
South cruise	0.84	–0.49	0.71	–0.02	0.74	0.84	0.85	0.82	–0.01	0.85
Local	–0.03	0.18	–0.05	0.01	0.26	0.01	–0.85	–0.83	0.01	–0.85

Table 9: Target heading 40 Hz normalized parameter means

Name	**SINTH**	**COSTH**	**XPL**	**YVI**	**ZVL**	**FVP**	**VVP**	**HVP**	**T**
North cruise	–1.01	0.63	0.00	–0.05	–1.14	0.18	0.14	–0.02	0.17
South cruise	0.98	–0.26	0.62	0.05	0.88	0.42	0.40	0.02	0.43
Local	–0.03	–0.47	–0.87	–0.00	0.28	–0.83	–0.76	–0.00	-0.83

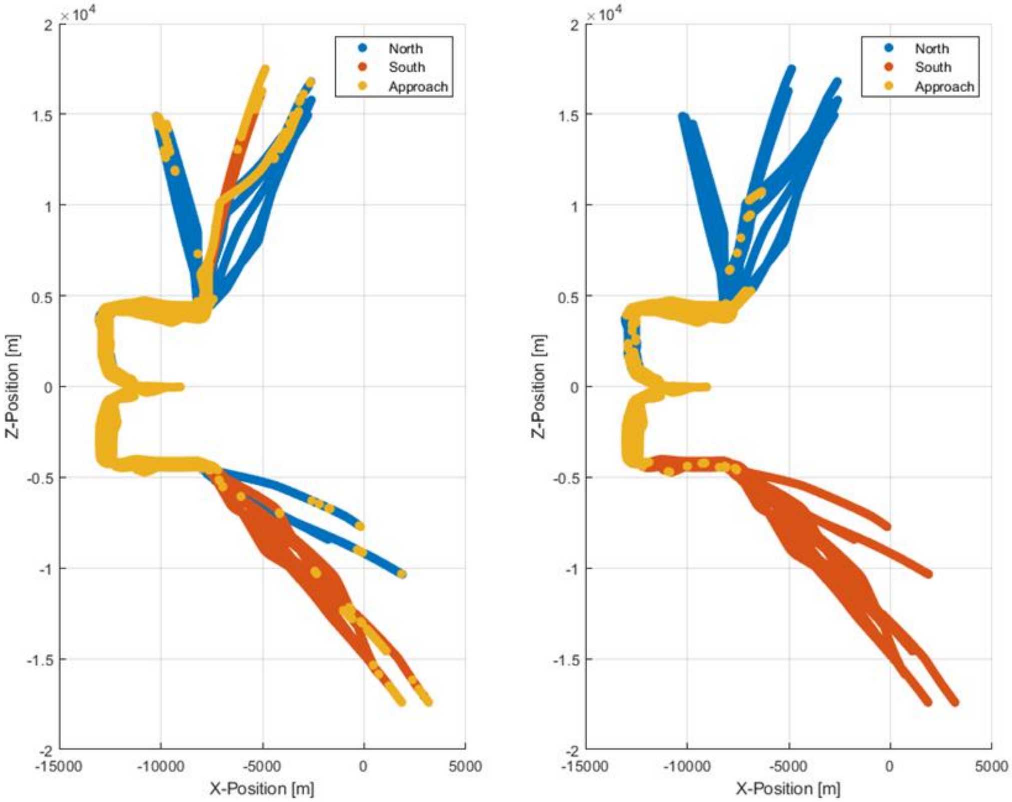


Figure 21: Top-down view of 40 Hz optimizations using landing runway relative coordinates

It follows then that a positionless optimization then would likely have fewer problems with states being fixed in position. Table 10: Positionless 40 Hz normalized parameter meansTable 10 shows the normalized means for such an optimization, which now shares the 4 Hz definitions for states, with one exception, the horizontal plane-relative velocity extrema. In the 40 Hz case, the hazard case is characterized with a high HVP mean, where in 4 Hz, hazard is characterized with a low HVP mean, suggesting that these state definitions may be describing different phenomena. However, consider that the sign of HVP may not be relevant for determining the state of the system. The aircraft system is symmetric, such that any effects recorded for positive HVP would also be possible if the situation was mirrored so the HVP were negative. That would imply that the mean value of HVP in each state should be near zero. This makes the extreme mean HVP values in all hazardous state definitions likely to be more a result of the specific data in the trace, and less meaningful as a description of hazard. If we instead inspect the standard deviation of HVP in both positionless models however, we can see that it is much higher in the hazard definition than those for other states, as we would expect. This shows consistency in definition beyond just the normalized means, suggesting that the state definitions proposed here are describing real system states.

Table 10: Positionless 40 Hz normalized parameter means

Name	**PA**	**YVI**	**FVP**	**VVP**	**HVP**	**T**	**CSE**
Low-speed	0.32	0.02	–0.41	–0.40	–0.01	–0.40	0.37
High-speed	–0.65	–0.00	0.84	0.83	–0.01	0.84	–0.78
Hazard	0.42	–0.12	–0.60	–0.61	0.10	–0.66	0.61

To confirm that the position-fixated states are no longer present, consider the top-down view in Figure 22, where we see a distribution much more akin to the 4 Hz states.

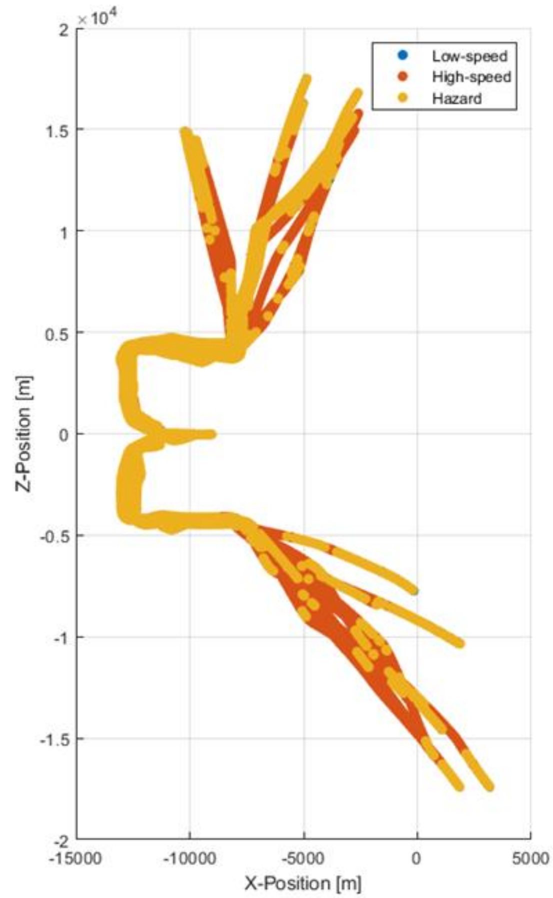


Figure 22: Top-down view of 40 Hz positionless optimization using landing runway relative coordinates

3.4 Difficulties with Applying Basic Machine Learning in Path Determination

With state definitions determined, the trace can now be analyzed for paths. If the states determined are to be considered the ground truth for system behavior, we can generate a ground truth for paths from them by looking at the current and then the next state for each microstate. In the system's operations, these paths are determined by the combination of the microstate and the microinput, so the task becomes determining a method for producing a path from the readings.

Unfortunately, our analysis suggests that basic machine learning techniques are not sufficient to determine path from current microstate and microinput. This could be a result of any of three factors:

1. Paths do not necessarily have distinguishable characteristics in readings
2. Inverse time scaling of the trace makes paths difficult to observe
3. Parameter connections are complex, making the true prediction of the system complex

3.4.1 Path Detection with Basic Machine Learning

In theory, we could train a simple machine learning (ML) classifier to distinguish paths in a similar way to how we determined states. This would be slightly different however, as we would have a ground truth, making any classifier we developed able to use supervised learning techniques. These classifiers tend to be simpler, because they can check their accuracy against the ground truth to determine success rather than optimizing against an abstract classification index like the Calinski-Harabasz index. To do this, we must:

1. Use the known states to generate path IDs for each reading in the trace. For example, a reading starting from S1 and followed by S1 would be path 1, a reading in S1 and followed by S2 would be path 2, and so on.
2. Train a supervised ML classifier to interpret readings into path IDs, using the generated path IDs as ground truth.
3. Decompose the most effective ML classifier to determine characteristics of each path.

This process can be further improved by training a ML classifier for each initial state, instead of a general classifier for distinguishing paths. This reduces the number of paths that need to be distinguished by a single classifier from n^2 to n, where n is the number of states, and would take advantage of the existing state classifier we have already developed. It would also have the additional benefit of narrowing the data which the classifier needs to account for to only that in its initial state, making it theoretically simpler to distinguish.

MATLAB natively supports many ML classifiers but given the many successive classifiers I needed to construct, I opted to only train classifiers from the list that could produce a result quickly in parallel to other training. With this option, I was able to train many different variations of classifiers at once and select the result that produced the greatest accuracy for classifying paths when using a five-fold cross validation, with one-fifth of the training data is reserved for checking accuracy. The classifiers then considered in this approach were variations on decision trees and *K*-Nearest Neighbors (KNN).

Decision trees classify readings by sequentially passing them through Boolean checks, for instance, checking if pitch angle is above a given threshold to determine if the aircraft if going to stall and transition into the hazardous state. They can be trained to have a varying degree of fidelity, measured in the number and specificity of checks, but were most frequently selected in high fidelity variants, suggesting that the paths between states are difficult to distinguish.

KNN classifies readings by instead comparing new readings directly to the training set, placing the readings in a hyperspace as done before with the classifiers used for state definition. Then, it determines the *k* nearest readings to the unknown reading and determines which path ID has the highest count in the *k* selected. If the same number of points are randomly sampled from each state to train the classifier, it is probable that this path ID is also the ID for the new reading, so KNN outputs this path ID as its classification. For instance, Figure 23 shows how a varying *k* value might change the classification of the central point, with each successive circle enclosing a corresponding *k* nearest neighbors. Further variations on this method include altering the distance metric, which can improve KNN performance on higher dimensional data, and enabling distance-based weighting, where classification is biased towards readings that are closer to the unknown point, which can improve performance but requires further training to tune.

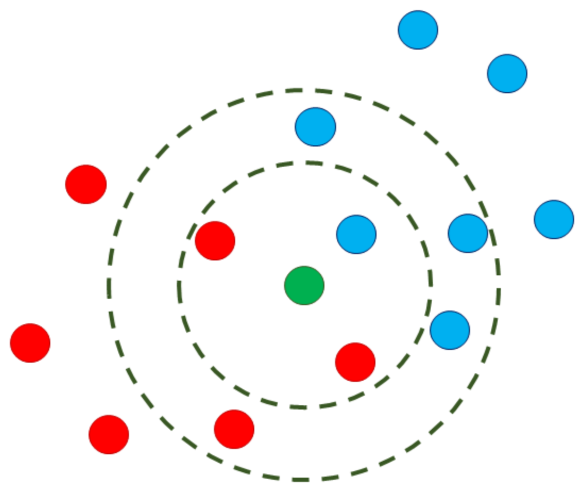


Figure 23: K-Nearest Neighbors visualization

KNN highly values density of points. This in itself is not a problem, as an indicator of a clear distinction between definitions is the density of readings matching each description in the state space—higher densities suggest consistent behavior. This emphasis however results in KNN being heavily biased in training towards paths with the highest number of readings in the training set. Given that we expect that most paths are *stable* paths (paths that return to the current state) a training set that includes all the paths out of our state of interest will be biased towards identifying stable paths if KNN classifiers are selected. To reduce this bias, we randomly selected the same number of readings for each path when training classifiers. We compared both training methods to determine whether any consistent gaps persist despite random sampling.

Unfortunately, this technique of using simple ML classifiers is not sufficient for finding definitions for path behaviors. No classifier was able to successfully parse paths from the coherent state definitions, with consistent issues regardless of the parameters included. In general, paths that return to their original state, which we will call *stable* paths, are the most consistently identified, but other path identifiability varies state to state.

We used confusion matrices, like the example shown in Figure 24, to visualize the effectiveness of each classifier clearly. Once paths are classified, we compare the ground truth path we generated by the state definitions, and the predicted path from the trained classifier. Starting with the central matrix, rows indicate the ground truth path ID, and columns indicate predicted path ID, and the value in the corresponding element indicates the total number paths found in the trace with those IDs. For example, element (1,2) in our example matrix has a value of 83, telling us that 83 paths were labeled as paths to high-speed cruise, but were actually paths to low-speed cruise. Ideally this, main matrix would be a diagonal matrix, as this would indicate that all paths were classified correctly. Elements that are closer to this ideal number are color-coded in darker blues, while elements that are farther away are colored in orange.

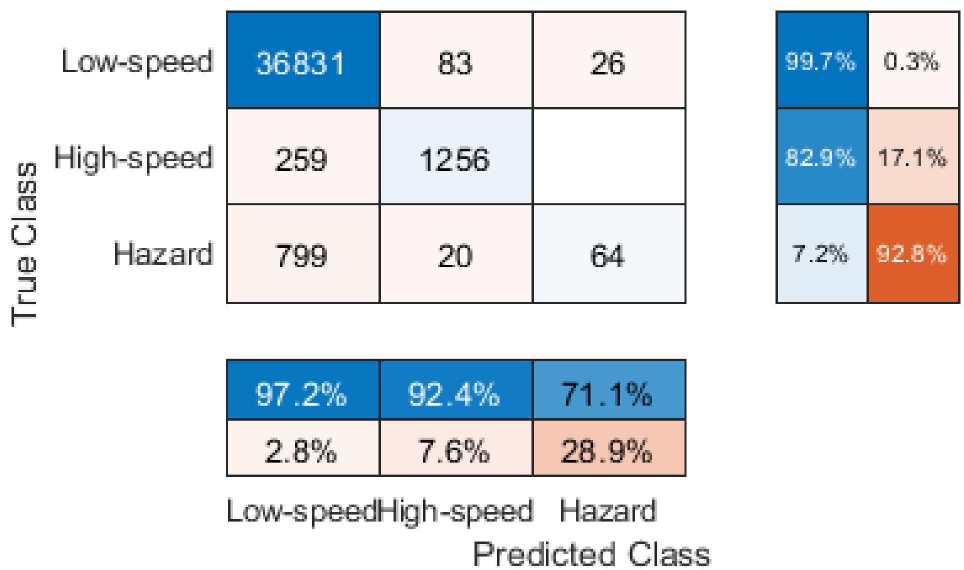


Figure 24: Example confusion matrix

The submatrices correspond to measurements of the true positive rate (TPR) and positive predictive value (PPV) respectively. TPR, measured in the left column of the right matrix, indicates the rates at which each path was correctly identified, mathematically measured as what percentage of the row is in the diagonal. PPV is then measured in the top row of the bottom matrix, and indicates the rates at which prediction is correct, mathematically measured as the percentage of each column that is in the diagonal. Higher percentages in the diagonal indicate a more effective classifier, so higher TPR and PPV also indicate better performance. Each submatrix is color-coded to match the main matrix, with blue indicating higher performance, and orange suggesting lower.

Classification of Paths out of Low-Speed Cruise

In general, both methods of classifying paths out of low-speed cruise showed bias towards predicting stable paths, and paths into the hazard state were generally the most difficult to correctly identify. Even accounting for bias, paths to hazard are disproportionately classified as stable paths, suggesting that the path to hazard and the stable path are very similar, more so than paths to high-speed, which has fewer misclassifications, despite occurring more frequently. As shown in Figure 25 and Figure 26, the standard sampling method of providing all paths as training data appears to have resulted in classifiers with a higher PPV, while random sampling resulted in a higher TPR.

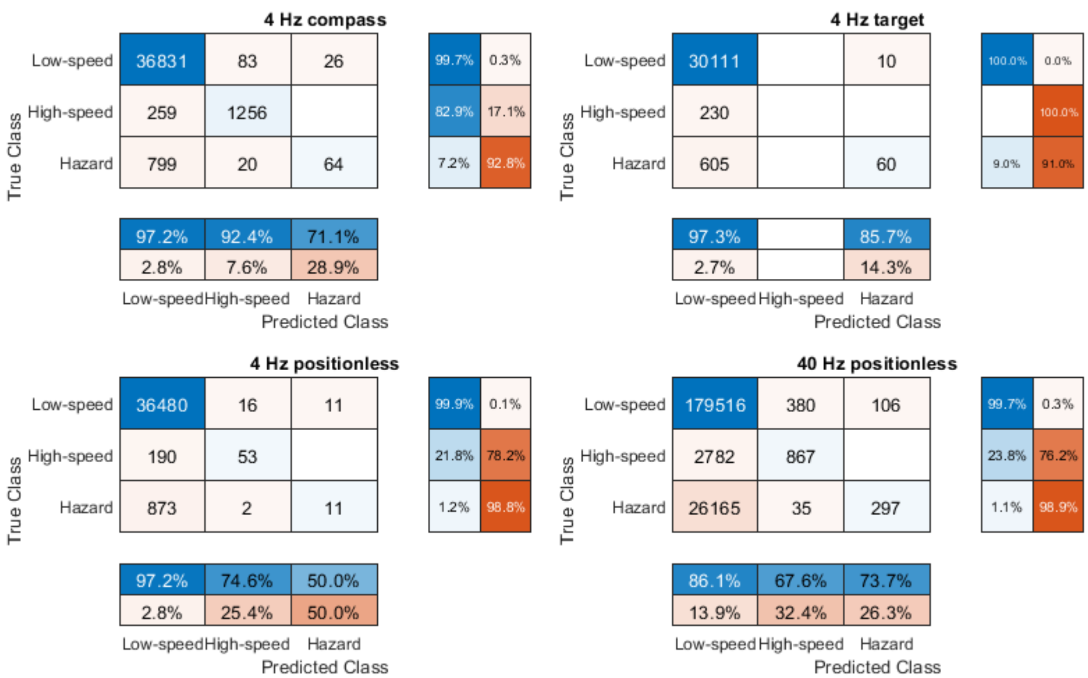


Figure 25: Low-speed cruise, direct prediction with standard sampling confusion matrices

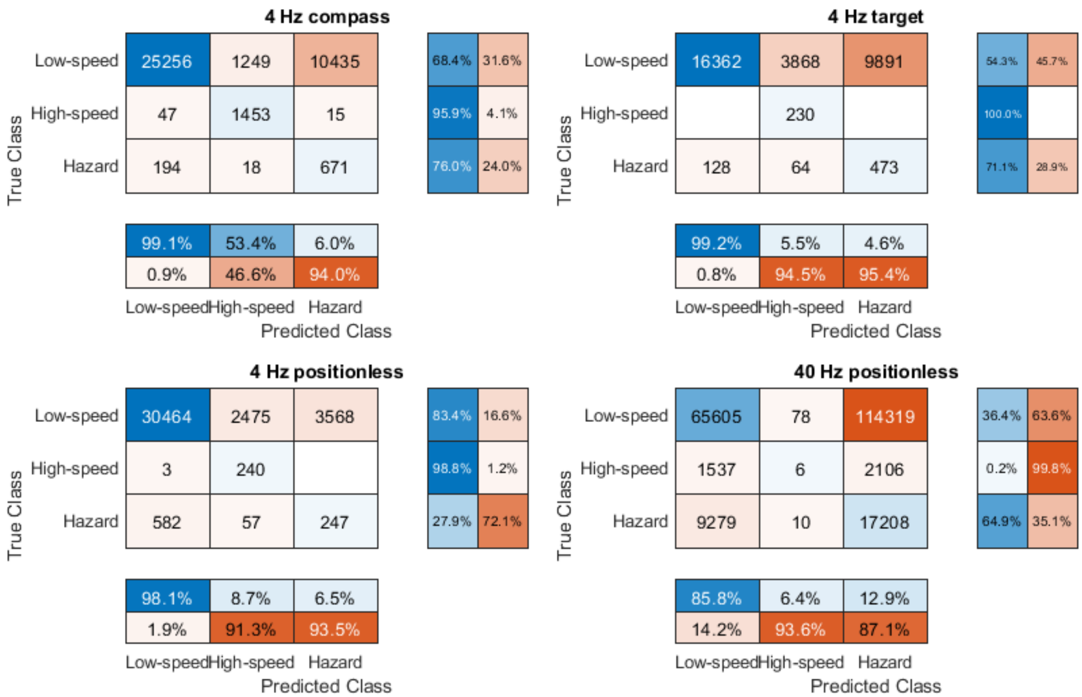


Figure 26: Low-speed cruise, direct prediction with random sampling confusion matrices

40 Hz random data shows the largest amount of misclassifications of the stable path as the hazard path, suggesting that this high frequency shows the most similarity between the two paths.

Classification of Paths out of High-Speed Cruise

Similar trends to those seen in the paths out of low-speed cruise are visible in high-speed cruise. Standard sampling shows biasing towards the stable path, has a higher PPV, and lower TPR as shown in Figure 27 and Figure 28. Note that paths from high-speed to hazard and low-speed are not being confused in the same way that paths to hazard and low-speed were when they originate in low-speed. This suggests that these paths look more different than they did in low-speed, but the relative infrequency of paths to hazard from high-speed mask behavior.

Additionally, note that the total number of paths to hazard is smaller in the 40 Hz model than in all others. If this were a true path, this value would likely be conserved, or at least remain a similar magnitude. Instead, it seems plausible that the increased sampling rate captured low-speed readings between high-speed to hazard readings in 4 Hz, suggesting that to navigate from high-speed to hazard, the aircraft quickly passes through low-speed in our trace data.

