Developing an Abstraction Layer for the Visualization of HSMM-Based Predictive Decision Support

by

Hank Hsin Han Huang

S.B., Management Science M.I.T., 2008

S.B., Computer Science and Engineering M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 2009

[September 2009]

Author_____

Hank Huang
Department of Electrical Engineering and Computer Science
August 2009

Certified by_____

Mary L. Cummings
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by_____

Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Developing an Abstraction Layer for the Visualization of HSMM-Based Predictive Decision Support

By
Hank Hsin Han Huang

Submitted to the
Department of Electrical Engineering and Computer Science

August 2009

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Hidden semi-Markov models (HSMMs) have been previously proposed as real-time operator behavior prediction models that could be used by a supervisor to detect future anomalous behaviors. Because of the disconnect between HSMM prediction results and the data format anticipated by the decision support visualization (DSV) display designer, an abstraction layer was developed to transform HSMM results into data in the anticipated format. In order to transform the raw HSMM results, a model accuracy scoring metric was created to assess HSMM prediction data and produce model performance trend data with a graphical depiction of variance and lower bounds. A prediction-generating (PG) algorithm was devised to utilize the model accuracy scoring metric and the HSMM library functions to generate multi-step ahead predictions up to 3 minutes into the future.

In order to implement a responsive decision support system monitoring up to 10 operators simultaneously, original design requirements constrained maximum latency at 500ms, as suggested by previous research. However, the PG algorithm yielded significant system latency, and thus, computational enhancements were put in place to speed up the algorithm. Moreover, trade-offs were made between the length of input to the PG algorithm and the length of predictions generated. Both parameters were linearly proportional to latency. Other research has shown that a maximum latency of less than 200ms may be more desirable, and thus, the total number of operators supported would be down to 4 per the given system.

The resulting proof-of-concept system operates in real-time, providing a team supervisor the most up-to-date supervision of up to 4 UV operators simultaneously. A pilot study was conducted to test the usability of the system where no major issues were found, and the study proved that the system operates as per the design requirements.

Thesis Supervisor: Mary L. Cummings
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgements

This thesis would not have come to completion without the help of many people along the way, and I would like to express my deepest gratitude.

First, I would like to thank my thesis supervisor, Prof. Missy Cummings. I would like to thank her for believing in me, taking me under her wing as a student, and trusting me to work on this project. Without her guidance along the way, I do not know how I could find my way to the finish line. Thank you also for being so accommodating to my schedule. I am wishing for any future opportunity to work with you again.

Thank you also to Yves Boussemart. Thank you for all of your advice and the countless discussion sessions we had. Your mathematical expertise on HSMMs and your mentoring allowed me to meet most of the deadlines. I thank you for giving me that extra push towards the finish line every time I needed it.

I would like to thank Jonathan Las Fargeas and Ryan Castonia for working with me on this research project. The usability pilot testing would not have been possible without you. I would also like to thank the seven other HAL UROPs for their time and effort to work on the pilot study, even though you all had to work on other research projects at the same time, Meghan, Erick, Peter, Nick, Paul, Tony, and Vicki.

To Eric Huang, my loving baby brother: thanks for keeping my motivation up. Living with you this summer and watching you work as hard as I did definitely helped me to stay focused and motivated. I wish we will have more opportunities to spend more time together.

Last but not least, I would like to thank my parents for all of your love and support for the past 23 years. Without you, I would never be where I am today. Thank you both for believing in me throughout every single round of education, and thank you both for always keeping an eye out for me to make sure I was always on the right track and moving forward. I am the most fortunate to be your son.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AMS	Automated Monitoring System
API	Application Programming Interface
ARMA	Autoregressive Moving Average
CA	Client Application
CTA	Cognitive Task Analysis
DBN	Dynamic Bayesian Network
DSV	Decision Support Visualization
ED	Expected Duration
EMAS	Expected Model Accuracy Score
ETA	Expected Time of Arrival
EWW	Event Window Width
GUI	Graphical User Interface
HALE	High Altitude Long Endurance Air Unmanned Vehicle
HMM	Hidden Markov Model
HSMM	Hidden Semi-Markov Model
ISR	Intelligence, Surveillance, and Reconnaissance
LB	Lower Bound
MALE	Medium Altitude Long Endurance Air Unmanned Vehicle
MAS	Model Accuracy Score
PC	Prediction Confidence
PG	Prediction-Generating
RESCHU	Research Environment for Supervisory Control of Heterogeneous Unmanned Vehicles
SA	Server Application
SD	Standard Deviation
UUV	Underwater Unmanned Vehicle
UV	Unmanned Vehicle