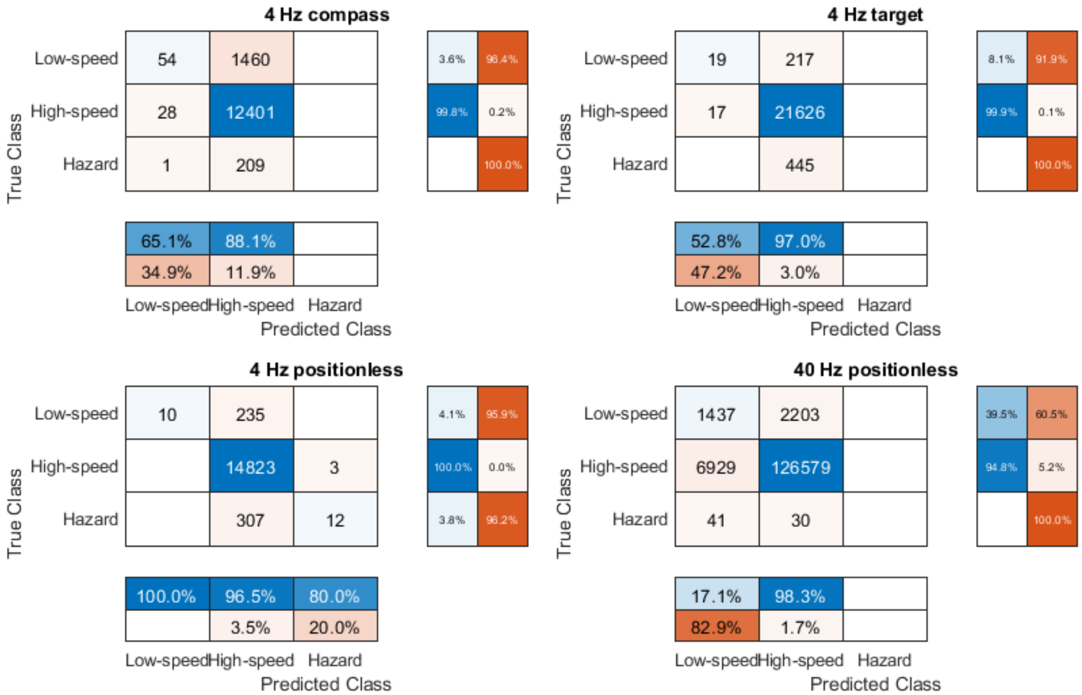


Figure 27: High-speed cruise, direct prediction with standard sampling confusion matrices

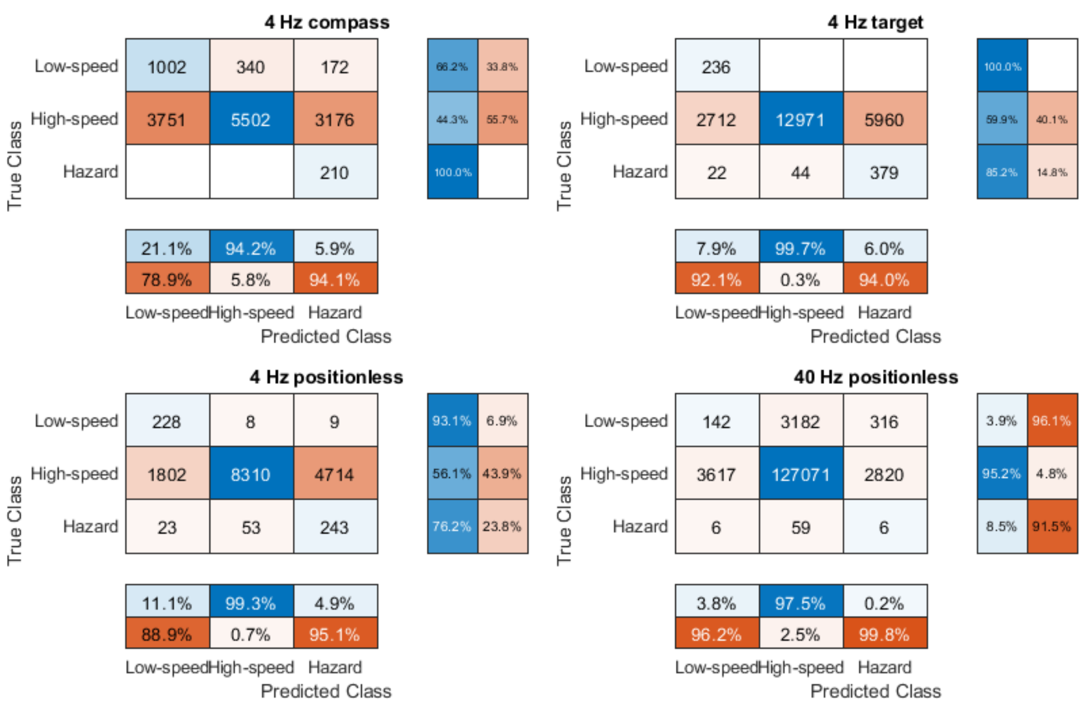


Figure 28: High-speed cruise, direct prediction with random sampling confusion matrices

Classification of Paths out of Hazard

Of the paths from the three initial states, paths out of hazard are the most consistently identifiable. In all three 4 Hz classifiers, regardless of training method, the lowest TPR and PPV is 73.6%. Figure 29 and Figure 30 once again show that paths to hazard and low-speed are often misclassified as the other, but in lower rates than in other states. Paths to low-speed are more frequently misclassified as paths to high-speed than paths to hazard are to high-speed, suggesting that paths to low-speed are more similar to high-speed than the stable path is. As in the previous case, 40 Hz sees dramatically fewer paths to high-speed than the other models, suggesting a similar path to low-speed is necessary first in all but the most specific cases. Otherwise, the 40 Hz data performs much worse than the other models however, with many more misclassifications.

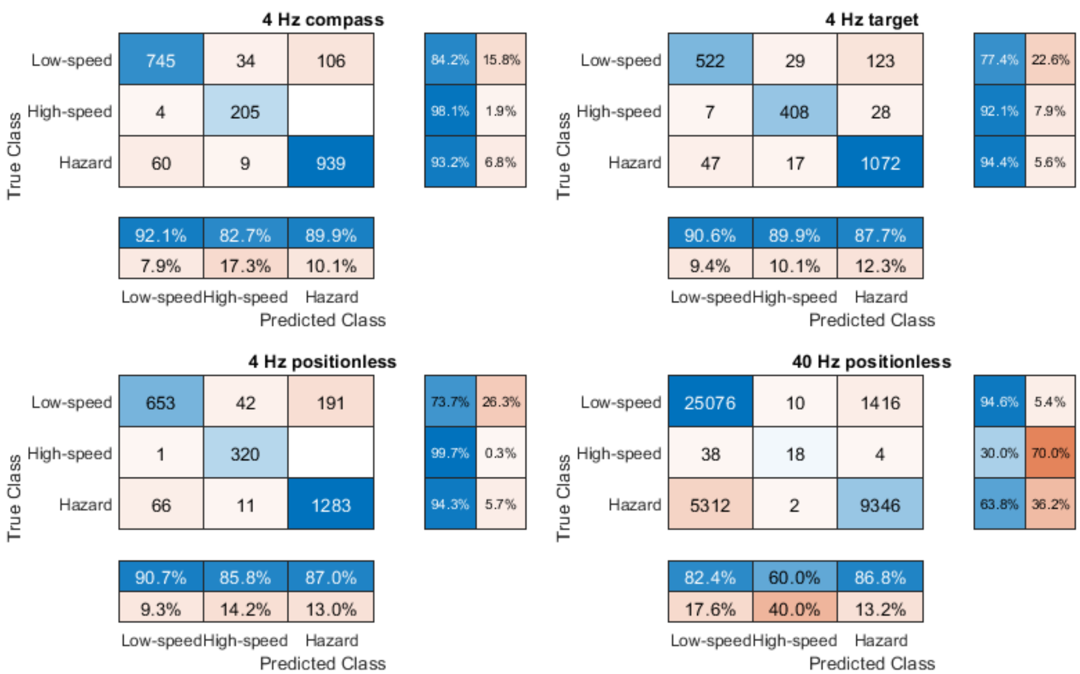


Figure 29: Hazard, direct prediction with standard sampling confusion matrices

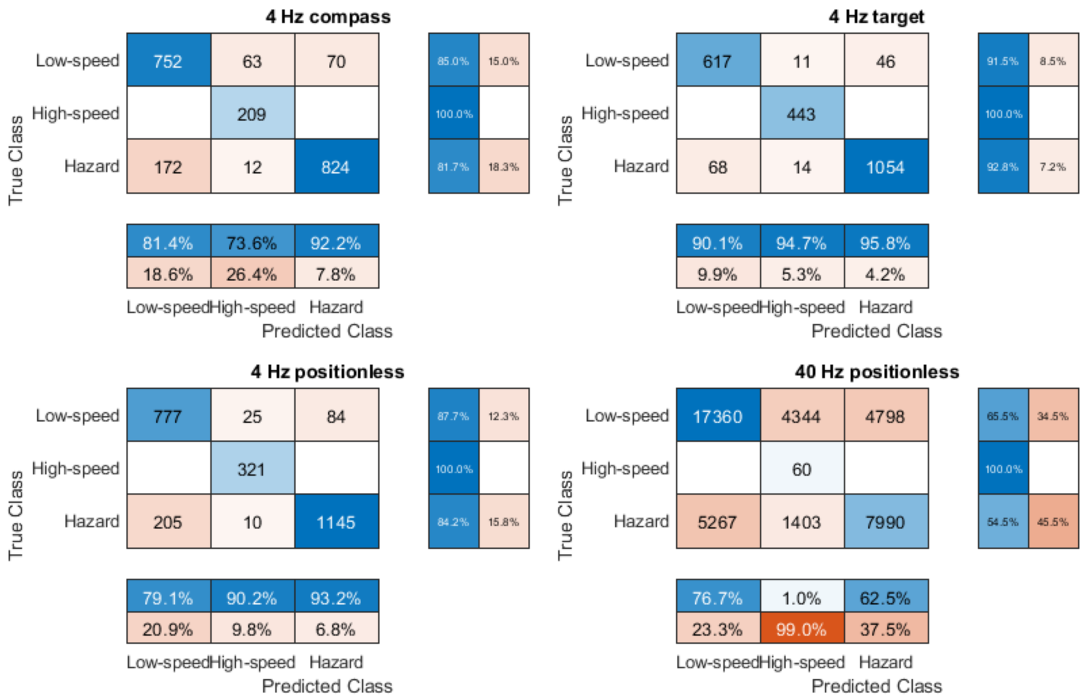


Figure 30: Hazard, direct prediction with random sampling confusion matrices

3.4.2 Path Detection with a Multi-Classifier Model

In theory, all the classifiers created in this exercise have been attempting to capture the same behavior, just defined slightly differently. This includes the descriptions of state we have generated. With this concept in mind, it seems plausible that a joint definition can be reached, where multiple classifiers can be applied at once and results compared.

To do this, we first constructed a combined state model using all three consistent 4 Hz models. Each model classified the trace as before, and classifications were weighted by their probability of being generated in their respective GMM function. This resulted in an equally biased classifier, generating as close to all three models as possible. Then, we passed each trace reading through the standard, direct path classifiers based on their combined state. We weighted these paths predictions by the confusion matrix PPV values, including the other terms in the matrix column as other weighted towards other path IDs.

This approach did not result in any improvement over other methods. Figures Figure 31–Figure 33 shows how the bias towards the stable path was consistent as in other cases, and how paths out of hazard remained the most consistently simple to identify, with the caveat that this method proved the least effective at locating these paths.

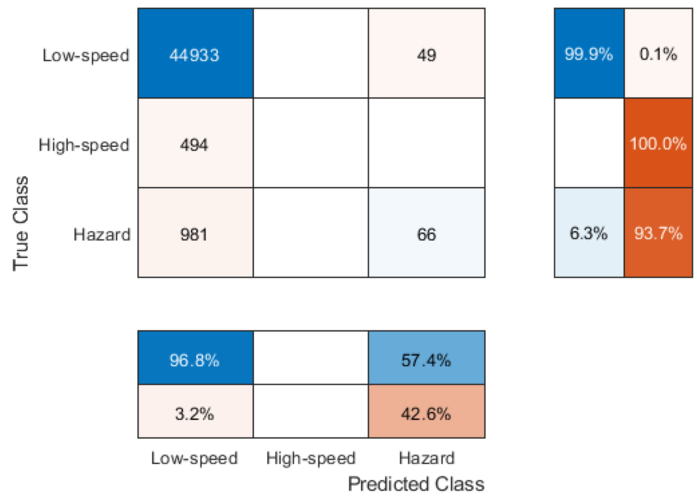


Figure 31: Merged path model for paths out of low-speed cruise

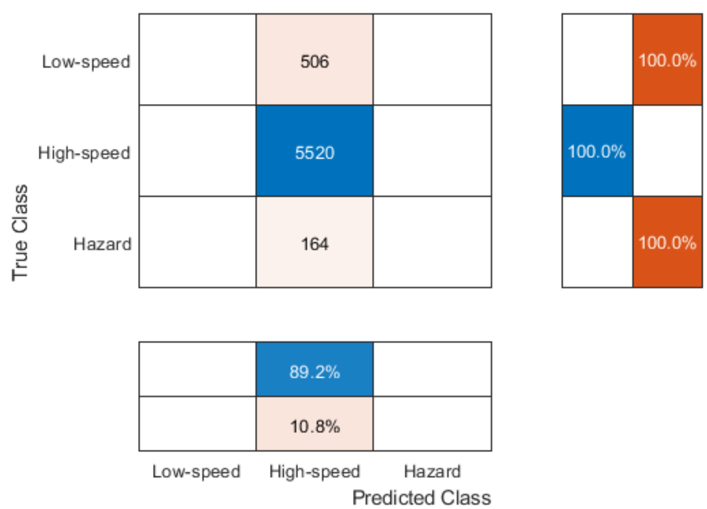


Figure 32: Merged path mdoel for paths outs of high-speed cruise

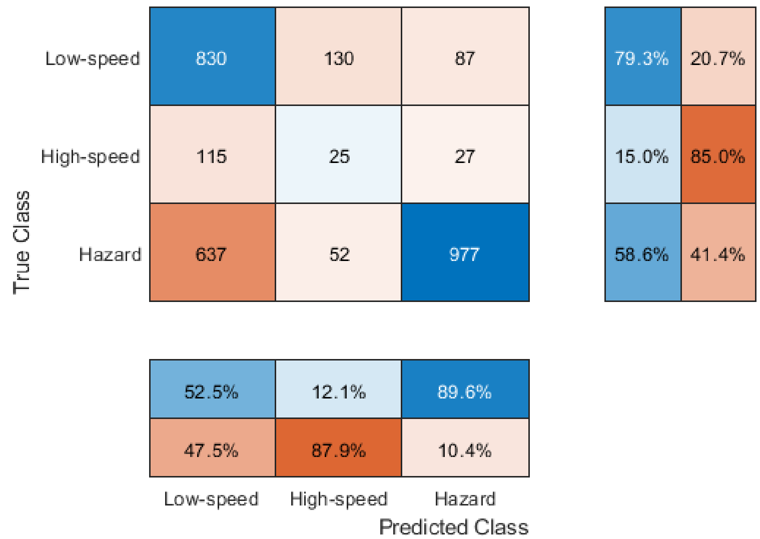


Figure 33: Merged path model for paths out of hazard

3.4.3 Path Reading Comparisons

Generally, we expect that ML classifiers will fail when each path has similar values in each parameter and control. To visualize where classifiers may see this issue, we normalize parameter and control means for path and compare them, similarly to how we compared state definitions. Paths that show similar metric means and standard deviations will be more difficult to distinguish, with more metrics sharing behavior being more difficult to distinguish.

Figure 34 shows each of these comparisons of paths out of low-speed cruise, examining each of the four consistent models found. As expected from the classifier performances in the confusion matrices, paths to high-speed cruise are consistently showing different metrics, particularly FVP, VVP, T, elevators, and throttle. Each of these metrics have a clear mean outside the standard deviations of the other paths and have a generally smaller standard deviation. The stable path and hazard path then show little differentiation at all, explaining why path to hazard is difficult to identify.

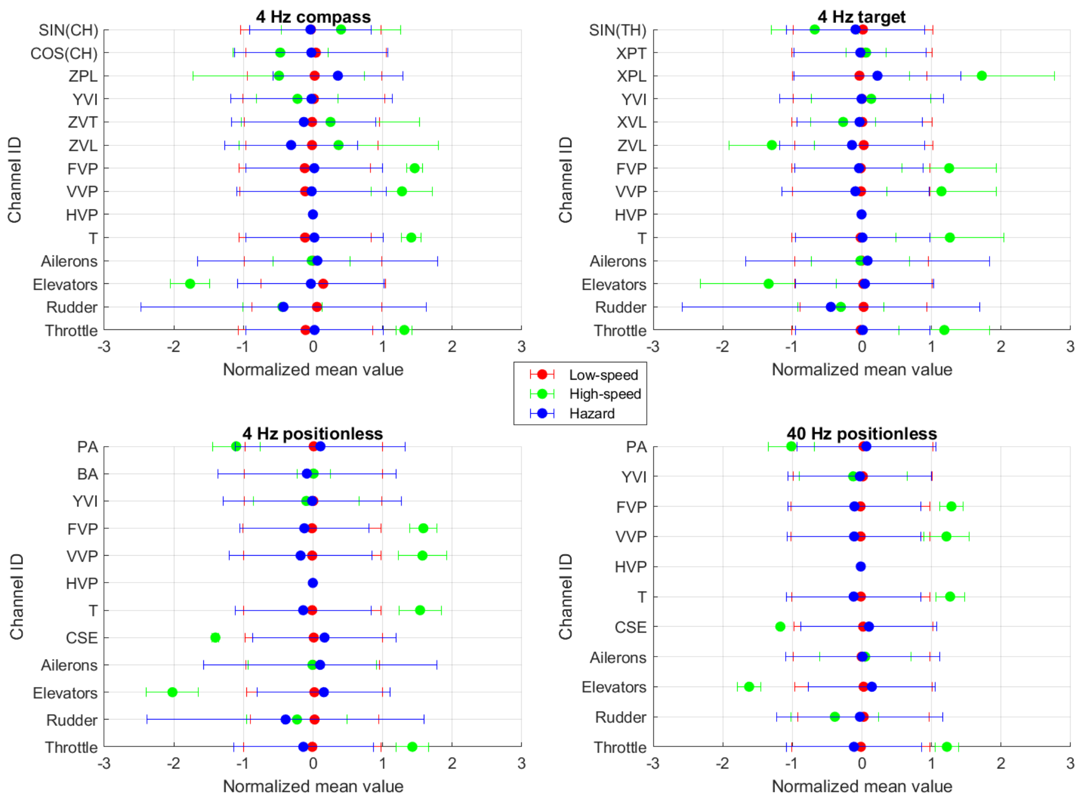


Figure 34: Normalized metrics in paths out of low-speed cruise

Paths out of high-speed cruise, shown in Figure 35, are much more difficult to distinguish based on mean and standard deviation. Mean values are generally much more tightly packed, and almost exclusively within one standard deviation of one another. Note that the wide standard deviation of 4 Hz positionless hazard decreases in 40 Hz, while low-speed increases. This suggests that paths seen as high-speed to hazard in 4 Hz are indeed high-speed to low-speed to hazard paths at higher sampling rates, confirming behavior in the confusion matrices.

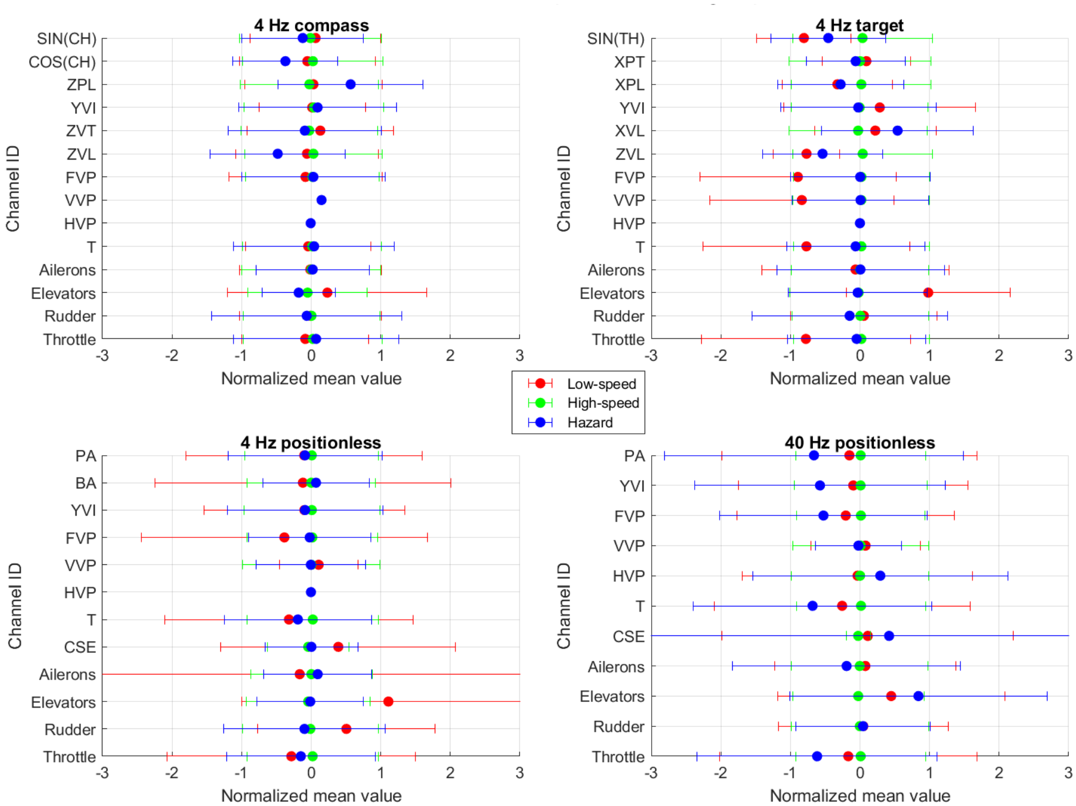


Figure 35: Normalized metrics in paths out of high-speed cruise

Interestingly, paths out of hazard much more distinct than those from low-speed. In Figure 36, we can see the same trend of high-speed paths being isolated and easily identifiable, while low-speed and stable paths are more tightly packed. However, the mean values of these two paths are slightly more distinct than those seen in low-speed, which is apparently enough to consistently distinguish paths as shown in the confusion matrices.