1 Introduction

In future unmanned vehicle (UV) systems, increasingly robust automation will eventually allow one operator to manage multiple vehicles in a supervisory control setting (Cummings, Bruni, Mercier, & Mitchell, 2007). As computers automate low-level sensory and piloting tasks, human operators can concentrate on higher-level decision making like navigation and payload management. In addition, future UV systems will most likely consist of a team of operators lead by a supervisor, much like present-day air traffic control settings as seen in Figure 1.1.

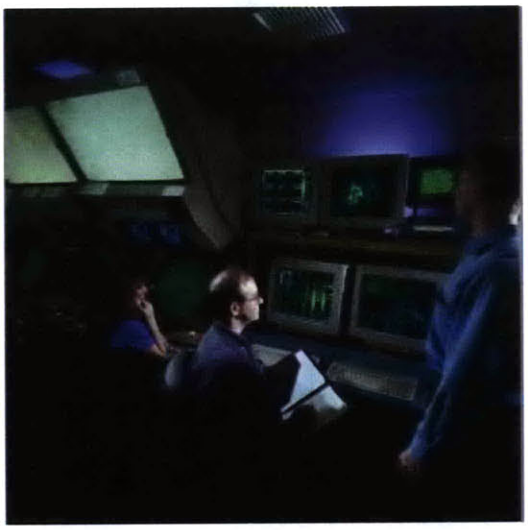


Figure 1.1: A present-day air traffic control center (Hart, 1995)

As operators' responsibilities shift towards the higher-level tasks of controlling multiple systems such as group schedule management instead of single vehicle navigation, a mistake made by an operator could become more costly. Thus, the supervisor's ability to monitor operator actions and limit the number of operator mistakes will be crucial factors to the success of future UV operations in complex, time-critical settings. The supervisor of such settings is likely to experience high workload while attempting to monitor both operators and multiple unmanned vehicles, all while maintaining an appropriate level of situation awareness of the ongoing operation.

Automation can be used to assist the supervisor in assessing operator performance, in that it can provide automated detection, and perhaps prediction of possible problematic behaviors. While automation is useful in monitoring specific current states of complex systems such as nuclear power plants, one novel and recent application of automation for supervisor support is through the use of learning algorithms trained to detect off-nominal operator behavior given a set of known normal behaviors. Figure 1.2 illustrates a conceptual future UV system with supervisor decision support.