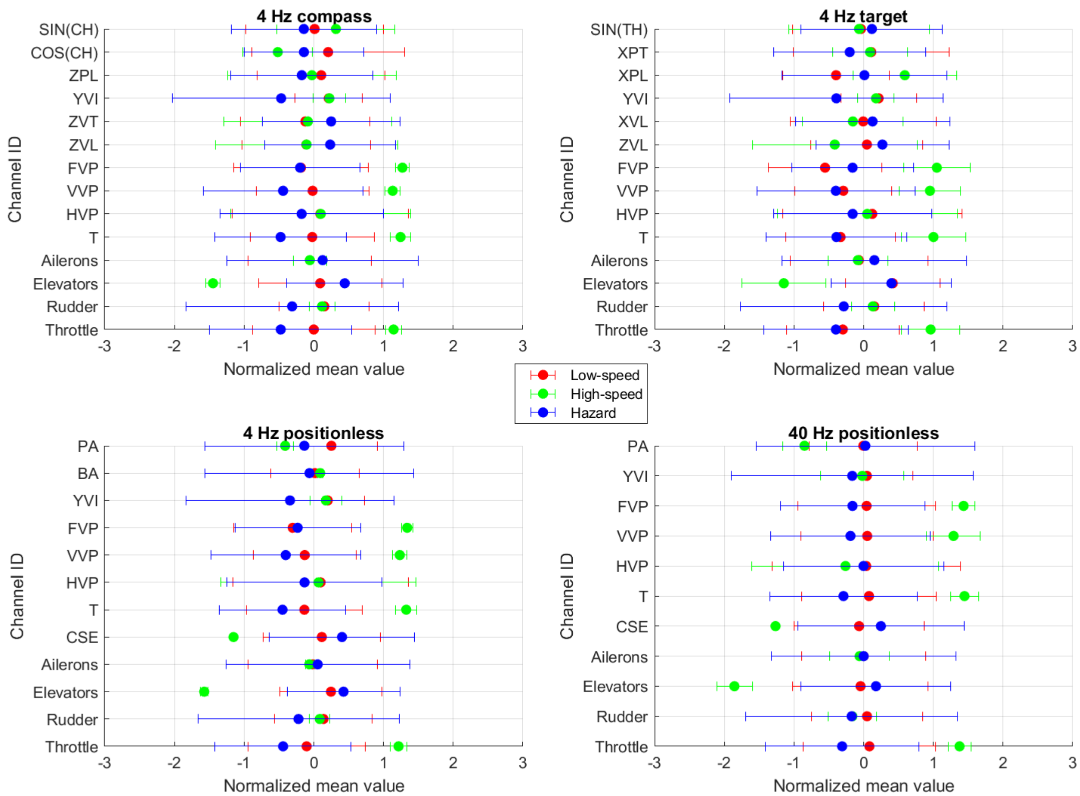


Figure 36: Normalized metrics in paths out of hazard

Overall, we can consider many of the issues visible in the confusion matrices as being direct results of system metrics behaving similarly. Under the right conditions, these paths can be distinguished, even when mean metric values are closer, but without consistency in difference, a classifier examining a single reading and direct predicting behavior will not be able to differentiate paths.

3.4.4 Path Detection with Microstate Prediction and Complex Interactions

An alternative to predicting path directly from readings is to predict the next microstate, classify the result into a state, and use the predicted state and the initial state to label the path. In this way, we could predict path without having to rely on direct classifiers, avoiding the issues with metrics appearing similar.

To predict microstate, we elected to construct linear regression models for each parameter. To ensure that the regressions were trained to predict any behavior unique to the state, we produced unique regression sets for each initial state. Thus, path labeling from readings follows the following procedure:

1. Normalize the readings parameters and controls against the entire trace.
2. Classify the normalized readings into a state.
3. Normalize the original readings parameters and controls against the readings with the same state in the trace.
4. Predict the value of each parameter in the next microstate using the linear regression models specific to the current state.
5. Normalize the resultant microstate parameters against the entire trace.
6. Classify the normalized predicted microstate into a state.
7. Classify the original readings with the corresponding path ID for the current and predicted subsequent state.

Once again however, this method was no more accurate at producing correct path identifications than the last. Although, the confusion matrices are different in behavior, showing faults for different paths. In Figures Figure 37–Figure 39, we can see that performance has declined relative to the previous method. Paths from hazard still appear the simplest to identify, but both TPR and PPV are affected negatively in all cases.

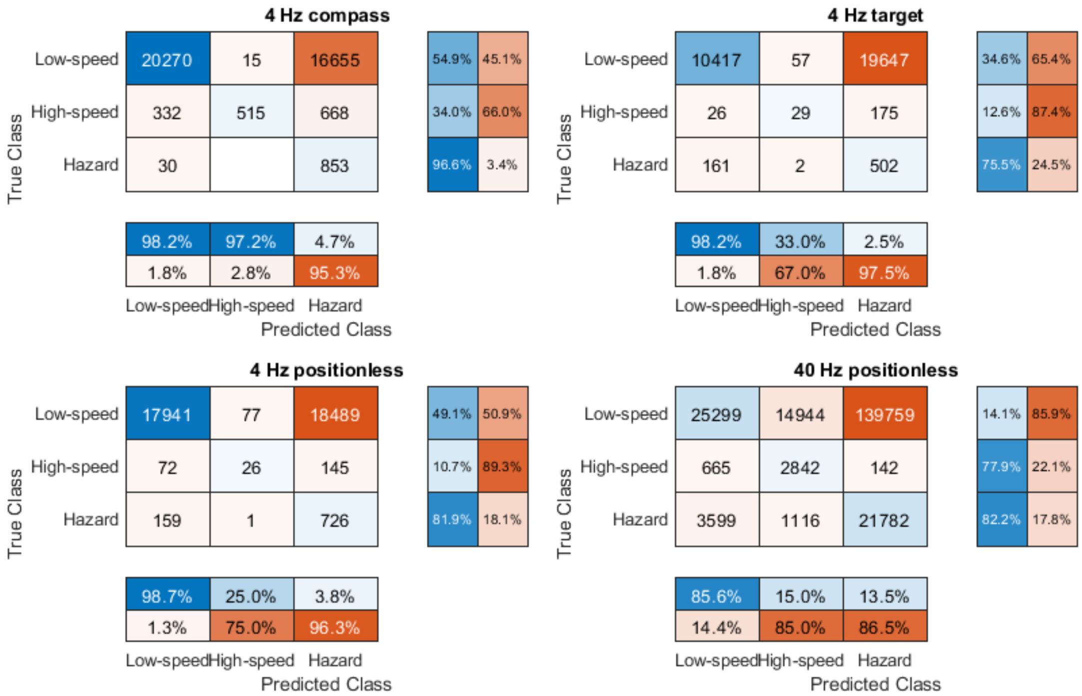


Figure 37: Low-speed cruise, microstate prediction confusion matrices

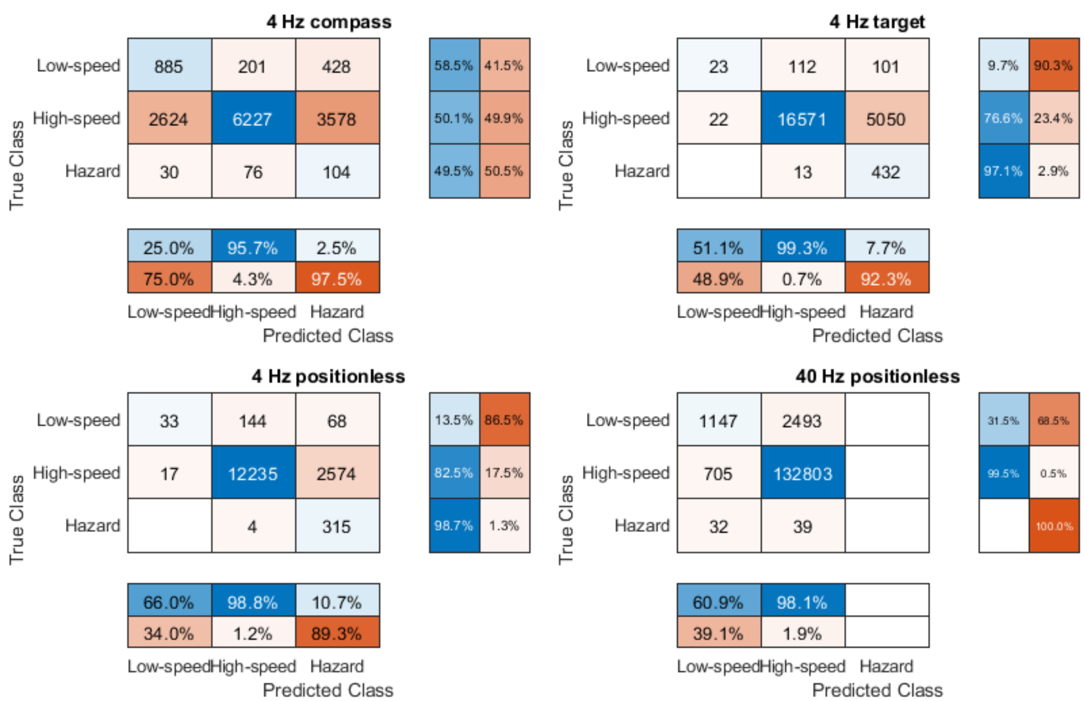


Figure 38: High-speed cruise, microstate prediction confusion matrices

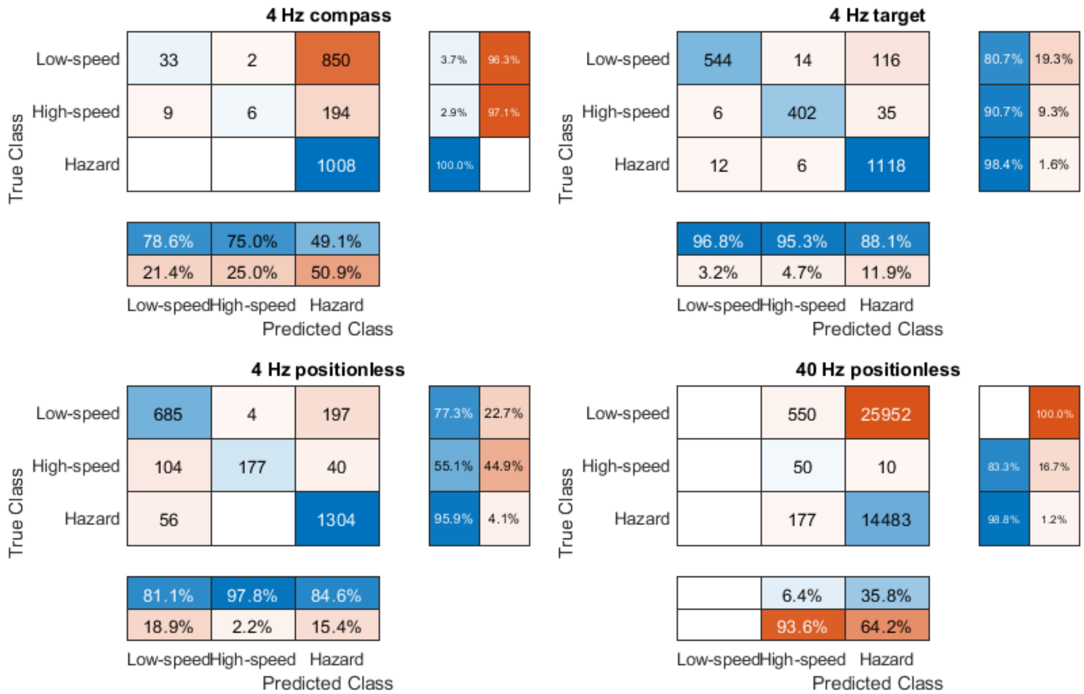


Figure 39: Hazard, microstate prediction confusion matrices

These issues are likely due to difficulties with extracting effective parameter predictors, as the metrics in the data set showed little correlation to one another. RMSE in these models could often exceed one, making many metrics poor predictors of parameters. Such behavior is likely due to unknown, or complex linkages between these metrics. Linear regressions alone appear not to be capable of predicting parameters with enough precision unless extensive tuning is done.

To then improve on this method, we could develop more sophisticated prediction tools, relying on neural networks or other ML methods to automate the process. Alternatively, a designer could manually include known physics models for different parameters, for example, calculating next position from position and velocity, but this requires knowledge of the metrics which may not always be available. It is also plausible that parameter prediction cannot be done from the microstate used to identify state alone, and other information from the complete set in the trace is necessary. However, the larger number of dimensions included in a model expand the *n*-dimensional space, making it more difficult to ensure that enough data is collected to verify behaviors (Köppen, 2000). Careful inclusion and exclusion testing would need to be performed to avoid implying false connections between metrics.

3.4.5 Inverse Time Scaling

As noted previously, it is apparent that some paths occur to quickly to be correctly labeled in the 4 Hz model and are likely more visible in 40 Hz. This suggests that detecting these paths requires high sampling rates, despite their worse performance in path identification.

High sampling rates alone increase the proportion of paths taken up by stable paths, as unstable paths then take up a smaller proportion of the total time. This first biases classifiers towards the largest represented sets in the training data, the stable path, which can be difficult to correct for as seen in the random sampled tests. It also forces state transitions to occur over a shorter period of time as path lengths decrease. This makes readings more similar by giving less time to change, which makes them more difficult to distinguish. It also expands state definitions by shortening time spent on an unstable path "between states". This makes state boundaries less distinct, and less differentiable for most applications.

Multiple potential solutions exist. First, 40 Hz behavior could potentially be integrated into 4 Hz traces by adjusting metric smoothing windows, preventing metric values from being over smoothed by wide windows. Alternatively, the window could be applied non-uniformly to emphasize current value or deemphasize future values to draw a harsher line between one reading and the next. Another option would be to include the standard deviation of measurements in the smoothing window as new metrics. This could take into account how variable the true reading is at any given point in the trace but would double the number of metrics in the trace, causing performance and dimensionality issues as previously discussed.

Another method for distinguishing behavior would be to compare where different classifiers succeed at making predictions, and selectively applying classifiers only when they have a higher degree of accuracy. This transition between classifiers could also consider the standard deviations of metrics in the smoothing window as a method of measuring imminent variability. Overall, this would be difficult to implement without bias as seen in the multi-classifier model attempted here but could be effective.

An extreme option would be to modify the trace to allow for multi-frequency sampling, but this seems to be the most complex to implement correctly.

4. DISCUSSION AND CONCLUSION

Overall, this document demonstrates that a basic approach for constructing state machines from a trace can be extrapolated across multiple systems and capture their respective behaviors with some success. For systems with simple system factors, entirely autonomous methods can construct basic state machines from a trace, but some work is still needed to expand on the method.

The basic methodology functions adequately for constructing complex factor state machines. With it, we can find state definitions which appear consistently in the trace, but we cannot determine how users can alter state enough to construct useful path definitions.

4.1 Conclusions on the Use of System Factors

System factors remain a useful tool for separating and understanding system behaviors, if only qualitatively. Each factor of continuity, parallelism, temporality, and boundedness has strong implications on what needs to be done to decompose the system trace into a state machine. In general, we can expect that systems with more continuity, parallelism, and temporality will be more difficult to study, and those with less boundedness to be less so.

Continuity implies a lack of distinction between system conditions, and therefore system states. Parallelism increases the total number of ways the system can be interacted with, and obscures operations. Temporality behaves similarly to continuity, by blending obscuring state definitions in continuous time. Boundedness decreases the number of unknown characteristics that need to be defined and checked in the process.

These definitions provide a basis for beginning to model a system and force the analyst to consider system behavior before recording a trace.

4.2 Conclusions on the Use of Logical Tools for Simple Systems

Logical tools operate well on simple factor systems but can result in state machines that make no implications beyond replicating exact behavior in the trace. The exception to this is the state machine constructed in reverse order, which, because of its looser path combination rules, can introduce recursion into the model. However, it can also result in paths that are not traversable in all cases, making it less effective as a tool for analysis.

4.3 Conclusions on the Use of Machine Learning for Complex Systems

Complex factors result in many additional complexities in the analysis procedure. Using statistical tools, we were able to isolate state definitions that were consistent in several different parameter schemes and sampling rates and matched up with perceived real-world behavior, as well as known anomalous behaviors. Overall, these states are helpful for demonstrating that we can extract trends in trace data, but the states themselves are not particularly sophisticated. Given the curse of dimensionality, it seems plausible that the only method for developing more informative states with this technique is to reduce the total number of parameters used in the state generation process. In doing so, an analyst could iterate through many parameters sets and tune many separate classifiers, comparing results between them to find classifications consistent despite extremely different parameters provided.

Path description is even less developed, as some paths were unable to be defined by the classification methods attempted. Direct path classification attempts failed to identify meaningful distinguishing behaviors in parameters and controls due to the similarity of these values between paths. More sophisticated supervised ML tools may be able to distinguish paths, but given the metric similarities, it seems unlikely that such tools will be more effective. Alternatively, microstate prediction could be improved using more advanced prediction techniques given the wealth of ML tools available. This approach seems the most plausible area for improvement.

4.4 Closing Thoughts

Overall, the methods examined in this document for constructing a state machine from a trace proved to be sufficient for generating simple state definitions in multiple systems with dramatically different qualities, but more work is needed to expand on these methods.

APPENDIX A. CHAPTER 2 SCRIPTS

Setup

To construct system state machines for the simple test case, first compile each of the scripts in this appendix into separate files, named exactly as their heading.

CoffeeMiner.mat

Next, run `CoffeeMiner.mat` with all of the other scripts in this appendix in the same folder. This will extract trends in the matrix stored in `CoffeeMachine_03.dat`. The user should define string and integer labels for this file in `CoffeeActLabels_s.dat` and `CoffeeActLabels_i.dat` respectively.

To interpret forward iteration results, open the `paths_pre` variable, which stores a path in the state machine in each index. Each path is stored as a sequence of integers, led by the initial state ID and concluded by the final state ID. For example, [1;4;30] corresponds to a path starting in state 1, accepting input 4, and ending in state 30. Intermediate values indicate user inputs provided during the state transition, with each integer corresponding to the input of the same index in the labeling file. Similar formatting for reverse iteration results can be found in `paths_pos`. Note that inputs and states use the same labeling set, so the first new states beyond “Start”, “Data Fault”, and “End”, will index to values greater than the number of inputs plus three.

This script also includes some additional functionalities not used in the final research, tracking the frequency of use of different paths, only tracking instances of use that include specific inputs, and so on. To track frequency of use, set `toggle_ana` to one. Variables `concen_state_pre` and `concen_path_pre` provide the percentage of use instances that included said state or path in forward iteration respectively. Variables `probs_state_pre` and `probs_path_pre` contain the respective probability of navigating to a given state or path respectively given an initial state. Rows in the cell array correspond to initial state and contain a two-column matrix. The first column of this matrix includes the corresponding end state or path ID, and the second column then includes the probability of navigation given the initial state. These variables also have corresponding variables storing the reverse iteration information but utilize the `_pos` suffix instead of the `_pre`.

To force the entire model to include a given input, set `targ_node_AID` to have a value corresponding to the mandatory input ID.

```
%% Coffee Model Miner Version 19
%  Constructs path-based model of coffee machine operation based on trace
%  data collected from video.
%  Requires:
%       - CoffeeMachine_03.dat
%       - CoffeeActLabels_s.dat
%       - CoffeeActLabels_i.dat
%       - globalPathPre
%       - globalPathPos
%       - localPath (02)
%       - terminalSeek (02)
%       - stateIterator
%       - stateEnforce
%       - stateJoin (09)
%       - stateSubset
%       - pathAnalyze (02)
%  Changes from 18:
%       - Utilizes split stateJoin/stateSubset
%       - Removes validate

clc
clear

%% Input target data
toggle_plot_pre = 1;        % Plot previous path
toggle_plot_pos = 1;        % Plot following path
toggle_txt_labels = 1;      % Use text labels in plots
toggle_mand = 0;            % Utilize mandatory action plotting
toggle_ana = 1;             % Collect concentration, timing, and probability
data
targ_node_AID = 0;          % Designate mandatory action in path


%% Instance Data
threshold = 0;                                          % Threshold for data
exclusion
data_master = load('CoffeeMachine_03.dat');             % Data set
start_AID = 1;                                          % Action ID for Start
action
dataF_AID = 2;                                          % Action ID for Data
Failure action
end_AID = 3;                                            % Action ID for End
action
act_num_master = max(data_master(:, 2));                % Number of individual
actions
if toggle_txt_labels                                    % Action Labels
    labelFID = fopen('CoffeeActLabels_s.dat');
```

```
    stateLabel = 'State';
else
    labelFID = fopen('CoffeeActLabels_i.dat');
    stateLabel = 'S';
end
labels = textscan(labelFID, '%s', 'Delimiter', '\n');
labels_master = labels{1};
fclose(labelFID);
fprintf('Begin analysis:\n');


%% Iterate from sample start
fprintf('    Forward iteration commencing...\n');
% Iterate
data_pre = stateIterator(start_AID, targ_node_AID, dataF_AID, end_AID,
data_master, act_num_master, threshold, 1);


% Join and subset states
[data_pre, paths_pre] = stateEnforce(start_AID, dataF_AID, end_AID, data_pre,
act_num_master, 1);
act_num_pre = max(data_pre(:, 2));

% Calculate state paths
state_paths_store = [];
for path1_ID = 1:length(paths_pre)
    path1 = paths_pre{path1_ID};
    state_strt = path1(1);
    state_end = path1(end);

    % Append state path to set
    if isempty(state_paths_store)
        state_paths_store = [state_strt, state_end];
        path_map_store = [path1_ID, 1];
    else
        path2_ID = 1;
        while path2_ID <= size(state_paths_store, 1)
            path2 = state_paths_store(path2_ID, :);
            if isequal([state_strt, state_end], path2)
                break
            end
            path2_ID = path2_ID + 1;
        end
        if path2_ID > size(state_paths_store, 1)
            state_paths_store = [state_paths_store; state_strt, state_end];
        end
        path_map_store = [path_map_store; path1_ID, path2_ID];
    end
end
state_paths_pre = sortrows(state_paths_store);
path_map_pre = path_map_store;

for path1_ID = 1:size(state_paths_pre, 1)
    path1 = state_paths_pre(path1_ID, :);
    path2_ID = 1;
```

```
    while path2_ID <= size(state_paths_store, 1)
        path2 = state_paths_store(path2_ID, :);
        if isequal(path1, path2)
            break
        end
        path2_ID = path2_ID + 1;
    end

    for path3_ID = 1:size(path_map_store)
        if path_map_store(path3_ID, 2) == path2_ID
            path_map_pre(path3_ID, 2) = path1_ID;
        end
    end
end

fprintf('    Forward iteration complete\n');

% Update labels
act_num = act_num_pre;
for state1 = 1:(act_num - act_num_master)
    NLabel = stateLabel;
    NLabel = strcat(NLabel, num2str(state1));
    labels_master = [labels_master{:}, {NLabel}]';
end


%% Visualize Pre Iteration
if toggle_plot_pre
    %% Reset data
    % Instance storage
    strt_node_AID = start_AID;
    end_node_AID = end_AID;
    strt_seq_IDs = [];                                  % Sequence ID for start
nodes
    end_seq_IDs = [];                                   % Sequence ID for end
nodes
    act_cnt = zeros(act_num, 1);

    % Remove actions
    data_store = data_pre;
    seq1_ID = 1;
    while seq1_ID <= size(data_pre, 1)
        if ismember(data_pre(seq1_ID, 2), [dataF_AID, (end_AID +
1):act_num_master])
            data_pre(seq1_ID, :) = [];
        else
            seq1_ID =seq1_ID + 1;
        end
    end

    % Scan for sample terminals
    [strt_seq_IDs, end_seq_IDs, act_cnt] = terminalSeek(data_pre,
strt_node_AID, targ_node_AID, end_node_AID, end_AID, act_num);

    % Determine actions with low measurable behavior
```

```
    skip_AIDs = [];
    for act1 = (act_num_master + 1):act_num
        if (act_cnt(act1) <= threshold) && not(ismember(act1, skip_AIDs)) &&
not(act1 == dataF_AID)
            skip_AIDs = [skip_AIDs, act1];
        end
    end
    skip_AIDs = sort(skip_AIDs);

    % Add states to labels
    labels = {};
    for act1 = 1:act_num_pre
        if not(ismember(act1, [skip_AIDs, dataF_AID, (end_AID +
1):act_num_master]))
            labels = [labels(:)', labels_master(act1)];
        end
    end
    labels = labels';

    figure(1)
    if toggle_mand
        title1 = 'Mandatory Previous States';
        % Set edges
        act_path_pre = globalPathPre(data_pre, act_num, strt_seq_IDs,
end_seq_IDs, dataF_AID, skip_AIDs);
        act_mand_pre = act_path_pre(:, 2);

        % Mandatory previous action model
        disp_path1 = []; % Construct edges
        for act1 = 1:act_num_pre
            for act2 = act_mand_pre{act1}
                disp_path1 = [disp_path1; act1, act2];
            end
        end
    else
        title1 = 'State Diagram as Determined by Forward Iteration';
        % Set edges
        disp_path1 = [];
        for seq1_ID = 1:(size(data_pre, 1) - 1)
            state1 = data_pre(seq1_ID, 2);
            if not(state1 == end_AID)
                state2 = data_pre((seq1_ID + 1), 2);
                if isempty(disp_path1)
                     disp_path1 = [state1, state2];
                else
                    match = 0;
                    edge_ID = 1;
                    while (edge_ID <= size(disp_path1, 1)) && not(match)
                        if isequal(disp_path1(edge_ID, :), [state1, state2])
                            match = 1;
                        end
                        edge_ID = edge_ID + 1;
                    end
                    if not(match)
                        disp_path1 = [disp_path1; state1, state2];
                    end
```

```
                end
            end
        end
        disp_path1 = sortrows(disp_path1);
    end

    diagram1 = digraph(disp_path1(:, 1), disp_path1(:, 2));
    diagram1 = rmnode(diagram1, [skip_AIDs, dataF_AID, (end_AID +
1):act_num_master]);
    NLabels = labels;
    plot(diagram1, 'Layout', 'layered', 'NodeLabel', NLabels);
    title(title1);

    % Reset
    data_pre = data_store;
end


%% Iterate from sample end
fprintf('    Reverse iteration commencing...\n');
% Iterate
data_pos = stateIterator(start_AID, targ_node_AID, dataF_AID, end_AID,
data_master, act_num_master, threshold, -1);

% Join and subset states
[data_pos, paths_pos] = stateEnforce(start_AID, dataF_AID, end_AID, data_pos,
act_num_master, -1);
act_num_pos = max(data_pos(:, 2));

% Calculate state paths
state_paths_store = [];
for path1_ID = 1:length(paths_pos)
    path1 = paths_pos{path1_ID};
    state_strt = path1(1);
    state_end = path1(end);

    % Append state path to set
    if isempty(state_paths_store)
        state_paths_store = [state_strt, state_end];
        path_map_store = [path1_ID, 1];
    else
        path2_ID = 1;
        while path2_ID <= size(state_paths_store, 1)
            path2 = state_paths_store(path2_ID, :);
            if isequal([state_strt, state_end], path2)
                break
            end
            path2_ID = path2_ID + 1;
        end
        if path2_ID > size(state_paths_store, 1)
            state_paths_store = [state_paths_store; state_strt, state_end];
        end
        path_map_store = [path_map_store; path1_ID, path2_ID];
    end
end
```

```
state_paths_pos = sortrows(state_paths_store);
path_map_pos = path_map_store;

for path1_ID = 1:size(state_paths_pos, 1)
    path1 = state_paths_pos(path1_ID, :);
    path2_ID = 1;
    while path2_ID <= size(state_paths_store, 1)
        path2 = state_paths_store(path2_ID, :);
        if isequal(path1, path2)
            break
        end
        path2_ID = path2_ID + 1;
    end

    for path3_ID = 1:size(path_map_store)
        if path_map_store(path3_ID, 2) == path2_ID
            path_map_pos(path3_ID, 2) = path1_ID;
        end
    end
end

fprintf('    Reverse iteration complete\n');

% Correct State IDs
act_num = act_num_pos + act_num_pre - act_num_master;
for seq1_ID = find(data_pos(:, 2) > act_num_master)'
    data_pos(seq1_ID, 2) = data_pos(seq1_ID, 2) - act_num_master +
act_num_pre;
end
for path_ID = 1:size(paths_pos, 1)
    path = paths_pos{path_ID};
    if not(path(1) == start_AID)
        path(1) = path(1) - act_num_master + act_num_pre;
    end
    if not(path(end) == end_AID)
        path(end) = path(end) - act_num_master + act_num_pre;
    end
    paths_pos{path_ID} = path;
end

% Update labels
for state1 = 1:(act_num_pos - act_num_master)
    NLabel = stateLabel;
    NLabel = strcat(NLabel, num2str(state1 + act_num_pre - act_num_master));
    labels_master = [labels_master{:}, {NLabel}]';
end


%% Visualize Post Iteration
if toggle_plot_pos
    %% Reset data
    % Instance storage
    strt_node_AID = start_AID;
    end_node_AID = end_AID;
```

```
    strt_seq_IDs = [];                                % Sequence ID for start
nodes
    end_seq_IDs = [];                                 % Sequence ID for end
nodes
    act_cnt = zeros(act_num, 1);

    % Remove actions
    data_store = data_pos;
    seq1_ID = 1;
    while seq1_ID <= size(data_pos, 1)
        if ismember(data_pos(seq1_ID, 2), [dataF_AID, (end_AID +
1):act_num_master])
            data_pos(seq1_ID, :) = [];
        else
            seq1_ID =seq1_ID + 1;
        end
    end

    % Scan for sample terminals
    [strt_seq_IDs, end_seq_IDs, act_cnt] = terminalSeek(data_pos,
strt_node_AID, targ_node_AID, end_node_AID, end_AID, act_num);

    % Determine actions with low measurable behavior
    skip_AIDs = [];
    for act1 = (act_num_master + 1):act_num
        if (act_cnt(act1) <= threshold) && not(ismember(act1, skip_AIDs)) &&
not(act1 == dataF_AID)
            skip_AIDs = [skip_AIDs, act1];
        end
    end
    skip_AIDs = sort(skip_AIDs);

    % Add states to labels
    labels = {};
    for act1 = 1:act_num
        if not(ismember(act1, [skip_AIDs, dataF_AID, (end_AID +
1):act_num_pre]))
            labels = [labels(:)', labels_master(act1)];
        end
    end
    labels = labels';

    figure(2)
    if toggle_mand
        title2 = 'Mandatory Post States';
        % Set edges
        act_path_pos = globalPathPos(data_pos, act_num, strt_seq_IDs,
end_seq_IDs, dataF_AID, skip_AIDs);
        act_mand_pos = act_path_pos(:, 2);

        % Mandatory post action model
        disp_path2 = []; % Construct edges
        for act1 = 1:act_num
            for act2 = act_mand_pos{act1}
                disp_path2 = [disp_path2; act1, act2];
            end
```

```
        end
    else
        title2 = 'State Diagram as Determined by Reverse Iteration';
        % Set edges
        disp_path2 = [];
        for seq1_ID = 1:(size(data_pos, 1) - 1)
            state1 = data_pos(seq1_ID, 2);
            if not(state1 == end_AID)
                state2 = data_pos((seq1_ID + 1), 2);
                if not(isequal([state1, state2], [start_AID, end_AID]))
                    if isempty(disp_path2)
                        disp_path2 = [state1, state2];
                    else
                        match = 0;
                        edge_ID = 1;
                        while (edge_ID <= size(disp_path2, 1)) && not(match)
                            if isequal(disp_path2(edge_ID, :), [state1,
state2])
                                match = 1;
                            end
                            edge_ID = edge_ID + 1;
                        end
                        if not(match)
                            disp_path2 = [disp_path2; state1, state2];
                        end
                    end
                end
            end
        end
        disp_path2 = sortrows(disp_path2);
    end

    diagram2 = digraph(disp_path2(:, 1), disp_path2(:, 2));
    diagram2 = rmnode(diagram2, [skip_AIDs, dataF_AID, (end_AID +
1):act_num_pre]);
    NLabels = labels;
    plot(diagram2, 'Layout', 'layered', 'NodeLabel', NLabels);
    title(title2);

    % Reset
    data_pos = data_store;
end

if toggle_ana
    %% Analyze path user behavior
    for direction = [-1, 1]
        switch direction
            case -1
                paths_dir = paths_pos;
                data_dir = data_pos;
                state_list = [start_AID, end_AID, (act_num_pre +
1):act_num_pos];
                path_map_dir = path_map_pos;
            case 1
                paths_dir = paths_pre;
                data_dir = data_pre;
```

```
                state_list = [start_AID, end_AID, (act_num_master +
1):act_num_pre];
                path_map_dir = path_map_pre;
        end

        % Analyze data
        [concen_p_dir, time_p_dir, prob_p_dir, concen_s_dir, time_s_dir,
prob_s_dir] = pathAnalyze(data_dir, paths_dir, path_map_dir, start_AID,
dataF_AID, end_AID, act_num_master);

        switch direction
            case -1
                concen_paths_pos = concen_p_dir;
                time_paths_pos = time_p_dir;
                prob_paths_pos = prob_p_dir;
                concen_state_pos = concen_s_dir;
                time_state_pos = time_s_dir;
                prob_state_pos = prob_s_dir;
            case 1
                concen_paths_pre = concen_p_dir;
                time_paths_pre = time_p_dir;
                prob_paths_pre = prob_p_dir;
                concen_state_pre = concen_s_dir;
                time_state_pre = time_s_dir;
                prob_state_pre = prob_s_dir;
        end
    end
    fprintf('    User data collection complete\n');
end
fprintf('Analysis complete\n');
```

collectPaths.mat

This function collects a list of all the path seen in the data set, outputting a cell array containing all the paths as described in the `CoffeeMiner.mat` section.

```
function paths = collectPaths(dataF_AID, end_AID, act_num_master, data)
    %% collectPaths 01
    %  Collects list of paths in data set

    %% Prep storage
    paths = {};

    %% Iterate through data
    seq1_ID = 1;
    while seq1_ID < size(data, 1)
        state1 = data(seq1_ID, 2);
        if state1 == end_AID
            seq1_ID = seq1_ID + 1;
        end

        % Find end of path
```

```
        seq2_ID = seq1_ID + 1;
        state2 = 0;
        while not(state2) && (seq2_ID <= size(data, 1))
            state2 = data(seq2_ID, 2);
            if not(state2 == end_AID) && not(state2 > act_num_master)
                state2 = 0;
                seq2_ID  = seq2_ID + 1;
            end
        end
        path = data(seq1_ID:seq2_ID, 2);

        if not(ismember(dataF_AID, path))
            % Store path
            if isempty(paths)
                paths = {path};
            else
                % Check inclusion in paths
                match = 0;
                path1_ID = 1;
                while not(match) && (path1_ID <= length(paths))
                    path1 = paths{path1_ID};
                    if isequal(path, path1)
                        match = 1;
                    else
                        path1_ID = path1_ID + 1;
                    end
                end

                if not(match)
                    paths = sortPaths([paths; {path}]);
                end
            end
        end

        seq1_ID = seq2_ID;
    end

end
```

globalPathPre.mat

This script collects the inputs and states that are mandatory for other inputs. The output, `act_mand_pre`, is a cell array with each row index corresponding to an input ID. The first column includes a list of inputs and states that occur prior to the row ID input in every use instance, and the second indicates those that always occur after the row ID input.

```
function act_mand_pre = globalPathPre(data, act_num, smpl_strt_IDs,
smpl_end_IDs, dataF_AID, skip_AIDs)
    %% Analyze global mandatory path
    %  Outputs complete paths as well as reduced paths
    % Instance storage
```

```
    act_cnt = zeros(act_num, 1);
    path_glob_pre = zeros(act_num) + 1;       % Switches for mandatory
inclusion in previous path
    absent = zeros(act_num, 1);

    % Scan path
    for smpl_ID = 1:length(smpl_strt_IDs)
        start_ID = smpl_strt_IDs(smpl_ID);
        end_ID = smpl_end_IDs(smpl_ID);
        smpl = data(start_ID:end_ID, :);

        for act1 = 1:act_num
            if not(ismember(act1, smpl))% Mark if not present
                absent(act1) = absent(act1) + 1;
            end
        end

        for act1_ID = 1:size(smpl, 1)
            act1 = smpl(act1_ID, 2);
            if (act1 == dataF_AID) || ismember(act1, skip_AIDs)
                continue
            end
            path_prev = smpl(1:(act1_ID - 1), 2);
            if not(ismember(dataF_AID, path_prev))
                for act2 = 1:act_num
                    if not(ismember(act2, path_prev)) || ismember(act2,
skip_AIDs)
                        path_glob_pre(act1, act2) = 0;
                    end
                end
            end
        end
    end
    for act1 = 1:act_num % Convert absent to boolean
        if absent(act1) == length(smpl_strt_IDs)
             absent(act1) = 1;
        else
            absent(act1) = 0;
        end
    end

    %% Collect mandatory actions
    % Instance storage
    act_mand_pre = {};

    % Begin iteration
    for act1 = 1:act_num
        act_mand_pre = [act_mand_pre(:)', {[]}];
        if not(absent(act1))
            for act2 = 1:act_num
                if path_glob_pre(act1, act2) == 1
                    act_mand_pre{act1} = [act_mand_pre{act1}, act2];
                end
            end
        end
    end
```

```
act_mand_pre = act_mand_pre';


%% Reduce Routes
% Prep cells
act_pre = cell(act_num, 2);
for act1 = 1:act_num
    act_pre{act1, 1} = act_mand_pre{act1};
end
act_mand_pre = act_pre;

% Reduce mandatory actions (Previous)
for act1 = 1:act_num % Collect actions which act1 is a component of
    state1_pre = [act_mand_pre{act1}, act1];
    for act2 = 1:act_num
        state2_pre = [act_mand_pre{act2}, act2];
        if all(ismember(state1_pre, state2_pre)) && not(act2 == act1)
            act_mand_pre{act1, 2} = [act_mand_pre{act1, 2}, act2];
        end
    end
end

for act1 = 1:act_num % Reduce component actions to direct routes
    comp1 = act_mand_pre{act1, 2};
    remove = zeros(1, length(comp1));
    for act2_ID = 1:length(comp1) % Seek through component actions
        act2 = comp1(act2_ID);
        comp2 = act_mand_pre{act2, 2};
        act2_ID_store = act2_ID;

        for act3 = comp2 % Seek through component actions component
actions
            for act2_ID = 1:length(comp1) % Seek matching component
actions
                act2 = comp1(act2_ID);
                if act3 == act2
                    remove(act2_ID) = 1;
                end
            end
        end
        act2_ID = act2_ID_store;
        act2 = comp1(act2_ID);
    end

    comp1_store = comp1;
    comp1 = []; % Remove redudant actions
    for act2_ID = 1:length(remove)
        if not(remove(act2_ID))
           comp1 = [comp1, comp1_store(act2_ID)];
        end
    end
    act_mand_pre{act1, 3} = comp1;
end

act_mand_pre = act_mand_pre(:, [1, 3]);
```

```
end
```

globalPathPos.mat

This script functions identically to `globalPathPre.mat` but operates on data sets in reverse iteration.

```
function act_mand_pos = globalPathPos(data, act_num, smpl_strt_IDs,
smpl_end_IDs, dataF_AID, skip_AIDs)
    %% globalPathPos 01
    %       Combines paths at sample end

    act_cnt = zeros(act_num, 1);
    path_glob_pos = zeros(act_num) + 1;        % Switches for mandatory
inclusion in following path
    absent = zeros(act_num, 1);

    % Scan path
    for smpl_ID = 1:length(smpl_strt_IDs)
        start_ID = smpl_strt_IDs(smpl_ID);
        end_ID = smpl_end_IDs(smpl_ID);
        smpl = data(start_ID:end_ID, :);

        for act1 = 1:act_num
            if not(ismember(act1, smpl))% Mark if not present
                absent(act1) = absent(act1) + 1;
            end
        end

        for act1_ID = 1:size(smpl, 1)
            act1 = smpl(act1_ID, 2);
            if (act1 == dataF_AID) || ismember(act1, skip_AIDs)
                continue
            end
            path_pos = smpl((act1_ID + 1):size(smpl, 1), 2);
            if not(ismember(dataF_AID, path_pos))
                for act2 = 1:act_num
                    if not(ismember(act2, path_pos)) || ismember(act2,
skip_AIDs)
                        path_glob_pos(act1, act2) = 0;
                    end
                end
            end
        end
    end
    for act1 = 1:act_num % Convert absent to boolean
        if absent(act1) == length(smpl_strt_IDs)
             absent(act1) = 1;
        else
```

```
            absent(act1) = 0;
        end
    end

    %% Collect mandatory actions
    % Instance storage
    act_mand_pos = {};

    % Begin iteration
    for act1 = 1:act_num
        act_mand_pos = [act_mand_pos(:)', {[]}];
        if not(absent(act1))
            for act2 = 1:act_num
                if path_glob_pos(act1,act2) == 1
                    act_mand_pos{act1} = [act_mand_pos{act1}, act2];
                end
            end
        end
    end
    act_mand_pos = act_mand_pos';


    %% Reduce Routes
    % Prep cells
    act_pos = cell(act_num, 2);
    for act1 = 1:act_num
        act_pos{act1, 1} = act_mand_pos{act1};
    end
    act_mand_pos = act_pos;

    % Reduce mandatory actions (Post)
    for act1 = 1:act_num % Collect actions which act1 is a component of
        state1_pos = [act_mand_pos{act1}, act1];
        for act2 = 1:act_num
            state2_pos = [act_mand_pos{act2}, act2];
            if all(ismember(state1_pos, state2_pos)) && not(act2 == act1)
                act_mand_pos{act1, 2} = [act_mand_pos{act1, 2}, act2];
            end
        end
    end

    for act1 = 1:act_num % Reduce component actions to direct routes
        comp1 = act_mand_pos{act1, 2};
        remove = zeros(1, length(comp1));
        for act2_ID = 1:length(comp1) % Seek through component actions
            act2 = comp1(act2_ID);
            comp2 = act_mand_pos{act2, 2};
            act2_ID_store = act2_ID;

            for act3 = comp2 % Seek through component actions component
actions
                for act2_ID = 1:length(comp1) % Seek matching component
actions
                    act2 = comp1(act2_ID);
                    if act3 == act2
```

```
                    remove(act2_ID) = 1;
                end
            end
        end
        act2_ID = act2_ID_store;
        act2 = comp1(act2_ID);
    end

    comp1_store = comp1;
    comp1 = []; % Remove redudant actions
    for act2_ID = 1:length(remove)
        if not(remove(act2_ID))
           comp1 = [comp1, comp1_store(act2_ID)];
        end
    end
    act_mand_pos{act1, 3} = comp1;
end

act_mand_pos = act_mand_pos(:, [1, 3]);
end
```

localPath.mat

This function produces simple Markov chain analysis of the trace behavior. Variable path_T1_pos stores the probabilities of an input being made given the previous input. This information is stored in a matrix, with each potential sequence of inputs stored in a separate row, such that the first index in the row stores the probability, the second index stores the second input, and the third index stores the initial input.