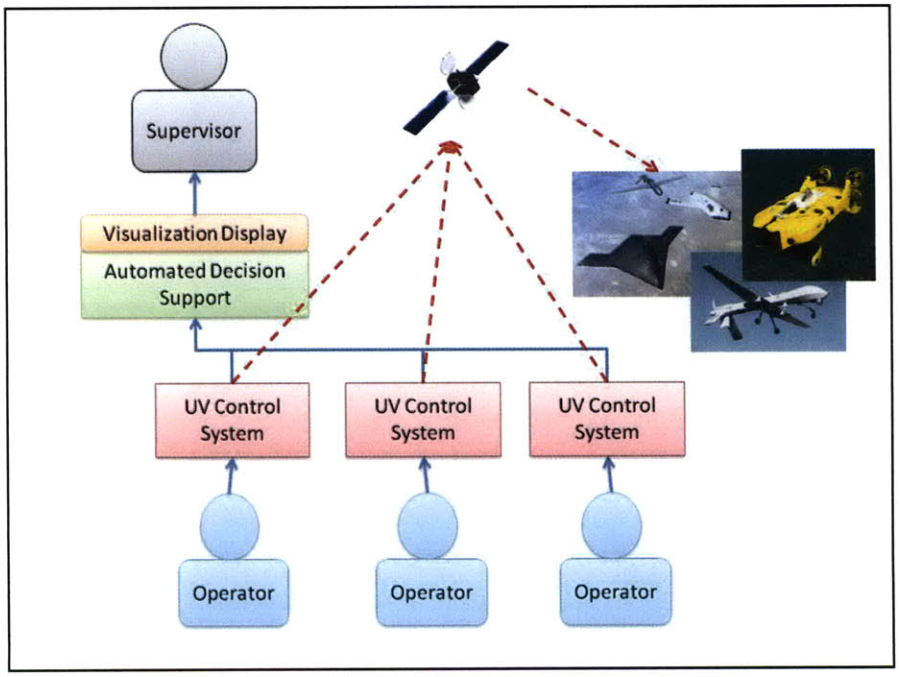


Figure 1.2: A conceptual future UV system with supervisor decision support

In this conceptual future UV system, each operator interacts with a UV control system to supervise and direct the vehicles. These UV control systems are typically connected through satellite signals. In these futuristic systems, the UV control systems could be monitored by an Automated Monitoring System (AMS). The AMS samples interactions between the operators and the UV control systems to assess operator behaviors. In addition, the AMS generates predictions of future operator behaviors. Assessment and predictions results are then presented via a visualization display, through which the supervisor is able to monitor the operators (Figure 1.2).

The ability to detect deviations from normal operator behavior presupposes knowing what "normal" behavior entails. Even for operators trained to follow specific operating procedures, correct behavior is often difficult to define, especially in command and control settings where uncertainty is significant and unanticipated events are typical. Recent research has shown that pattern recognition and prediction techniques such as hidden semi-Markov models (HSMMs) can be used to model the temporal behavior of operators of multiple heterogeneous UV systems (Boussemart & Cummings, 2008). More specifically, HSMMs provide the ability to generate predictions on future operator inputs through statistical inference based on past operator inputs and resultant learned patterns of operator-UV interaction. While such models can detect different patterns of behavior, they cannot assess goodness of operator interactions, which is why a supervisor is needed to assess if an anomalous pattern of behavior is good or bad.

A sequence of generated predictions from an HSMM represents the expected (i.e. most likely) operator behaviors with respect to a training set. Thus, deviations from the expected behavior can be detected as possible anomalous conditions and a supervisor of a team of UV operators can be alerted to these possible deviations. A design of a user-friendly graphical trend display has been proposed as a decision support tool with embedded alerts to aid the supervisor (Castonia, 2009). The proposed graphical user interface (GUI), which depicts operator behavior on a continuous spectrum, is designed to help supervisors visualize behavioral deviations of operators as predicted by the HSMMs, as well as historical trends of model accuracy.

There exists a disconnect between discrete data points generated by a HSMM and the continuous flow of data stream that the GUI demands. In order to generate predictions, the HSMM is queried upon the arrival of an operator-UV interaction event. These predictions are represented as probability density functions and thus can be difficult to comprehend for non-technical personnel, including team supervisors (Tversky & Kahneman, 1974). Because these predictions are generated only when a new event arrives, the data points are discrete. However, the GUI, which relies on a trend interface, is designed to display continuous curves to represent model accuracy. Hence, an

abstraction layer that maps the discrete datasets from the HSMM to the continuous data stream required by the GUI plays a vital role in linking the HSMM to the GUI from which decisions are made.

This thesis describes the abstraction layer that links the underlying HSMMs with the proposed graphical display. The product serves as a proof-of-concept for a real-time HSMM-based decision support tool for a supervisor of multiple operators controlling multiple UVs. The system also serves as a prototype reference for future development of decision support tools for supervisors of UV system operators that embed learning algorithms.