Variable path_T2_pos stores information similarly, only calculating probabilities given two known inputs instead of one. Each row of the matrix is the probability, final input, initial input, and second input.

```
function [path_T1_pos, path_T2_pos] = localPath(data, strt_seq_IDs,
end_seq_IDs, dataF_AID, skip_AIDs)
    %% Analyze Local Path
    %  Iterates through local path in specified sample data
    % Instance storage
    path_T1_pos = [];   % Probability (3) of action (2) following given
actions (1)
    path_T2_pos = [];   % Probability (4) of action (3) following given
actions (1-2)

    %% Scan path
    % Iterate through samples
    for smpl_ID = 1:length(strt_seq_IDs)
        strt_seq_ID = strt_seq_IDs(smpl_ID);
        end_seq_ID = end_seq_IDs(smpl_ID);
```

```
        for seq_ID = strt_seq_ID:end_seq_ID
            act1 = data(seq_ID, 2);
            if (seq_ID == end_seq_ID) || (act1 == dataF_AID) ||
ismember(act1, skip_AIDs)
                continue
            elseif seq_ID == (end_seq_ID - 1)
                act2 = data((seq_ID + 1), 2);
                if (act2 == dataF_AID) || ismember(act2, skip_AIDs)
                    act2 = 0;
                end
                act3 = 0;
            else
                act2 = data(seq_ID + 1, 2);
                if (act2 == dataF_AID) || ismember(act2, skip_AIDs)
                    act2 = 0;
                end
                act3 = data(seq_ID + 2, 2);
                if (act3 == dataF_AID) || ismember(act3, skip_AIDs)
                    act3 = 0;
                end
            end

            % Append to first order path
            if act2
                if not(isempty(path_T1_pos))
                    match = 0;
                    for check_index = 1:size(path_T1_pos, 1)
                        path = path_T1_pos(check_index, :);
                        if (path(1) == act1) && (path(2) == act2)
                            path_T1_pos(check_index, 3) =
path_T1_pos(check_index, 3) + 1;
                            match = 1;
                            break
                        end
                    end
                    if not(match)
                        path_T1_pos = [path_T1_pos; [act1, act2 , 1]];
                    end
                else
                    path_T1_pos = [act1, act2, 1];
                end
            end

            % Append to second order path
            if act3
                if not(isempty(path_T2_pos))
                    match = 0;
                    for check_index = 1:size(path_T2_pos, 1)
                        path = path_T2_pos(check_index, :);
                        if (path(1) == act1) && (path(2) == act2) && (path(3)
== act3)
                            path_T2_pos(check_index, 4) =
path_T2_pos(check_index, 4) + 1;
                            match = 1;
                            break
                        end
```

```
                end
                if not(match)
                    path_T2_pos = [path_T2_pos; [act1, act2 ,act3 ,1]];
                end
            else
                path_T2_pos = [act1, act2, act3, 1];
            end
        end
    end
end

%% Sort and convert to probability
path_T1_pos = sortrows(path_T1_pos);
path_T2_pos = sortrows(path_T2_pos);

path_ID = 1;
while path_ID <= size(path_T1_pos, 1)
    path = path_T1_pos(path_ID, :);
    sum = 0;
    match_path_ID = path_ID;
    match_path = path_T1_pos(match_path_ID, :);
    while path(1) == match_path(1)
        sum = sum + match_path(3);
        match_path_ID = match_path_ID + 1;
        if match_path_ID > size(path_T1_pos, 1)
            break
        end
        match_path = path_T1_pos(match_path_ID, :);
    end
    path_T1_pos(path_ID:(match_path_ID - 1), 3) =
path_T1_pos(path_ID:(match_path_ID - 1), 3) / sum;
    path_ID = match_path_ID;
end

path_ID = 1;
while path_ID <= size(path_T2_pos, 1)
    path = path_T2_pos(path_ID, :);
    sum = 0;
    match_path_ID = path_ID;
    match_path = path_T2_pos(match_path_ID, :);
    while path(1:2) == match_path(1:2)
        sum = sum + match_path(4);
        match_path_ID = match_path_ID + 1;
        if match_path_ID > size(path_T2_pos, 1)
            break
        end
        match_path = path_T2_pos(match_path_ID, :);
    end
    path_T2_pos(path_ID:(match_path_ID - 1), 4) =
path_T2_pos(path_ID:(match_path_ID - 1), 4) / sum;
    path_ID = match_path_ID;
end
end
```

stateSubset.mat

This function reads the trace, determines if paths subset each other, modifies path definitions to remove subsetting, and then outputs a trace with the updated state palcements.

```
function data = stateSubset(start_AID, dataF_AID, end_AID, data, paths,
act_num_master, direction)
    %% stateSubset 01
    % Detects when paths between states overlap

    %% Prep data
    % Find sample terminals
    strt_smpl_seq_IDs = find(data(:, 2) == start_AID);
    end_smpl_seq_IDs = find(data(:, 2) == end_AID);

    if direction  == 1
        %% Find new paths for forward iterated states
        % Collect paths by state
        paths_by_state = {};
        strt_states = [];
        end_states = {};
        path1_ID = 1;
        while path1_ID <= length(paths)
            path1 = paths{path1_ID};
            strt_state1 = path1(1);
            path2_ID = path1_ID + 1;
            state_match = 1;
            while state_match && (path2_ID < length(paths))
                path2 = paths{path2_ID};
                strt_state2 = path2(1);
                if not(strt_state1 == strt_state2)
                    state_match = 0;
                else
                    path2_ID = path2_ID + 1;
                end
            end

            % Isolate paths
            paths_out = paths(path1_ID:(path2_ID - 1));
            end_states_sub = [];
            for path3_ID = 1:length(paths_out)
                path3 = paths_out{path3_ID};
                end_state3 = path3(end);
                if not(ismember(end_state3, end_states_sub))
                    end_states_sub = [end_states_sub; end_state3];
                end
            end

            % Store paths
            strt_states = [strt_states; strt_state1];
            end_states = [end_states; {end_states_sub}];
            paths_by_state = [paths_by_state; {paths_out}];
```

```
            path1_ID = path2_ID;
        end

        % Compare paths between states
        rem_paths = {};
        for state_sub_ID = 1:length(strt_states)
            state_sub = strt_states(state_sub_ID);
            end_states_sub = end_states{state_sub_ID};
            paths_sub = paths_by_state{state_sub_ID};
            for state_sup_ID = 1:length(strt_states)
                if not(state_sub_ID == state_sup_ID) % Skip self subsetting
                    state_sup = strt_states(state_sup_ID);
                    end_states_sup = end_states{state_sup_ID};
                    paths_sup = paths_by_state{state_sup_ID};
                    if all(ismember(end_states_sub, end_states_sup)) &&
(length(paths_sup) >= length(paths_sub)) % Check possibility of inclusion
                        % Collect remainder paths
                        rem_paths1 = {};
                        rem_reach_sub = {};
                        for path_sub_ID = 1:length(paths_sub)
                            path_sub = paths_sub{path_sub_ID};
                            len_sub = length(path_sub);
                            for path_sup_ID = 1:length(paths_sup)
                                path_sup = paths_sup{path_sup_ID};
                                len_sup = length(path_sup);
                                if len_sup > len_sub
                                    path_comp = path_sup((end - len_sub +
2):end);
                                    if isequal(path_comp, path_sub(2:end))
                                        rem_path1 = path_sup(1:(end - len_sub
+ 1));

                                        rem_path_ID = 1;
                                        while rem_path_ID <=
length(rem_paths1)
                                            rem_path =
rem_paths1{rem_path_ID};
                                            if isequal(rem_path, rem_path1)
                                                break
                                            end
                                            rem_path_ID = rem_path_ID + 1;
                                        end

                                        if rem_path_ID > length(rem_paths1)
                                            rem_paths1 = [rem_paths1;
{rem_path1}];
                                            rem_reach_sub = [rem_reach_sub;
{[]}];
                                        end

                                        rem_reach_sub{rem_path_ID} =
[rem_reach_sub{rem_path_ID}; path_sub_ID];
                                    end
                                end
                            end
                        end
```

```
                        % Determine remainder path utility
                        rem_paths1_store = rem_paths1;
                        rem_paths1 = {};
                        for rem_path1_ID = 1:length(rem_paths1_store)
                            rem_path1 = rem_paths1_store{rem_path1_ID};
                            rem_reach_sub1 = rem_reach_sub{rem_path1_ID};
                            if all(ismember(1:length(paths_sub),
rem_reach_sub1)) % Check if remainder path reaches all sub paths
                                in_path_sups = [];
                                for path_sup_ID = 1:length(paths_sup)
                                    path_sup = paths_sup{path_sup_ID};
                                    if length(path_sup) > length(rem_path1)
                                        if isequal(rem_path1,
path_sup(1:length(rem_path1)))
                                            in_path_sups = [in_path_sups;
path_sup_ID];
                                        end
                                    end
                                end

                                if length(in_path_sups) ==
length(paths_sub) % Check if remainder path does not lead to non-sub paths
                                    rem_paths1 = [rem_paths1; {rem_path1}];
                                end
                            end
                        end

                        % Store results
                        for rem_path1_ID = 1:length(rem_paths1)
                            rem_path1 = [rem_paths1{rem_path1_ID};
state_sub];
                            rem_paths = [rem_paths; {rem_path1}];
                        end
                    end
                end
            end
        end

        %% Merge with paths
        paths_store = sortPaths([paths; rem_paths]);
        paths = {};
        path1_ID = 1;
        while path1_ID < length(paths_store)
            path1 = paths_store{path1_ID};
            state1 = path1(1);
            path2_ID = path1_ID + 1;
            while path2_ID <= length(paths_store)
                path2 = paths_store{path2_ID};
                state2 = path2(1);
                if not(state1 == state2)
                    break
                else
                    path2_ID = path2_ID + 1;
                end
            end
```

```
        bound1 = path1_ID;
        bound2 = path2_ID - 1;


        for path1_ID = bound1:bound2
            do_not_inc = 0;
            path1 = paths_store{path1_ID};
            for path3_ID = bound1:bound2
                path3 = paths_store{path3_ID};
                len3 = length(path3);
                if len3 < length(path1)
                    path_comp = path1(1:(len3 - 1));
                    if isequal(path_comp, path3(1:(end - 1)))
                        do_not_inc = 1;
                        break
                    end
                end
            end

            if not(do_not_inc)
                paths = [paths; {path1}];
            end
        end

        path1_ID = path2_ID;
    end
end

% Collect paths by state
paths_by_state = {};
strt_states = [];
end_states = {};
path1_ID = 1;
while path1_ID <= length(paths)
    path1 = paths{path1_ID};
    strt_state1 = path1(1);
    path2_ID = path1_ID + 1;
    state_match = 1;
    while state_match && (path2_ID < length(paths))
        path2 = paths{path2_ID};
        strt_state2 = path2(1);
        if not(strt_state1 == strt_state2)
            state_match = 0;
        else
            path2_ID = path2_ID + 1;
        end
    end

    % Isolate paths
    paths_out = paths(path1_ID:(path2_ID - 1));
    end_states_out = [];
    for path3_ID = 1:length(paths_out)
        path3 = paths_out{path3_ID};
        end_state3 = path3(end);
        if not(ismember(end_state3, end_states_out))
            end_states_out = [end_states_out; end_state3];
```

```
            end
        end

        % Store paths
        strt_states = [strt_states; strt_state1];
        end_states = [end_states; {end_states_out}];
        paths_by_state = [paths_by_state; {paths_out}];

        path1_ID = path2_ID;
    end

    %% Store data
    % Iterate through samples
    data_store = data;
    data = [];
    for smpl_ID = 1:length(strt_smpl_seq_IDs)
        strt_smpl_seq_ID = strt_smpl_seq_IDs(smpl_ID);
        end_smpl_seq_ID = end_smpl_seq_IDs(smpl_ID);
        smpl_data = data_store(strt_smpl_seq_ID:end_smpl_seq_ID, :);
        smpl = smpl_data(:, 2);

        % Iterate through single sample
        path_strt_ID = 1;
        smpl_end_ID = length(smpl);
        while path_strt_ID < smpl_end_ID
            path_strt_AID = smpl(path_strt_ID);

            % Collect current path
            path_end_ID = path_strt_ID + 1;
            path_end_AID = smpl(path_end_ID);
            while not(path_end_AID == end_AID) && not(path_end_AID >
act_num_master)
                path_end_ID = path_end_ID + 1;
                path_end_AID = smpl(path_end_ID);
            end
            path_curr = smpl(path_strt_ID:path_end_ID);

            % Collect known paths from start state
            if path_strt_AID == start_AID
                path_set = paths_by_state{1};
            else
                path_set = paths_by_state{path_strt_AID - act_num_master +
1};
            end

            % Iterate through current path
            match = 0;
            seq1_ID = 1;
            while (seq1_ID < length(path_curr)) && not(match)
                path_seg = path_curr(1:seq1_ID);
                % Match current path segment against known paths
                path_kno_ID = 1;
                while (path_kno_ID <= length(path_set)) && not(match)
                    path_kno = path_set{path_kno_ID};
```

```
                    if (isequal(path_seg, path_kno(1:(end - 1)))) &&
not(length(path_kno) ==  length(path_curr))
                        match = 1;
                    else
                        path_kno_ID = path_kno_ID + 1;
                    end
                end
                if not(match)
                    seq1_ID = seq1_ID + 1;
                end
            end

            % Update sample
            if match
                path_strt_ID = path_strt_ID + seq1_ID;
                smpl_data_store = smpl_data;
                smpl_data = smpl_data_store(1:(path_strt_ID - 1), :);
                smpl_data = [smpl_data; smpl_data(end, :)];
                smpl_data(end, 2) = path_kno(end);
                smpl_data = [smpl_data;
smpl_data_store(path_strt_ID:end, :)];
                smpl = smpl_data(:, 2);
                smpl_end_ID = length(smpl);
                capture = 1;
            else
                path_strt_ID = path_end_ID;
            end
        end

        % Store data
        if isempty(data)
            data = smpl_data;
        else
            data = [data; smpl_data];
        end
    end
end
```

stateEnforce.mat

This function merges preliminary states and subsets paths iteratively until paths are consistent and no further simplifications can be made to the state machine. Output `data1` is the updated trace with new state definitions, and `paths1` is the new path set extracted from the trace.

```
function [data1, paths1] = stateEnforce(start_AID, dataF_AID, end_AID,
data_master, act_num_master, direction)
    %% stateEnforce 01
    % Forces state definitions and determinism to apply uniformly
```

```
    %% Iterate
    fprintf('        State enforcement commencing...\n');
    % Prep iterators
    complete = 0;
    data1 = data_master;
    paths1 = collectPaths(dataF_AID, end_AID, act_num_master, data1);

    % Begin enforcement
    while not(complete)
        [data2, paths2] = stateJoin(start_AID, dataF_AID, end_AID, data1,
act_num_master, direction);
        data3 = stateSubset(start_AID, dataF_AID, end_AID, data2, paths2,
act_num_master, direction);
        paths3 = collectPaths(dataF_AID, end_AID, act_num_master, data3);

        if direction == -1
            capture = 1;
        end

        if isequal(data3, data1)
            complete = 1;
        else
            data1 = data3;
            paths1 = paths3;
        end
    end
    fprintf('        State enforcement complete\n');
end
```

stateIterator.mat

This function examines the trace and detects states with the global and local contexts and appends detected states to the trace. This trace is then the output.

```
function data = stateIterator(start_AID, targ_node_AID, dataF_AID, end_AID,
data_master, act_num_master, threshold, direction)
    %% Prep Path Iterator
    %  Detect states iteratively and add to data set
    %  Changes from 01
    %      - Changed how states were stored to require matching paths in
    %      - Requires that next node not be start node when creating states

    % Start conditions
    act_num = act_num_master;
    complete = [0];
    switch direction
        case 1
            end_node_AID = end_AID;
            data = data_master;
            new_state_AIDs = [start_AID];
```

```
        case -1
            end_node_AID = start_AID;
            data = flipud(data_master);
            new_state_AIDs = [end_AID];
    end


    %% Iterate
    while not(all(complete))
%         fprintf('    New States %d\n', length(new_state_AIDs));
        strt_node_list = new_state_AIDs;
        end_node_list = end_node_AID;
        new_state_AIDs = [];
        path_list = [];
        for strt_node_AID = strt_node_list
            for end_node_AID = end_node_list
                path_list = [path_list; strt_node_AID, end_node_AID];
            end
        end
        complete = zeros(size(path_list, 1), 1);

        path_ID = 1;
        while path_ID <= length(complete)
            %% Initial Scan
            % Instance action IDs
            strt_node_AID = path_list(path_ID, 1);
            end_node_AID = path_list(path_ID, 2);
            state_AIDs = (act_num_master + 1):act_num;

            % Scan for sample terminals
            switch direction
                case 1
                    [strt_seq_IDs, end_seq_IDs, act_cnt] = terminalSeek(data,
strt_node_AID, targ_node_AID, end_node_AID, end_AID, act_num);
                case -1
                    [strt_seq_IDs, end_seq_IDs, act_cnt] = terminalSeek(data,
strt_node_AID, targ_node_AID, end_node_AID, start_AID, act_num);
            end

            % Determine actions with low measurable behavior
            skip_AIDs = [];
            for act1 = 1:act_num
                if (act_cnt(act1) <= threshold) && not(ismember(act1,
skip_AIDs)) && not(act1 == dataF_AID)
                    skip_AIDs = [skip_AIDs, act1];
                end
            end
            skip_AIDs = sort(skip_AIDs);
            if isempty(strt_seq_IDs)
    %              fprintf('        ERROR: Path %s does not exist\n',
labels_master{strt_node_AID});
            elseif length(skip_AIDs) > (act_num - 2)
    %              fprintf('        ERROR: Not enough data to characterize
behavior on %s\n', labels_master{strt_node_AID});
            else
                %% Analyze global mandatory path
```

```
    %             fprintf('        Path %s\n', labels_master{strt_node_AID});
                % Determine mandatory preceding actions and find states
                act_path_pre = globalPathPre(data, act_num, strt_seq_IDs,
end_seq_IDs, dataF_AID, skip_AIDs);
                act_mand_pre = act_path_pre(:, 2);
                act_num_pre = size(act_mand_pre, 1);
                act_mand_pre_root = cell(act_num_pre, 1);
                for act1 = 1:act_num_pre
                    branch1 = act_mand_pre{act1};
                    for act2 = branch1
                        root1 = act_mand_pre_root{act2, 1};
                        if not(ismember(act1, root1))
                            act_mand_pre_root{act2, 1} = [root1, act1];
                        end
                    end
                end

                %% Analyze local mandatory path
                [act_avail_pos_T1, ~] = localPath(data, strt_seq_IDs,
end_seq_IDs, dataF_AID, skip_AIDs);

                %% Update data set
                data_store = data;
                data = [];
                seq1_ID = 1;
                states = {};
                while seq1_ID <= size(data_store, 1)
                    data = [data; data_store(seq1_ID, :)];
                    if ismember(seq1_ID, strt_seq_IDs)
                        end_seq_ID = end_seq_IDs(strt_seq_IDs == seq1_ID);
                        while seq1_ID <= end_seq_ID
                            if ismember(seq1_ID, strt_seq_IDs)
                                smpl = [];
                            else
                                data = [data; data_store(seq1_ID, :)];
                            end
                            smpl = [smpl; data_store(seq1_ID, :)];

                            if size(smpl, 1) > 1 % Only add states after
first action
                                act_avail_IDs = act_avail_pos_T1(:, 1) ==
data(end, 2);
                                act_out_local =
act_avail_pos_T1(act_avail_IDs, 2)';
                                act_out = [];
                                for act1 = act_out_local
                                    if all(ismember(act_path_pre{act1, 1},
smpl))
                                        act_out = [act_out, act1];
                                    end
                                end

                                switch direction
                                    case 1
                                        end_exclude = not(ismember(end_AID,
smpl(:, 2)));
```

```
                                        case -1
                                            end_exclude = not(ismember(start_AID,
smpl(:, 2)));
                                    end

                                    if not(ismember(dataF_AID, smpl(:, 2))) &&
end_exclude && (length(act_out) > 1) && not(ismember(data_store((seq1_ID +
1), 2), [start_AID, end_AID]))
                                        state1_ID = 1;
                                        while state1_ID <= length(states)
                                            if isequal(smpl(:, 2),
states{state1_ID})

                                                break
                                            end
                                            state1_ID = state1_ID + 1;
                                        end
                                        if state1_ID > length(states)
                                            states = [states(:)', smpl(:, 2)];
                                            act_num_pre = act_num_pre + 1;
                                            state1_AID = act_num_pre;
                                        else
                                            state1_AID = state1_ID + act_num;
                                        end
                                        data = [data; data_store(seq1_ID, 1),
state1_AID, data_store(seq1_ID, 3)];
                                        data = [data; data_store((seq1_ID +
1):end_seq_ID, :)];

                                        seq1_ID = end_seq_ID + 1;
                                    else
                                        seq1_ID = seq1_ID + 1;
                                    end

                                else
                                    seq1_ID = seq1_ID + 1;
                                end
                            end
                        else
                            seq1_ID = seq1_ID + 1;
                        end
                    end
                    new_state_num = act_num_pre - act_num;

                    % Update states
                    if new_state_num
                        for state1 = 1:new_state_num
                            new_state_AIDs = [new_state_AIDs, (act_num +
state1)];
                        end
                    end
                    act_num = max(data(:, 2));
                end


                %% Iterate
                if (length(skip_AIDs) > (act_num - 1)) || isempty(strt_seq_IDs)
|| (new_state_num == 0)
```

```
                complete(path_ID) = 1;
            end
            path_ID = path_ID + 1;
        end
    end

    if direction < 0
        data = flipud(data);
    end
end
```

stateJoin.mat

This function detects when states share identical paths, and relabels them to the same state ID. This updated trace is then output `data` and the updated path set is output `paths`.

```
function [data, paths] = stateJoin(start_AID, dataF_AID, end_AID, data,
act_num_master, direction)
    %% stateJoin 09
    %  Detects when states share identical action paths to future state
    %  and relabels them as the same state
    %  Requires:
    %      - sortPaths
    %  Changes from 08
    %      - Removes subsetting

    %% Prep data
    act_num = max(data(:, 2));
    switch direction
        case 1
            strt_node_AID = start_AID;
            end_node_AID = end_AID;
        case -1
            strt_node_AID = end_AID;
            end_node_AID = start_AID;
            data = flipud(data);
    end

    %% Find state to state paths
    % Iterate until all possible states merged
    reduce_states = 1;
    while reduce_states
        % Instance storage
        branches = cell(act_num - act_num_master + 2, 1);
        paths = cell(act_num - act_num_master + 2);

        % Iterate through start states
        states = [start_AID, end_AID, (act_num_master + 1):act_num];
        for state1 = states
            branches1 = [];
```

```
paths1 = cell(act_num - act_num_master + 2, 1);
if not(state1 == end_node_AID)
    for state1_seq_ID = find(data(:, 2) == state1)'
        state2_seq_ID = state1_seq_ID + 1;
        state2 = data(state2_seq_ID, 2);
        while not(ismember(state2, states))
            state2_seq_ID = state2_seq_ID + 1;
            state2 = data(state2_seq_ID, 2);
        end
        path = data((state1_seq_ID + 1):state2_seq_ID, 2);

        if not(ismember(dataF_AID, path))
            if not(ismember(state2, branches1))
                branches1 = [branches1, state2];
            end

            % Check if path recorded
            include = 0;
            path_ID = 1;
            if state2 > act_num_master
                paths1_2 = paths1{state2 - act_num_master + 2};
            elseif state2 == start_AID
                paths1_2 = paths1{1};
            else
                paths1_2 = paths1{2};
            end
            if isempty(paths1_2)
                paths1_2 = {path};
            else
                while path_ID <= length(paths1_2)
                    if isequal(path, paths1_2{path_ID})
                        include = 1;
                        break
                    end
                    path_ID = path_ID + 1;
                end
                if not(include)
                    paths1_2 = [paths1_2(:)', path];
                end
            end

            if state2 > act_num_master
                paths1{state2 - act_num_master + 2} = paths1_2;
            elseif state2 == start_AID
                paths1{1} = paths1_2;
            else
                paths1{2} = paths1_2;
            end
        end
    end
end
branches1 = sort(branches1);

% Store data
state1_ID = find(states == state1);
branches{state1_ID} = branches1;
```

```
            for state2_ID = 1:length(states)
                paths_list = paths1{state2_ID};
                % Sort paths
                if not(isempty(paths_list))
                    paths_list = sortPaths(paths_list')';
                end
                paths(state1_ID, state2_ID) = {paths_list};
            end
        end

        %% Detect shared paths
        % Prep iteration
        checked = zeros(act_num - act_num_master + 2, 1);
        convert_sink = [];
        convert_source = {};

        % Iterate through states
        for state1_ID = 1:length(states)
            state1 = states(state1_ID);
            branches1 = branches{state1_ID};

            % Find matching future states
            match_states = [];
            state2_ID = state1_ID + 1;
            while state2_ID <= length(states)
                if not(checked(state2_ID))
                    branches2 = branches{state2_ID};
                    if (isequal(branches1, branches2) && (direction == 1)) ||
(any(ismember(branches1, branches2)) && (direction == -1))
                        match_states = [match_states, states(state2_ID)];
                    end
                end
                state2_ID = state2_ID + 1;
            end

            % Check if paths match
            if not(isempty(match_states))
                paths1 = paths(state1_ID, :);
                match_states_store = match_states;
                match_states = [];

                if direction == -1
                    % Compile ongoing list of paths in possible super state
                    match_paths = {};
                    for state3_ID = 1:length(states)
                        paths1_3 = paths1{state3_ID};
                        for path_ID = 1:length(paths1_3)
                            match_paths = [match_paths, {paths1_3{path_ID}}];
                        end
                    end
                end

                % Iterate through states
                d_len = 1;
                while d_len > 0     % Iterate until no change is match_states
                    curr_len = length(match_states);
```

```
for state2 = match_states_store
    if ismember(state2, match_states)
        continue
    elseif state2 > act_num_master
        state2_ID = state2 - act_num_master + 2;
    elseif state2 == start_AID
        state2_ID = 1;
    else
        state2_ID = 2;
    end
    paths2 = paths(state2_ID, :);
    if isequal(paths1, paths2) && (direction == 1)
        match_states = [match_states, state2];
        checked(state2_ID) = 1;
    elseif direction == -1
        for state3_ID = 1:length(states)
            paths2_3 = paths2{state3_ID};
            path1_ID = 1;
            match_check = 0;

            % Find match
            while (path1_ID <= length(match_paths)) &&
not(match_check)
                path2_ID = 1;
                while path2_ID <= length(paths2_3) &&
not(match_check)
                    if isequal(match_paths{path1_ID},
paths2_3{path2_ID})
                        match_check = 1;
                    end
                    path2_ID = path2_ID + 1;
                end
                path1_ID = path1_ID + 1;
            end

            if match_check
                match_states = [match_states, state2];
                for path3_ID = 1:length(paths2_3)
                    path2_3 = paths2_3{path3_ID};
                    path_include = 0;
                    for path1_ID = 1:length(match_paths)
                        path1 = match_paths{path1_ID};
                        if isequal(path1, path2_3)
                            path_include = 1;
                            break
                        end
                    end
                    if not(path_include)
                        match_paths = [match_paths,
path2_3];
                    end
                end
                checked(state2_ID) = 1;
            end
        end
    end
```

```
                end
                match_states = sort(match_states);
                d_len = length(match_states) - curr_len;
            end
        end

        % Update match data
        checked(state1_ID) = 1;
        if not(isempty(match_states))
            convert_sink = [convert_sink, state1];
            convert_source = [convert_source(:)', match_states];
        end
    end
    convert_source = convert_source';


    %% Join states
    % Prep convert detection
    convert_sources = [];
    for state1_ID = 1:length(convert_source)
        convert_sources = [convert_sources, convert_source{state1_ID}];
    end
    convert_sources = sort(convert_sources);

    % Convert
    for seq1_ID = find(ismember(data(:, 2), convert_sources))'
        state1 = data(seq1_ID, 2);
        state2_ID = 1;
        while state2_ID <= length(convert_source)
            if ismember(state1, convert_source{state2_ID})
                data(seq1_ID, 2) = convert_sink(state2_ID);
                break
            end
            state2_ID = state2_ID + 1;
        end
    end


    %% Reduce states
    if not(isempty(convert_sources))
        % Reduce
        for seq1_ID = find(data(:, 2) > convert_sources(1))'
            state1 = data(seq1_ID, 2);
            reduce = nnz((convert_sources - state1) < 0); % Count number
of states removed
            data(seq1_ID, 2) = state1 - reduce;
        end

        % Reset trackers
        act_num = max(data(:, 2));
    else
        reduce_states = 0;
    end
end
```

```
    %% Correct output
    % Reverse data
    if direction == -1
        data = flipud(data);
        paths = paths';
    end

    % Reorganize paths
    paths_store = paths;
    paths = {};
    for state1_ID = 1:size(paths_store, 1)
        for state2_ID = 1:size(paths_store, 2)
            paths1_2 = paths_store{state1_ID, state2_ID};
            if not(isempty(paths1_2))
                for path_ID = 1:length(paths1_2)
                    path = paths1_2{path_ID};
                    if not(ismember(dataF_AID, path))
                        if direction == -1
                            path = flipud(path);
                            % Append end action ID
                            if state2_ID == 2
                                path = [path; end_AID];
                            else
                                path = [path; (act_num_master + state2_ID -
2)];
                            end
                        else
                            % Append start action ID
                            if state1_ID == 1
                                path = [start_AID; path];
                            else
                                path = [(act_num_master + state1_ID - 2);
path];
                            end
                        end

                        % Store
                        paths = [paths; path];
                    end
                end

            end
        end
    end

    % Sort paths
    paths = sortPaths(paths);
end
```

pathAnalyze.mat

This function provides the concentration analysis data for CoffeeMiner.mat. Variables concen_state_pre and concen_path_pre provide the percentage of use instances that included said state or path in forward iteration respectively. Variables probs_state_pre and probs_path_pre contain the respective probability of navigating to a given state or path respectively given an initial state. Rows in the cell array correspond to initial state and contain a two-column matrix. The first column of this matrix includes the corresponding end state or path ID, and the second column then includes the probability of navigation given the initial state. These variables also have corresponding variables storing the reverse iteration information but utilize the _pos suffix instead of the _pre.

```
function [concens_p, timings_p, probs_p, concens_s, timings_s, probs_s] =
pathAnalyze(data, paths, path_map, start_AID, dataF_AID, end_AID,
act_num_master)
    %% pathAnalyze 02
    % Collects concentration, timing, and probability data
    % Changes from 01
    %     - Computes data from states to state as well as path specifically
    %     - State based probability calculation

    %% Prep storage
    state_cnt = zeros(max(data(:, 2)), 1);
    smpl_cnt = 0;
    state_path_cnt = max(path_map(:, 2));

    concens_p = zeros(length(paths), 1);
    timings_p = cell(length(paths), 1);
    probs_p = cell(max(data(:, 2)), 1);

    concens_s = zeros(state_path_cnt, 1);
    timings_s = cell(state_path_cnt, 1);
    probs_s = cell(max(data(:, 2)), 1);


    %% Iterate through data
    % Find sample terminals
    strt_seq_IDs = find(data(:, 2) == start_AID);
    end_seq_IDs = find(data(:, 2) == end_AID);

    % Iterate through samples
    for smpl_ID = 1:length(strt_seq_IDs)
        strt_seq_ID = strt_seq_IDs(smpl_ID);
        end_seq_ID = end_seq_IDs(smpl_ID);
        smpl_data = data((strt_seq_ID:end_seq_ID), :);
        smpl_path = smpl_data(:, 2);
        if not(ismember(dataF_AID, smpl_path))
```

```
            smpl_cnt = smpl_cnt + 1;
            concen_p_smpl = zeros(length(paths), 1);
            concen_s_smpl = zeros(state_path_cnt, 1);
            % Isolate paths in sample
            path_strt_IDs = [1; find(smpl_path(:) > act_num_master)];
            for path1_ID = 1:length(path_strt_IDs)
                if path1_ID == 1 && smpl_ID == 7
                    gotcha = 1;
                end
                path1_strt_ID = path_strt_IDs(path1_ID);
                if path1_ID == length(path_strt_IDs)
                    path1_end_ID = length(smpl_path);
                else
                    path1_end_ID = path_strt_IDs(path1_ID + 1);
                end
                path1 = smpl_path(path1_strt_ID:path1_end_ID);
                strt_state = path1(1);
                state_cnt(strt_state) = state_cnt(strt_state) + 1;
                end_state = path1(end);

                % Match sub path to listed paths
                path2_ID = 1;
                while path2_ID <= length(paths)
                    path2 = paths{path2_ID};
                    if isequal(path1, path2)
                        break
                    end
                    path2_ID = path2_ID + 1;
                end
                path_ID_s = path_map(path2_ID, 2);

                % Store path data
                concen_p_smpl(path2_ID) = concen_p_smpl(path2_ID) + 1;
                timings_p{path2_ID} = [timings_p{path2_ID},
(smpl_data(path1_end_ID, 3) - smpl_data(path1_strt_ID, 3))];
                path_probs = probs_p{strt_state};
                if isempty(path_probs)
                    path_probs = [path2_ID, 1];
                else
                    row_ID = find(path_probs(:, 1) == path2_ID);
                    if isempty(row_ID)
                        path_probs = sortrows([path_probs; path2_ID, 1]);
                    else
                        path_probs(row_ID, 2) = path_probs(row_ID, 2) + 1;
                    end
                end
                probs_p{strt_state} = path_probs;

                % Store state path data
                concen_s_smpl(path_ID_s) = concen_s_smpl(path_ID_s) + 1;
                timings_s{path_ID_s} = [timings_s{path_ID_s},
(smpl_data(path1_end_ID, 3) - smpl_data(path1_strt_ID, 3))];
                state_probs = probs_s{strt_state};
                if isempty(state_probs)
                    state_probs = [end_state, 1];
                else
```

```
                    row_ID = find(state_probs(:, 1) == end_state);
                    if isempty(row_ID)
                        state_probs = sortrows([state_probs; end_state, 1]);
                    else
                        state_probs(row_ID, 2) = state_probs(row_ID, 2) + 1;
                    end
                end
                probs_s{strt_state} = state_probs;
            end

            % Store concentration data
            concens_p = concens_p + (concen_p_smpl > 0);
            concens_s = concens_s + (concen_s_smpl > 0);
        end
    end

    %% Adjust values
    % Concentrations
    concens_p = concens_p / smpl_cnt;
    concens_s = concens_s / smpl_cnt;

    % Timings
    timings_store_p = timings_p;
    timings_p = [];
    for path1_ID = 1:length(paths)
        path_timings = timings_store_p{path1_ID};
        timings_p = [timings_p; mean(path_timings), std(path_timings, 1)];
    end

    timings_store_s = timings_s;
    timings_s = [];
    for state_AID = 1:state_path_cnt
        path_timings = timings_store_s{state_AID};
        timings_s = [timings_s; mean(path_timings), std(path_timings, 1)];
    end

    % Probabilities
    for state_AID = 1:max(data(:, 2))
        if not(isempty(probs_p{state_AID}))
            probs_p{state_AID}(:, 2) = probs_p{state_AID}(:, 2) /
state_cnt(state_AID);
        end
        if not(isempty(probs_s{state_AID}))
            probs_s{state_AID}(:, 2) = probs_s{state_AID}(:, 2) /
state_cnt(state_AID);
        end
    end
end
```

pathClimber.mat

This is a supplementary script which simulates state machine navigation in both machines at once. Available inputs are provided to the user in sequence, who is then presented with the option to select between said inputs, which is then entered to the state machine.

```
%% Path Climber 01
%  Climbs the path tree for a system and provides available actions
%  Requires:

clc
clear

%% Set toggles
toggle_txt_labels = 1;

%% Instance data
data_master = load('CoffeeMachine_03.dat');          % Data set
start_AID = 1;                                       % Action ID for Start
action
dataF_AID = 2;                                       % Action ID for Data
Failure action
end_AID = 3;                                         % Action ID for End
action
act_num_master = max(data_master(:, 2));             % Number of individual
actions
load('paths_pre.mat');                               % Paths in forward
iteration
load('paths_pos.mat');                               % Paths in reverse
iteration
if toggle_txt_labels                                 % Action Labels
    labelFID = fopen('CoffeeActLabels_s.dat');
    stateLabel = 'State';
else
    labelFID = fopen('CoffeeActLabels_i.dat');
    stateLabel = 'S';
end
labels = textscan(labelFID, '%s', 'Delimiter', '\n');
labels_master = labels{1};
fclose(labelFID);

%% Climb tree
% Instance iteration variables
state_pre = 1;
state_pos = state_pre;
act_path_pre = [state_pre];
act_path_pos = [state_pos];

% Iterate through complete path
fprintf('Start:\n');
while not(state_pre == end_AID) && not(state_pos == end_AID)
    % Check if new paths needed
```

```
    for dir = [-1, 1]
        switch dir
            case -1
                act_path_dir = act_path_pos;
                paths_dir = paths_pos;
                state_dir = state_pos;
            case 1
                act_path_dir = act_path_pre;
                paths_dir = paths_pre;
                state_dir = state_pre;
        end
        if (act_path_dir(end) > act_num_master) || (act_path_dir(end) ==
start_AID)
            % Seek paths with current start state
            start_path_ID = 0;
            end_path_ID = 0;
            path_ID = 1;
            while path_ID <= length(paths_dir)
                curr_path = paths_dir{path_ID};
                if (curr_path(1) == state_dir) && not(start_path_ID)
                    start_path_ID = path_ID;
                elseif start_path_ID && not(curr_path(1) == state_dir)
                    end_path_ID = path_ID - 1;
                    break
                end
                path_ID = path_ID + 1;
            end
            if not(end_path_ID) % If no end detected
                end_path_ID = length(paths_dir);
            end

            % Store paths
            paths_avail = cell(end_path_ID - start_path_ID + 1, 1);
            for path_ID = start_path_ID:end_path_ID
                paths_avail{path_ID - start_path_ID + 1} =
paths_dir{path_ID};
            end

            switch dir
                case -1
                    paths_avail_pos = paths_avail;
                    curr_path_pos = [state_pos];
                    seq_ID_pos = 2;
                case 1
                    paths_avail_pre = paths_avail;
                    curr_path_pre = [state_pre];
                    seq_ID_pre = 2;
            end
        end
    end

    % Determine available actions
    for dir = [-1, 1]
        switch dir
            case -1
                seq_ID_dir = seq_ID_pos;
```

```
            paths_avail_dir = paths_avail_pos;
        case 1
            seq_ID_dir = seq_ID_pre;
            paths_avail_dir = paths_avail_pre;
    end

    act_avail_dir = [];
    for path_ID = 1:length(paths_avail_dir)
        act_AID = paths_avail_dir{path_ID}(seq_ID_dir);
        if not(ismember(act_AID, act_avail_dir))
            act_avail_dir = [act_avail_dir; act_AID];
        end
    end
    act_avail_dir = sort(act_avail_dir);

    switch dir
        case -1
            act_avail_pos = act_avail_dir;
        case 1
            act_avail_pre = act_avail_dir;
    end
end

act_avail = [];
for act_AID = act_avail_pos'
    if ismember(act_AID, act_avail_pre')
        act_avail = [act_avail; act_AID];
    end
end

% Display actions
fprintf('    Available actions: ');
if toggle_txt_labels
    fprintf('\n');
end
for act_ID = 1:length(act_avail)
    act_AID = act_avail(act_ID);
    if toggle_txt_labels
        act_label = labels_master{act_AID};
        fprintf('        %d) %s\n', act_ID, act_label);
    else
        fprintf('%d, ', act_AID);
    end
end
if toggle_txt_labels
    fprintf('        %d) %s\n', (act_ID + 1), 'Other');
else
    fprintf('%d\n', (act_AID + 1));
end

% Select action
act_sel = input('        Select input: ');
if not(act_sel)
    fprintf('Devitation from model\n');
    break
end
```

```
    if toggle_txt_labels
        act_AID_sel = act_avail(act_sel);
    else
        act_AID_sel = act_sel;
    end

    % Append data
    act_path_pre = [act_path_pre; act_AID_sel];
    curr_path_pre = [curr_path_pre; act_AID_sel];
    act_path_pos = [act_path_pos; act_AID_sel];
    curr_path_pos = [curr_path_pos; act_AID_sel];

    % Update paths available
    for dir = [-1, 1]
        switch dir
            case -1
                paths_avail = paths_avail_pos;
                curr_path = curr_path_pos;
            case 1
                paths_avail = paths_avail_pre;
                curr_path = curr_path_pre;
        end

        % Check paths against current path
        paths_avail_store = paths_avail;
        paths_avail = {};
        for path_ID = 1:length(paths_avail_store)
            path_comp = paths_avail_store{path_ID}(1:length(curr_path));
            if all(isequal(path_comp, curr_path))
                paths_avail = [paths_avail; paths_avail_store{path_ID}];
            end
        end

        switch dir
            case -1
                paths_avail_pos = paths_avail;
                if (length(paths_avail) == 1) && (length(paths_avail{1}) ==
length(curr_path) + 1)
                    state_pos = paths_avail{1}(end);
                    act_path_pos = [act_path_pos; state_pos];
                else
                    seq_ID_pos = seq_ID_pos + 1;
                end
            case 1
                paths_avail_pre = paths_avail;
                if (length(paths_avail) == 1) && (length(paths_avail{1}) ==
length(curr_path) + 1)
                    state_pre = paths_avail{1}(end);
                    act_path_pre = [act_path_pre; state_pre];
                else
                    seq_ID_pre = seq_ID_pre + 1;
                end
        end
    end
end
```

```
if act_sel
    fprintf('Drink complete\n');
end
```

APPENDIX B. CHAPTER 3 STATE ID SCRIPTS

Setup

To extract states with genetic optimization, first compile each of the scripts in this appendix into separate files, named exactly as their heading.

ClassMicro.mat

To define optimized state descriptions, run `ClassMicro.mat`. This can take some time. Running this script will prompt a user input of "Run type" which determines which suffix of trace file will be used to produce state definitions. This should be done in parallel to updating `para_inc` to label the specific parameters which should be included as consideration for the state definition. Note that `para_fixed` will require the optimization to include the specific parameters listed.

To interpret results, `x_star` outputs the optimized configuration, `model_GMM` outputs the corresponding optimized model, `classes_GMM`, the corresponding labels for each microstate, and `spred` includes the relative inclusion rates of each gene during optimization. For diagnostics, stats includes information on generational behaviors, `GA550.mat` includes the specific information.

```
%% classMicro_04
% This script classifies microstates using the targeted eps method
% Changes from 03
%     - Added channel filtering
%     - Updated x_star to output channel IDs

clc
clear

%% Program settings
run_type = input('Input run type: ', 's');
para_inc = [3; 4; 11; [15:21]'];

% Load data
data_master = load(strcat('trainingStates_trimmed_', run_type, '.mat'));
data_master = data_master.trainingStates_trimmed(:, para_inc);
[micro_cnt, chan_cnt] = size(data_master);
```

```
% Prep settings
class_cnt_bounds = [3, 18];
para_fixed = [11; [15:18]']; % Must be included in para_inc
[~, para_fixed] = ismember(para_fixed, para_inc);
chan_var_cnt = chan_cnt - length(para_fixed);
para_bounds = zeros(chan_var_cnt, 2);
para_bounds(:, 2) = 1;
bounds = [class_cnt_bounds; para_bounds];

bits = zeros((1 + chan_var_cnt), 1);
bits(1) = 4;
bits(2:end) = 1;

% Optimization settings
gen_para = sum(bits);
pop_size = 4 * gen_para;
cross_freq = 0.5;
mut_freq = (gen_para + 1) / (2 * pop_size * gen_para);
options = goptions([]);
options(11) = pop_size;
options(12) = cross_freq;
options(13) = mut_freq;
options(14) = 300;


%% Normalize data
% Prep storage
data_norm = data_master;
stdDev_vals = zeros(chan_cnt, 1);
mean_vals = stdDev_vals;

% Iterate through channels
for chan_ID = 1:chan_cnt
    stdDev_vals(chan_ID) = std(data_master(:, chan_ID));
    mean_vals(chan_ID) = mean(data_master(:, chan_ID));
end

% Iterate through dataset
for chan_ID = 1:chan_cnt
    stdDev_val = stdDev_vals(chan_ID);
    mean_val = mean_vals(chan_ID);
    if stdDev_val == 0
        data_norm(:, chan_ID) = mean_val;
    else
        for micro_ID = 1:micro_cnt
            data_norm(micro_ID, chan_ID) = (data_norm(micro_ID, chan_ID) -
mean_val) / stdDev_val;
        end
    end
end


%% Classify data
fprintf('Initializing GMM...\n');
% Construct modeling function
```

```
opt_func = @(opt_vars) calc_obj(opt_vars, data_norm, para_fixed);

% Optimize
[x_star, f_star, stats, nfit, fgen, lgen, lfit, spred] = GA550(opt_func, [],
options, bounds(:, 1)', bounds(:, 2)', bits');
class_cnt = round(x_star(1), 0);
para_bool = x_star(2:end);

para_include = para_fixed;
para_bool_ID = 1;
for para_ID = 1:(chan_cnt - 2)
    if not(ismember(para_ID, para_include))
        if para_bool(para_bool_ID)
            para_include = [para_include; para_ID];
        end
        para_bool_ID = para_bool_ID + 1;
    end
end
para_include = sort(para_include);
data_train = data_norm(:, para_include);
x_star = [class_cnt; para_inc(para_include)];

model_GMM = fitgmdist(data_train, class_cnt, 'RegularizationValue', 0.0001,
'Options', statset('Display', 'off', 'MaxIter', 500));
classes_GMM = cluster(model_GMM, data_train);

% Save data
save(strcat('lgen_', run_type, '.mat'), 'lgen');
save(strcat('stats_', run_type, '.mat'), 'stats');
save(strcat('x_star_', run_type, '.mat'), 'x_star');
save(strcat('model_GMM_', run_type, '.mat'), 'model_GMM');
save(strcat('classes_GMM_', run_type, '.mat'), 'classes_GMM');
save(strcat('spred_', run_type, '.mat'), 'spred');
fprintf('Done!\n');
```

GA550.mat

This is a modified variant of Dr. Crossley's genetic optimization function as presented in Purdue University's "Multidisciplinary Design Optimization in Aerospace Engineering" course (Crossley, 2020). It outputs the variables defined in `ClassMicro.mat`. The modifications to the original script allow the system to store and output statistics on the relative presence of different genes with each generation, allowing for some further verification of success and study of how different genes may be related to success.

```
function [xopt,fopt,stats,nfit,fgen,lgen,lfit] = GA550(fun, ...
    x0,options,vlb,vub,bits,P1,P2,P3,P4,P5,P6,P7P,P8,P9,P10)
%GA550 minimizes a fitness function using a simple genetic algorithm.
%
%   X=GA550('FUN',X0,OPTIONS,VLB,VUB) uses a simple
```

```
%       genetic algorithm to find a minimum of the fitness function
%       FUN.  FUN can be a user-defined M-file: FUN.M, or it can be a
%   string containing the function itself.  The user may define all
%       or part of an initial population X0. Any undefined individuals
%   will be randomly generated between the lower and upper bounds
%   (VLB and VUB).  If X0 is an empty matrix, the entire initial
%   population will be randomly generated.  Use OPTIONS to specify
%   flags, tolerances, and input parameters.  Type HELP GOPTIONS
%       for more information and default values.
%
%   X=GA550('FUN',X0,OPTIONS,VLB,VUB,BITS) allows the user to
%   define the number of BITS used to code non-binary parameters
%   as binary strings.  Note: length(BITS) must equal length(VLB)
%   and length(VUB).  If BITS is not specified, as in the previous
%   call, the algorithm assumes that the fitness function is
%   operating on a binary population.
%
%   X=GA550('FUN',X0,OPTIONS,VLB,VUB,BITS,P1,P2,...) allows up
%   to ten arguments, P1,P2,... to be passed directly to FUN.
%   F=FUN(X,P1,P2,...). If P1,P2,... are not defined, F=FUN(X).
%
%   [X,FOPT,STATS,NFIT,FGEN,LGEN,LFIT]=GA550(<ARGS>)
%          X       - design variables of best ever individual
%          FOPT    - fitness value of best ever individual
%          STATS   - [min mean max stopping_criterion] fitness values
%                    for each generation
%          NFIT  - number of fitness function evalations
%          FGEN    - first generation population
%          LGEN    - last generation population
%          LFIT    - last generation fitness
%
%       The algorithm implemented here is based on the book: Genetic
%       Algorithms in Search, Optimization, and Machine Learning,
%       David E. Goldberg, Addison-Wiley Publishing Company, Inc.,
%       1989.
%
%   Originally created on 1/10/93 by Andrew Potvin, Mathworks, Inc.
%   Modified on 2/3/96 by Joel Grasmeyer.
%   Modified on 11/12/02 by Bill Crossley.
%   Modified on 7/20/04 by Bill Crossley.

% Make best_feas global for stopping criteria (4/13/96)
global best_feas
global gen
global fit_hist
% Load input arguments and check for errors
if nargin<4,
    error('No population bounds given.')
elseif (size(vlb,1)~=1) | (size(vub,1)~=1),
    % Remark: this will change if algorithm accomodates matrix variables
    error('VLB and VUB must be row vectors')
elseif (size(vlb,2)~=size(vub,2)),
    error('VLB and VUB must have the same number of columns.')
elseif (size(vub,2)~=size(x0,2)) & (size(x0,1)>0),
    error('X0 must all have the same number of columns as VLB and VUB.')
elseif any(vlb>vub),
    error('Some lower bounds greater than upper bounds')
```

```
else
    x0_row = size(x0,1);
    for i=1:x0_row,
        if any(x0(x0_row,:)<vlb) | any(x0(x0_row,:)>vub),
            error('Some initial population not within bounds.')
        end % if initial pop not within bounds
    end % for initial pop
end % if nargin<4

if nargin<6,
    bits = [];
elseif (size(bits,1)~=1) | (size(bits,2)~=size(vlb,2)),
    % Remark: this will change if algorithm accomodates matrix variables
    error('BITS must have one row and length(VLB) columns')
elseif any(bits~=round(bits)) | any(bits<1),
    error('BITS must be a vector of integers >0')
end % if nargin<6

% Form string to call for function evaluation
if ~( any(fun<48) | any(fun>122) | any((fun>90) & (fun<97)) | ...
        any((fun>57) & (fun<65)) ),
    % Only alphanumeric characters implies that 'fun' is a separate m-file
    evalstr = [fun '(x'];
    for i=1:nargin-6,
        evalstr = [evalstr,',P',int2str(i)];
    end
else
    % Non-alphanumeric characters implies that the function is contained
    % within the single quotes
    evalstr = ['(',fun];
end

% Determine all options
% Remark: add another options index for type of termination criterion
if size(options,1)>1,
    error('OPTIONS must be a row vector')
else
    % Use default options for those that were not passed in
    options = goptions(options);
end
PRINTING = options(1);
BSA = options(2);
fit_tol = options(3);
nsame = options(4)-1;
elite = options(5);

% Since operators are tournament selection and uniform crossover and
% default coding is Gray / binary, set crossover rate to 0.50 and use
% population size and mutation rate based on Williams, E. A., and Crossley,
% W. A., "Empirically-derived population size and mutation rate guidelines
% for a genetic algorithm with uniform crossover," Soft Computing in
% Engineering Design and Manufacturing, 1998.  If user has entered values
% for these options, then user input values are used.
if options(11) == 0,
    pop_size = sum(bits) * 4;
else
```

```
    pop_size = options(11);
end
if options(12) == 0,
    Pc = 0.5;
else
    Pc = options(12);
end
if options(13) == 0,
    Pm = (sum(bits) + 1) / (2 * pop_size * sum(bits));
else
    Pm = options(13);
end
max_gen = options(14);
% Ensure valid options: e.q. Pc,Pm,pop_size,max_gen>0, Pc,Pm<1
if any([Pc Pm pop_size max_gen]<0) | any([Pc Pm]>1),
    error('Some Pc,Pm,pop_size,max_gen<0 or Pc,Pm>1')
end

% Encode fitness (cost) function if necessary
ENCODED = any(any(([vlb; vub; x0]~=0) & ([vlb; vub; x0]~=1))) |  ....
    ~isempty(bits);
if ENCODED,
    [fgen,lchrom] = encode(x0,vlb,vub,bits);
else
    fgen = x0;
    lchrom = size(vlb,2);
end

% Display warning if initial population size is odd
if rem(pop_size,2)==1,
    disp('Warning: Population size should be even.  Adding 1 to population.')
    pop_size = pop_size +1;
end

% Form random initial population if not enough supplied by user
if size(fgen,1)<pop_size,
    fgen = [fgen; (rand(pop_size-size(fgen,1),lchrom)<0.5)];
end
xopt = vlb;
nfit = 0;
new_gen = fgen;
isame = 0;
bitlocavg = mean(fgen,1);  % initial bit string affinity
BSA_pop = 2 * mean(abs(bitlocavg - 0.5));
fopt = Inf;
stats = [];

% Header display
if PRINTING>=1,
    if ENCODED,
        disp('Variable coding as binary chromosomes successful.')
        disp('')
        fgen = decode(fgen,vlb,vub,bits);
    end
    disp('                    Fitness statistics')
    if nsame > 0
```

```
        disp('Generation Minimum      Mean          Maximum       isame')
    elseif BSA > 0
        disp('Generation Minimum      Mean          Maximum       BSA')
    else
        disp('Generation Minimum      Mean          Maximum       not used')
    end
end

% Set up main loop
STOP_FLAG = 0;
for generation = 1:max_gen+1,
    old_gen = new_gen;

    % Decode binary strings if necessary
    if ENCODED,
        x_pop = decode(old_gen,vlb,vub,bits);
    else
        x_pop = old_gen;
    end

    % Get fitness of each string in population
    for i = 1:pop_size,
        x = x_pop(i,:);
        fitness(i) = eval([evalstr,')']);
        nfit = nfit + 1;
    end

    % Store minimum fitness value from previous generation (except for
    % initial generation)
    if generation > 1,
        min_fit_prev = min_fit;
        min_gen_prev = min_gen;
        min_x_prev = min_x;
    end

    % identify worst (maximum) fitness individual in current generation
    [max_fit,max_index] = max(fitness);

    % impose elitism - currently only one individual; this replaces worst
    % individual of current generation with best of previous generation
    if (generation > 1 & elite > 0),
        old_gen(max_index,:) = min_gen_prev;
        x_pop(max_index,:) = min_x_prev;
        fitness(max_index) = min_fit_prev;
    end

    % identify best (minimum) fitness individual in current generation and
    % store bit string and x values
    [min_fit,min_index] = min(fitness);
    min_gen = old_gen(min_index,:);
    min_x = x_pop(min_index,:);

    % Store best fitness and x values
    if min_fit < fopt,
        fopt = min_fit;
```

```
        xopt = min_x;
    end

    % Compute values for isame or BSA_pop stopping criteria
    if nsame > 0
        if generation > 1
            if min_fit_prev == min_fit
                isame = isame + 1;
            else
                isame = 0;
            end
        end
    elseif BSA > 0
        bitlocavg = mean(old_gen,1);
        BSA_pop = 2 * mean(abs(bitlocavg - 0.5));
    end


    % Calculate generation statistics
    if nsame > 0
        stats = [stats; generation-1,min(fitness),mean(fitness), ...
            max(fitness), isame];
    elseif BSA > 0
        stats = [stats; generation-1,min(fitness),mean(fitness), ...
            max(fitness), BSA_pop];
    else
        stats = [stats; generation-1,min(fitness),mean(fitness), ...
            max(fitness), 0];
    end

    % Display if necessary
    if PRINTING>=1,
        disp([sprintf('%5.0f %12.6g %12.6g %12.6g %12.6g',
stats(generation,1), ...
                stats(generation,2),stats(generation,3),
stats(generation,4),...
                stats(generation,5))]);
    end

    % Check for termination
    % The default termination criterion is bit string affinity.  Also
    % available are fitness tolerance across five generations and number of
    % consecutive generations with same best fitness.  These can be used
    % concurrently.
    if fit_tol>0,    % if fit_tol > 0, then fitness tolerance criterion used
        if generation>5,
            % Check for normalized difference in fitness minimums
            if stats(generation,1) ~= 0,
                if abs(stats(generation-5,1)-stats(generation,1))/ ...
                        stats(generation,1) < fit_tol
                    if PRINTING >= 1
                        fprintf('\n')
                        disp('GA converged based on difference in fitness
minimums.')
                    end
                    lfit = fitness;
```

```
                    if ENCODED,
                        lgen = x_pop;
                    else
                        lgen = old_gen;
                    end
                    return
                end
            else
                if abs(stats(generation-5,1)-stats(generation,1)) < fit_tol
                    if PRINTING >= 1
                        fprintf('\n')
                        disp('GA converged based on difference in fitness 
minimums.')
                    end
                    lfit = fitness;
                    if ENCODED,
                        lgen = x_pop;
                    else
                        lgen = old_gen;
                    end
                    return
                end
            end
        end
    elseif nsame > 0,     % consecutive minimum fitness value criterion
            if isame == nsame
                if PRINTING >= 1
                    fprintf('\n')
                    disp('GA stopped based on consecutive minimum fitness 
values.')
                end
                lfit = fitness;
                if ENCODED,
                    lgen = x_pop;
                else
                    lgen = old_gen;
                end
                return
            end
    elseif BSA > 0,   % bit string affinity criterion
        if BSA_pop >= BSA,
            if PRINTING >=1
                fprintf('\n')
                disp('GA stopped based on bit string affinity value.')
            end
            lfit = fitness;
            if ENCODED,
                lgen = x_pop;
            else
                lgen = old_gen;
            end
            return
        end
    end

    % Tournament selection
    new_gen = tourney(old_gen,fitness);
```

```
    % Crossover
    new_gen = uniformx(new_gen,Pc);

    % Mutation
    new_gen = mutate(new_gen,Pm);

    % Always save last generation.  This allows user to cancel and
    % restart with x0 = lgen
    if ENCODED,
        lgen = x_pop;
    else
        lgen = old_gen;
    end


end % for max_gen

% Maximum number of generations reached without termination
lfit = fitness;
if PRINTING>=1,
    fprintf('\n')
    disp('Maximum number of generations reached without termination')
    disp('criterion met.  Either increase maximum generations')
    disp('or ease termination criterion.')
end


% end genetic

function [gen,lchrom,coarse,nround] = encode(x,vlb,vub,bits)
%ENCODE Converts from variable to binary representation.
%   [GEN,LCHROM,COARSE,nround] = ENCODE(X,VLB,VUB,BITS)
%       encodes non-binary variables of X to binary.  The variables
%       in the i'th column of X will be encoded by BITS(i) bits.  VLB
%       and VUB are the lower and upper bounds on X.  GEN is the binary
%       representation of these X.  LCHROM=SUM(BITS) is the length of
%       the binary chromosome.  COARSE(i) is the coarseness of the
%       i'th variable as determined by the variable ranges and
%       BITS(i).  ROUND contains the absolute indices of the
%       X which where rounded due to finite BIT length.
%
%   Copyright (c) 1993 by the MathWorks, Inc.
%   Andrew Potvin 1-10-93.

% Remark: what about handling case where length(bits)~=length(vlb)?


lchrom = sum(bits);
coarse = (vub-vlb)./((2.^bits)-1);
[x_row,x_col] = size(x);

gen = [];
if ~isempty(x),
```

```
   temp = (x-ones(x_row,1)*vlb)./ ...
          (ones(x_row,1)*coarse);
   b10 = round(temp);
   % Since temp and b10 should contain integers 1e-4 is close enough
   nround = find(b10-temp>1e-4);
   gen = b10to2(b10,bits);
end

% end encode


function [x,coarse] = decode(gen,vlb,vub,bits)
%DECODE Converts from binary Gray code to variable representation.
%   [X,COARSE] = DECODE(GEN,VLB,VUB,BITS) converts the binary
%       population GEN to variable representation.  Each individual
%       of GEN should have SUM(BITS).  Each individual binary string
%       encodes LENGTH(VLB)=LENGTH(VUB)=LENGTH(BITS) variables.
%       COARSE is the coarseness of the binary mapping and is also
%       of length LENGTH(VUB).
%
%  this *.m file created by combining "decode.m" from the MathWorks, Inc.
%  originally created by Andrew Potvin in 1993, with "GDECODE.FOR" written
%  by William A. Crossley in 1996.
%
%   William A. Crossley, Assoc. Prof. School of Aero. & Astro.
%  Purdue University, 2001
%
%  gen is an array [population size , string length], each row is one
individual's chromosome
%  vlb is a row vector [number of parameters], each entry is the lower bound
for a variable
%  vub is a row vector [number of parameters], each entry is the upper bound
for a variable
%  bits is a row vector [number of parameters], each entry is number of bits
used for a variable
%

no_para = length(bits); % extract number of parameters using number of rows
in bits vector
npop = size(gen,1);     % extract population size using number of rows in gen
array
x = zeros(npop, no_para);  % sets up x as an array [population size, number
of parameters]
coarse = zeros(1,no_para); % sets up coarse as a row vector [number of
parameters]

for J = 1:no_para,  % extract the resolution of the parameters
    coarse(J) = (vub(J)-vlb(J))/(2^bits(J)-1);  % resolution of parameter J
end

for K = 1:npop,     % outer loop through each individual (there may be a more
efficient way to operate on the
                  % gen array) BC 10/10/01
    sbit = 1;       % initialize starting bit location for a parameter
    ebit = 0;       % initialize ending bit location
```

```
   for J = 1:no_para,   % loop through each parameter in the problem
    ebit = bits(J) + ebit;  % pick the end bit for parameter J
        accum = 0.0;                 % initialize the running sum for
parameter J
      ADD = 1;                       % add / subtract flag for Gray code; add
if(ADD), subtract otherwise
      for I = sbit:ebit,             % loop through each bit in parameter J
         pbit = I + 1 - sbit;        % pbit determines value to be added or
subtracted for Gray code
         if (gen(K,I))                   % if "1" is at current location
            if (ADD)                         % add if appropriate
               accum = accum + (2.0^(bits(J)-pbit+1) - 1.0);
               ADD = 0;                  % next time subtract
            else
               accum = accum - (2.0^(bits(J)-pbit+1) - 1.0);
               ADD = 1;                  % next time add
            end
         end
      end                                % end of I loop through each bit
      x(K,J) = accum * coarse(J) + vlb(J);          % decoded parameter J for
individual K
      sbit = ebit + 1;                                      % next parameter
starting bit location
   end                      % end of J loop through each parameter
end                     % end of K loop through each individual

%end gdecode


function [new_gen,mutated] = mutate(old_gen,Pm)
%MUTATE Changes a gene of the OLD_GEN with probability Pm.
%   [NEW_GEN,MUTATED] = MUTATE(OLD_GEN,Pm) performs random
%       mutation on the population OLD_POP.  Each gene of each
%       individual of the population can mutate independently
%       with probability Pm.  Genes are assumed possess boolean
%       alleles.  MUTATED contains the indices of the mutated genes.
%
%   Copyright (c) 1993 by the MathWorks, Inc.
%   Andrew Potvin 1-10-93.

mutated = find(rand(size(old_gen))<Pm);
new_gen = old_gen;
new_gen(mutated) = 1-old_gen(mutated);

% end mutate


function [new_gen,nselected] = tourney(old_gen,fitness)
%TOURNEY Creates NEW_GEN from OLD_GEN, based on tournament selection.
%    [NEW_GEN,NSELECTED] = TOURNEY(OLD_GEN,FITNESS) selects
%        individuals from OLD_GEN by competing consecutive individuals
%    after random shuffling.  NEW_GEN will have the same number of
%    individuals as OLD_GEN.
%        NSELECTED contains the number of copies of each individual
```

```
%    that survived.  This vector corresponds to the original order
%    of OLD_GEN.
%
%    Created on 1/21/96 by Joel Grasmeyer

% Initialize nselected vector and indices of old_gen
new_gen = [];
nselected = zeros(size(old_gen,1),1);
i_old_gen = 1:size(old_gen,1);

% Perform two "tournaments" to generate size(old_gen,1) new individuals
for j = 1:2,

  % Shuffle the old generation and the corresponding fitness values
  [old_gen,i_shuffled] = shuffle(old_gen);
  fitness = fitness(i_shuffled);
  i_old_gen = i_old_gen(i_shuffled);

  % Keep the best of each pair of individuals
  index = 1:2:(size(old_gen,1)-1);
  [min_fit,i_min] = min([fitness(index);fitness(index+1)]);
  selected = i_min + [0:2:size(old_gen,1)-2];
  new_gen = [new_gen; old_gen(selected,:)];

  % Increment counters in nselected for each individual that survived
  temp = zeros(size(old_gen,1),1);
  temp(i_old_gen(selected)) = ones(length(selected),1);
  nselected = nselected + temp;

end

% end tourney


function [new_gen,index] = shuffle(old_gen)
%SHUFFLE Randomly reorders OLD_GEN into NEW_GEN.
%    [NEW_GEN,INDEX] = MATE(OLD_GEN) performs random reordering
%        on the indices of OLD_GEN to create NEW_GEN.
%    INDEX is a vector containing the shuffled row indices of OLD_GEN.
%
%    Created on 1/21/96 by Joel Grasmeyer

[junk,index] = sort(rand(size(old_gen,1),1));
new_gen = old_gen(index,:);

% end shuffle


function [new_gen,sites] = uniformx(old_gen,Pc)
%UNIFORMX Creates a NEW_GEN from OLD_GEN using uniform crossover.
%     [NEW_GEN,SITES] = UNIFORMX(OLD_GEN,Pc) performs uniform crossover
%         on consecutive pairs of OLD_GEN with probability Pc.
%     SITES shows which bits experienced crossover.  1 indicates
%     allele exchange, 0 indicates no allele exchange.  SITES has
```

```
%     size(old_gen,1)/2 rows.
%
%     Created 1/20/96 by Joel Grasmeyer

new_gen = old_gen;
sites = rand(size(old_gen,1)/2,size(old_gen,2)) < Pc;
for i = 1:size(sites,1),
  new_gen([2*i-1 2*i],find(sites(i,:))) = old_gen([2*i
2*i-1],find(sites(i,:)));
end

% end uniformx
```

goptions.mat

This function is an unmodified subfunction related to `GA550.mat` (Crossley, 2020). This function defines the constraints of operation for genetic optimization.

```
function OPTIONS=goptions(parain);
%GOPTIONS Default parameters used by the genetic algorithm GENETIC.
%
% Note that since the original version was written, the Matlab Optimization
% Toolbox now uses "optimset" to set generic optimization parameters, so
% this format is somewhat outdated.
%
% The genetic algorithm parameters used for this implementation are:
%
%   OPTIONS(1)-Display flag:  0 = none, 1 = some, 2 = all  (Default: 1).
%   OPTIONS(2)-Termination bit string affinity value (Default: 0.90; set to
zero to turn off)
%   OPTIONS(3)-Termination tolerance for fitness (Default: 0; not normally
used).
%   OPTIONS(4)-Termination number of consecutive generations with same best
%   fitness (Default: 0; to use, set number, be sure OPTIONS(2) and
OPTIONS(3) = 0).
%   OPTIONS(5)-Number of elite individuals (Default: 0; no elitism).
%   OPTIONS(6)-
%   OPTIONS(7)-
%   OPTIONS(8)-
%   OPTIONS(9)-
%   OPTIONS(10)-
% Genetic Algorithm-specific inputs
%   OPTIONS(11)-Population size (fixed)
%   OPTIONS(12)-Probability of crossover
%   OPTIONS(13)-Probability of mutation
%   OPTIONS(14)-Maximum number of generations, always used as safeguard
%   (Default: 200).
%
%
% Explanation of defaults:
%   The default algorithm displays statistical information for each
%   generation by setting OPTIONS(1) = 1.  Plots are produced when
```

```
%   OPTIONS(1) = 2.
%   The OPTIONS(2) flag is originally set for termination criterion based
%   on X; here it is used if bit string affinity is selected.
%   The default fitness function termination tolerance,
%   OPTIONS(3), is set to 0, which terminates the optimization when 5
%   consecutive best generation fitness values are the same.  A positive
%   value terminates the optimization when the normalized difference
%   between the previous fitness and current generation fitness is less
%   than the tolerance.  See the code for details.
%  OPTIONS(4) has a default value of 5; this means if the best fitness
%   value in the population is unchanged for 5 consecutive generations
%  the GA is terminated.
%       The default algorithm uses a fixed population size, OPTIONS(11),
%       and no generational overlap.  The default population size is 30.
%   Three genetic operations:  selection, crossover, and mutation are
%   used for procreation.
%   The default selection scheme is tournament selection.
%       Crossover occurs with probability Pc=OPTIONS(12).  The default
%   crossover scheme is uniform crossover with Pc = 0.5.
%   Each allele of the offspring mutates independently with probability
%       Pm=OPTIONS(13); here the default is 0.01.
%       The default number of maximum generations, OPTIONS(14) is 200.
%
%   Last modified by Bill Crossley 07/20/04

% The following lines have been commented out by Steven Lamberson.
% They have been changed to what is seen below them. (06/30/06).
% This change was made in order to fix the following problems:
%   1 - code changed user supplied options(1)=0 to options(1)=1
%   2 - code changed user supplied options(2)=0 to options(2)=0.9

%if nargin<1; parain = []; end
%sizep=length(parain);
%OPTIONS=zeros(1,14);
%OPTIONS(1:sizep)=parain(1:sizep);
%default_options=[1,0.9,0,0,0,0,0,0,0,0,0,0,0,200];
%OPTIONS=OPTIONS+(OPTIONS==0).*default_options

if nargin<1; parain = []; end
sizep=length(parain);
OPTIONS=zeros(1,14)-1;
OPTIONS(1:sizep)=parain(1:sizep);
default_options=[1,0.9,0,0,0,0,0,0,0,0,0,0,0,200];
for i = 1:length(OPTIONS)
    if OPTIONS(i) == -1
        OPTIONS(i) = default_options(i);
    end
end
```

calc_obj.mat

This function measures the fitness of each classification tuning as generated by `GA550.mat` using the Calinski-Harabasz criterion. Note that fitness is optimized at low objective values, so the output is inverted. Additionally, some residual random sampling test cases are included, but are nonfunctional and do not affect operation.

```
%% calc_obj
% This script measures the efficacy of flight clustering for a given data set


function obj = calc_obj(opt_vars, data_set, para_include)
    perc_thresh = 0.75;

    %% Extract data
    % Extract base level data
    [micro_cnt, chan_cnt] = size(data_set);
    class_cnt = round(opt_vars(1), 0);
    para_include_bool = opt_vars(2:end);
    para_bool_ID = 1;
    for para_ID = 1:chan_cnt
        if not(ismember(para_ID, para_include))
            if para_include_bool(para_bool_ID)
                para_include = [para_include; para_ID];
            end
            para_bool_ID = para_bool_ID + 1;
        end
    end
    para_include = sort(para_include);

    % Refine data
    chan_cnt = length(para_include);
    data_set = data_set(:, para_include);


    %% Compute objective
%     [model_GMM, rand_perc] = flight_clust(data_set, class_cnt, [0.25, 0.75,
3]);
    rand_perc = 1;
    model_GMM = fitgmdist(data_set, class_cnt, 'RegularizationValue', 0.0001,
'Options', statset('Display', 'off', 'MaxIter', 500));
    classes_GMM = cluster(model_GMM, data_set);
    obs_cnts = zeros(class_cnt, 1);
    obs_store = cell(class_cnt, 1);
    centroids = zeros(class_cnt, chan_cnt);
    for class_ID = 1:class_cnt
        obs_IDs = find(classes_GMM == class_ID);
        obs_list = data_set(obs_IDs, :);
        obs_cnts(class_ID) = size(obs_list, 1);
        obs_store{class_ID} = obs_list;
        for chan_ID = 1:chan_cnt
```

```
            centroids(class_ID, chan_ID) = mean(obs_list(:, chan_ID));
        end
    end

    centroid_abs = mean(data_set, 1);
    SSb = 0;
    SSw = 0;
    for class_ID = 1:class_cnt
        obs_cnt = obs_cnts(class_ID);
        obs_list = obs_store{class_ID};
        centroid_class = centroids(class_ID);
        SSb = SSb + (obs_cnt * (norm(centroid_class - centroid_abs)^2));

        for obs_ID = 1:obs_cnt
            SSw = SSw + (norm(obs_list(obs_ID, :) - centroid_class)^2);
        end
    end

    % Compute Criterion
    VRC = (SSb / SSw) * (micro_cnt - class_cnt) / (class_cnt - 1);
    obj = -1 * VRC / 1000;

    % Adjust bounds
    obj = obj + 50 * max([0, ((rand_perc / perc_thresh) - 1)]);
end
```

APPENDIX C. CHAPTER 3 PATH ID SCRIPTS

Setup

To determine paths using a specific model, pretrained model, place the following script in a file titled `path_finder.mat`. Files will use one label to indicate sampling frequency, `freq`, which is either "04" or "40" respectively. Files will use another label, `type`, to indicate the specific constraints placed on the state definition. Use type "c" to indicate compass, type "t" to indicate target heading, and type "p" to indicate positionless.

In the same directory as this file, place the parameter trace data stored as a variable in the file `trainingStates_trimmed_`**`freqtype`**`.mat`. Place the corresponding control trace data in a file `trainingInputs_trimmed_`**`freqtype`**`.mat`. Replace "freqtype" in both file names with the corresponding strings for `freq` and `type` used. A third file, storing a matrix containing the indices of the end of each use instance in the trace for the sampling frequency, should be placed in the same directory and titled `end_IDs_`**`freq`**`.mat`. Replace "freq" with the corresponding `freq` string.

State models should be placed in directories inside the current, following the file path "Classifications/Optimized run **freqtype**/`model_GMM.mat`", replacing "freqtype" as before. Path models should be place in separates directories, labeled "Path Models/**freqtype**". Mean parameter and control values for each state should be stored in the corresponding directory in the file `mean_vals.mat`. These values should be stored in a matrix, with each row corresponding to a state, and each column to a metric. Similarly, standard deviations should be stored in the file `stdDev_vals.mat` in the same directory.

For microstate prediction, metric prediction models should be stored in internal directories to the previous freqtype path model directory. Each state should have a corresponding directory, title S#, where # corresponds to the state ID. Each model needs to be named **`metric`**`_model.mat`, with metric replaced by the corresponding metric ID code.

Direct prediction models can then be stored in this same directory using the filename `direct_`**`samp`**`_model.mat`, where samp is replaced by the sampling method, either "rand" or "stan".

path_finder.mat

This script compares a path prediction method to the ground truth using the path prediction models from requisite folders and labels from the provided trace. Note that `freq` and `type` define the trace used, as well as which parameters are relevant for path definition. These parameters are based on state definitions. The variable `predictor` will determine which type of model will be used for prediction, and `samp` will determine the sampling method used for training the original model. `Samp` only affects which model is loaded.

In general, this script is highly specialized to the exact case tested in this thesis and could be more efficiently adapted if written from scratch. Path models were individually generated from MATLAB's regression learner and classification learner apps and have not been automated in a script. I highly recommend storing normalization means and standard deviations in files and retrieving them in every scrip over recalculating and normalizing. This also goes for state and path labels, as relabeling all of the trace every run can lead to inconsistencies if scripts change accidentally.

```
%% path_finder03
% This script uses indivudal channel models to predict path
% Changes from 02
%     - Enabled

clc
clear

%% Program settings
% Primary settings
freq = '40';
type = 'p';
state_init = 3;
predictor = 'direct'; % micro / direct
samp = 'Stan'; % Rand / Stan

% Secondary settings
class_labels = {'Low-speed'; 'High-speed'; 'Hazard'};
state_order = load(strcat('state_order_', freq, '.mat'));
state_order = state_order.state_order;
class_cnt = size(state_order, 2);

if strcmp(freq, '04')
    if strcmp(type, 'c')
        chan_GMM = [1; 2; 9; 11; 12; [14:18]'; 23];
        label_names = {'SINCH'; 'COSCH'; 'ZPL'; 'YVI'; 'ZVT'; 'ZVL'; 'FVP';
'VVP'; 'HVP'; 'T'; 'B'};
        state_order = state_order(1, :);
```

```
        plot_title = strcat('4 Hz, compass, ', {' '}, predictor, ' prediction
confusion');
    elseif strcmp(type, 't')
        chan_GMM = [1; 5; 8; 11; [13:18]'];
        label_names = {'SINTH'; 'XPT'; 'XPL'; 'YVI'; 'XVL'; 'ZVL'; 'FVP';
'VVP'; 'HVP'; 'T'};
        state_order = state_order(2, :);
        plot_title = strcat('4 Hz, target, ', {' '}, predictor, ' prediction
confusion');
    elseif strcmp(type, 'p')
        chan_GMM = [3; 4; 11; [15:19]'];
        label_names = {'PA'; 'BA'; 'YVI'; 'FVP'; 'VVP'; 'HVP'; 'T'; 'CSE'};
        state_order = state_order(3, :);
        plot_title = strcat('4 Hz, positionless, ', {' '}, predictor, '
prediction confusion');
    elseif strcmp(type, 'm')
        chan_GMM = [1; [3:7]'; 10; 11; [13:21]'];
        label_names = {'SIN(TH)'; 'SIN(CH)'; 'COS(CH)'; 'PA'; 'BA'; 'XPT';
'XPL'; 'ZPL'; 'YVI'; 'ZVT'; 'XVL'; 'ZVL'; 'FVP'; 'VVP'; 'HVP'; 'T'; 'CSE'};
        state_order = 1:class_cnt;
        plot_title = strcat('4 Hz, merged, ', {' '}, predictor, ' prediction
confusion');
    end
elseif strcmp(freq, '40')
    if strcmp(type, 'c')
        chan_GMM = [1; 2; 7; 11; 12; [14:18]'];
        label_names = {'SINCH'; 'COSCH'; 'XPT'; 'YVI'; 'ZVT'; 'ZVL'; 'FVP';
'VVP'; 'HVP'; 'T'};
    elseif strcmp(type, 't')
        chan_GMM = [1; 2; 8; 11; [14:18]'];
        label_names = {'SINTH'; 'COSTH'; 'XPL'; 'YVI'; 'ZVL'; 'FVP'; 'VVP';
'HVP'; 'T'};
    elseif strcmp(type, 'p')
        chan_GMM = [3; 11; [15:19]'];
        label_names = {'PA'; 'YVI'; 'FVP'; 'VVP'; 'HVP'; 'T'; 'CSE'};
        state_order = state_order(3, :);
        plot_title = strcat('40 Hz, positionless, ', {' '}, predictor, '
prediction confusion');
    end
end

% Load main data
state_master = load(strcat('trainingStates_trimmed_', freq, type, '.mat'));
state_master = state_master.trainingStates_trimmed(:, chan_GMM);
[micro_cnt, state_chan_cnt] = size(state_master);

input_master = load(strcat('trainingInputs_trimmed_', freq, '.mat'));
input_master = input_master.trainingInputs_trimmed;
[~, input_chan_cnt] = size(input_master);
chan_cnt = state_chan_cnt + input_chan_cnt;

end_IDs = load(strcat('end_IDs_', freq, '.mat'));
end_IDs = end_IDs.end_IDs;

% Load class model data
directory = strcat('Classifications/Optimized run', {' '}, freq, type);
```

```
directory = directory{1};
state_model = load(strcat(directory, '/model_GMM_', freq, type, '.mat'));
state_model = state_model.model_GMM;
path_cnt = class_cnt^2;

% Set class correction
class_convert = state_order;
for class_ID = 1:class_cnt
    class_convert(class_ID) = find(state_order == class_ID);
end

% Load trend data
directory = strcat('Path Models/', freq, type);
normalization_means = load(strcat(directory, '/mean_vals.mat'));
normalization_means = normalization_means.mean_vals;
normalization_stdDevs = load(strcat(directory, '/stdDev_vals.mat'));
normalization_stdDevs = normalization_stdDevs.stdDev_vals;
directory = strcat('Path Models/', freq, type, '/');

mean_vals = normalization_means;
stdDev_vals = normalization_stdDevs;
for class_ID = 1:class_cnt
    store_ID = class_convert(class_ID);
    normalization_means(store_ID, :) = mean_vals(class_ID, :);
    normalization_stdDevs(store_ID, :) = stdDev_vals(class_ID, :);
end

if isequal(predictor, 'micro')
    % Load channel model data
    path_model_set = cell(class_cnt, state_chan_cnt);
    for class_ID = 1:class_cnt
        store_ID = class_convert(class_ID);
        for model_ID = 1:state_chan_cnt
            model_name = strcat(label_names{model_ID}, '_model');
            model = load(strcat(directory, 'S', num2str(class_ID), '/',
model_name, '.mat'));
            path_model_set{store_ID, model_ID} = model.(model_name);
        end
    end
end


%% Establish ground truth
% Prep storage
state_norm = state_master;
state_stdDev_vals = zeros(state_chan_cnt, 1);
state_mean_vals = state_stdDev_vals;

% Iterate through channels
for chan_ID = 1:state_chan_cnt
    state_stdDev_vals(chan_ID) = std(state_master(:, chan_ID));
    state_mean_vals(chan_ID) = mean(state_master(:, chan_ID));
end

% Iterate through dataset
```

```
for chan_ID = 1:state_chan_cnt
    stdDev_val = state_stdDev_vals(chan_ID);
    mean_val = state_mean_vals(chan_ID);
    if stdDev_val == 0
        state_norm(:, chan_ID) = mean_val;
    else
        for micro_ID = 1:micro_cnt
            state_norm(micro_ID, chan_ID) = (state_norm(micro_ID, chan_ID) -
mean_val) / stdDev_val;
        end
    end
end

% Prep storage
input_norm = input_master;
input_stdDev_vals = zeros(input_chan_cnt, 1);
input_mean_vals = state_stdDev_vals;

% Iterate through channels
for chan_ID = 1:input_chan_cnt
    input_stdDev_vals(chan_ID) = std(input_master(:, chan_ID));
    input_mean_vals(chan_ID) = mean(input_master(:, chan_ID));
end

% Iterate through dataset
for chan_ID = 1:input_chan_cnt
    stdDev_val = input_stdDev_vals(chan_ID);
    mean_val = input_mean_vals(chan_ID);
    if stdDev_val == 0
        input_norm(:, chan_ID) = mean_val;
    else
        for micro_ID = 1:micro_cnt
            input_norm(micro_ID, chan_ID) = (input_norm(micro_ID, chan_ID) -
mean_val) / stdDev_val;
        end
    end
end

% Cluster data
if type == 'm'
    class_list = state_model.predictFcn(state_norm);
else
    [class_list, ~, probs] = cluster(state_model, state_norm);
    for micro_ID = 1:micro_cnt
        class_list(micro_ID) = class_convert(class_list(micro_ID));
    end
end

% Path data
path_true_list = zeros(micro_cnt, 1);
for micro_ID = 1:(micro_cnt - 1)
    if not(ismember(micro_ID, end_IDs))
        path_true_list(micro_ID) = ((class_list(micro_ID) - 1) * class_cnt) +
class_list(micro_ID + 1);
    end
end
```

```
% Trim all data to remove instance ends
micro_cnt = (micro_cnt - length(end_IDs));
state_master(end_IDs, :) = [];
input_master(end_IDs, :) = [];
class_list(end_IDs) = [];
path_true_list(end_IDs) = [];

% Determine relevant micro IDs
keep_list = find(class_list == state_init);


%% Model paths
if isequal(predictor, 'micro')
    % Prep storage
    state_pred_list = state_master;

    % Predict behavior
    microdata_master = [state_master, input_master];
    for class_ID = 1:class_cnt
        % Isolate microstates in class
        micro_ID_list = find(class_list == class_ID);

        % Normalize in class
        microdata_norm = microdata_master;
        for chan_ID = 1:chan_cnt
            stdDev_val = normalization_stdDevs(class_ID, chan_ID);
            if stdDev_val
                microdata_norm(micro_ID_list, chan_ID) =
(microdata_norm(micro_ID_list, chan_ID) - normalization_means(class_ID,
chan_ID)) / stdDev_val;
            else
                microdata_norm(micro_ID_list, chan_ID) =
microdata_norm(micro_ID_list, chan_ID) - normalization_means(class_ID,
chan_ID);
            end
        end

        % Predict for class
        for chan_ID = 1:state_chan_cnt
            channel_model = path_model_set{class_ID, chan_ID};
            state_pred_list(micro_ID_list, chan_ID) =
(channel_model.predictFcn(microdata_norm(micro_ID_list, :)) *
normalization_stdDevs(class_ID, chan_ID)) + normalization_means(class_ID,
chan_ID);
        end
    end
    micro_ID_list = find(class_list == state_init);
    % micro_cnt = length(micro_ID_list);
    % ML_comm = state_pred_list(micro_ID_list, :);
    % ML_resp = path_true_list(micro_ID_list, :);
    %
    % stdDev_vals = zeros(state_chan_cnt, 1);
    % mean_vals = state_stdDev_vals;
    %
```

```
    % % Iterate through channels
    % for chan_ID = 1:state_chan_cnt
    %     stdDev_vals(chan_ID) = std(ML_comm(:, chan_ID));
    %     mean_vals(chan_ID) = mean(ML_comm(:, chan_ID));
    % end
    %
    % % Iterate through dataset
    % for chan_ID = 1:state_chan_cnt
    %     stdDev_val = stdDev_vals(chan_ID);
    %     mean_val = mean_vals(chan_ID);
    %     if stdDev_val == 0
    %         ML_comm(:, chan_ID) = mean_val;
    %     else
    %         for micro_ID = 1:micro_cnt
    %             ML_comm(micro_ID, chan_ID) = (ML_comm(micro_ID, chan_ID) -
mean_val) / stdDev_val;
    %         end
    %     end
    % end

    % Convert to class
    for chan_ID = 1:state_chan_cnt
        stdDev_val = state_stdDev_vals(chan_ID);
        mean_val = state_mean_vals(chan_ID);
        if stdDev_val == 0
            state_pred_list(:, chan_ID) = mean_val;
        else
            for micro_ID = 1:micro_cnt
                state_pred_list(micro_ID, chan_ID) =
(state_pred_list(micro_ID, chan_ID) - mean_val) / stdDev_val;
            end
        end
    end
    class_pred_list = cluster(state_model, state_pred_list);
    for micro_ID = 1:micro_cnt
        class_pred_list(micro_ID) = class_convert(class_pred_list(micro_ID));
    end

    path_pred_list = ((class_list - 1) * class_cnt) + class_pred_list;
    path_pred_list = path_pred_list(keep_list);

elseif isequal(predictor, 'direct')
    % Load path model
    path_model = load(strcat(directory, 'S', num2str(state_init), '/direct',
samp, '_model.mat'));
    path_model = path_model.direct_model;

    % Normalize metrics in state of interest
    microdata_norm = [state_master, input_master];
    microdata_norm = microdata_norm(keep_list, :);
    for chan_ID = 1:chan_cnt
        stdDev_val = stdDev_vals(state_init, chan_ID);
        if stdDev_val
            microdata_norm(:, chan_ID) = (microdata_norm(:, chan_ID) -
mean_vals(state_init, chan_ID)) / stdDev_val;
        else
```

```
            microdata_norm(:, chan_ID) = microdata_norm(:, chan_ID) -
mean_vals(state_init, chan_ID);
        end
    end
    if type == 'c'
        microdata_norm(:, state_chan_cnt) = [];
    end

    % Predict path IDs
    path_pred_list = path_model.predictFcn(microdata_norm);
end

% Plot results
path_true_list = path_true_list(keep_list, :);
cm = confusionchart(confusionmat(path_true_list, path_pred_list),
class_labels);
cm.RowSummary = 'row-normalized';
cm.ColumnSummary = 'column-normalized';
cm.Title = plot_title;
sortClasses(cm, class_labels);
```

REFERENCES

Aalto, A., Husberg, N., & Varpaaniemi, K. (2003). Automatic Formal Model Generation and Analysis of SDL. *International SDL Forum* (pp. 285-299). Stuttgart, Germany: Springer, Berlin, Heidelberg.

Boonchoo, T., Ao, X., Liu, Y., Zhao, W., Zhuang, F., & Qing, H. (2019). Grid-based DBSCAN: Indexing and inference. *Elsevier ScienceDirect Journals*, 271-284.

Crossley, W. (2020, April). Simple genetic optimizer. West Lafayette, Indiana, USA.

IBM. (2017, October 3). *System trace*. Retrieved from IBM: https://www.ibm.com/docs/en/zos/2.1.0?topic=aids-system-trace

IBM. (2020, September 21). *Common unsupervised learning approaches*. Retrieved from IBM: https://www.ibm.com/cloud/learn/unsupervised-learning

Jung, D., Ramanan, N., Amjadi, M., Karingula, S. R., Taylor, J., & Coelho Jr, C. N. (2021, June 11). *Time Series Anomaloy Detection with label-free Model Selction.* Retrieved from arxiv.org: https://arxiv.org/abs/2106.07473

Köppen, M. (2000). The curse of dimensionality. *Conference on Soft Computing in Industrial Applications.* Virtual Event.

Mathworks. (2021, October 3). *DBSCAN*. Retrieved from Mathworks: https://www.mathworks.com/help/stats/dbscan-clustering.html

MathWorks. (2021, October 13). *evalclusters*. Retrieved from MathWorks: https://www.mathworks.com/help/stats/evalclusters.html#shared-criterion

McGonagle, J., Pilling, G., & Dobre, A. (2021, October 3). *Gaussian Mixture Model*. Retrieved from Brilliant: https://brilliant.org/wiki/gaussian-mixture-model/

NASA Langley. (2016, April 10). *Introducing Formal Methods*. Retrieved from NASA.gov: https://shemesh.larc.nasa.gov/fm/fm-what.html

National Transportation Safety Board. (2013). *Descent Below Visual Glidepath and Impact With Seawall*. San Francisco: NTSB.

Puranik, T. G., & Mavris, D. N. (2018). Anomaly Detection in General Aviation Operations Using Energy Metrcis and Flight-Data Records. *Journal of Aerospace Information Systems*, 22-36.

Selvaraj, Y., Farooqui, A., Panahandeh, G., & Fabian, M. (2020). Automatically learning formal models: an industrial case from autonomous driving development. *International Conference on Model Driven Engineering Languages and Systems* (pp. 1-10). Virtual Event: Association for Computing Machinery.

Solar-Lezama, A., Rabbah, R. M., Bodik, R., & Ebicioglu, K. (2005). Programming by sketching for bit-streaming programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*

Yamakawa, S. (2021, October 3). *YSFlight*. Retrieved from YSFlight: https://ysflight.org/