1.1 Problem Statement

Linking a user-friendly display to HSMMs is a challenging task because HSMMs are based on sophisticated mathematical formalisms. This work devises a robust abstraction layer to shield non-technical users from the mathematics of HSMMs, yet enable them easy access to predictive data in order to make time-critical decisions. The abstraction layer translates raw HSMM results into comprehendible expressions in real time for graphical trend display visualizations.

1.2 Research Objectives

This thesis project creates a server application (SA) that resides between the decision support visualization (DSV) (i.e., the GUI) and a multi-operator, multi-UV control simulation. The SA houses the HSMMs, generates predictions based on libraries of learned patterns, and calculates deviations from these patterns. The SA also keeps track of model performance to alert the supervisor of the model's credibility. Most importantly, the thesis project devises an abstraction layer on top of the raw prediction results from the HSMMs so that the DSV can be properly populated in real time.

To accomplish the objectives mentioned above, the software architecture is designed to meet the following specifications: 1) because future UV systems will likely consist of a

team of operators, the SA must be able to handle real-time analyses of more than one stream of inputs from separate instances of the multi-UV control simulation concurrently, which represents inputs from multiple operators, 2) the system must be fault tolerant in the sense that anomalies in one UV simulation environment should not stall computations on the SA for other UV simulations (so a single point failure will not fail the entire system), and finally 3) the abstraction layer will enable the SA to provide understandable outputs to a GUI over a network.

This research effort is novel because it represents the first-known such attempt to connect HSMMs with real-time decision aiding tools. This is a difficult problem because raw results output from HSMMs are probabilistic in nature and linking them to visualizations which are understandable to operators with little technical background is a major hurdle in the realization of this effort.

1.3 Thesis Organization

This thesis is organized into the following chapters:

- Chapter 1, Introduction, describes the motivation and research questions of this thesis.
- Chapter 2, Background, introduces hidden Markov model (HMM) fundamentals, explains the differences and similarities between HMMs and hidden semi-Markov models (HSMMs), and outlines how predictions can be generated from HSMMs. This chapter also reviews previous work on time series analyses, decision making under uncertainty, and visualization of anomaly detection in order to gain insights on devising the abstraction layer for the HSMM.
- Chapter 3, Overview of System Architecture, highlights features of the multi-UV control simulation used in this effort and describes the proposed design of the graphical trend interface and the server application (SA). This chapter also explains the difficulty of transforming outputs from the HSMM to usable DSV inputs.

- Chapter 4, Server Application Design and Implementation, lists the design decisions and specifications of the SA that houses the HSMMs and the abstraction layer. It also details the communication protocols between the SA and the multi-UV control simulation, and between the SA and the DSV's trend display.
- Chapter 5, Display Rendering and Usability Results, describes the implementation details of the graphical trend display and the results from the pilot study for usability testing. Specifically, this chapter defines how each section of the display is populated in real time and how users react to the display after preliminary testing.
- Chapter 6, Software Testing Results and Discussion, explains the results of system testing and explores the boundary conditions of the system, including latencies of the system, number of predictions generated, and number of operators under supervision.
- Chapter 7, Conclusion, reviews the answers to the research questions, discusses the contributions of this thesis work, presents design recommendations, and suggests areas for future research.

2 Background

2.1 Hidden Markov Models (HMMs)

Hidden Markov models (HMMs) were first formally defined by Baum et al. (1966), and their application was popularized by Rabiner et al (1986). HMMs consist of stochastic Markov chains based around a set of hidden states whose value cannot be directly observed (Baum & Petrie, 1966). However, each state generates a set of observable symbols with probabilities according to an emission function. In a supervisory control setting, operator mental states are unobservable to the supervisor; however, operators' inputs to a given control system can be observed (Boussemart & Cummings, 2008). Thus, HMMs can be used to model operators' behaviors, as the hidden human mental states can be modeled by the hidden HMM states, and the operator-system interactions can be represented by the observable symbols.

An HMM, H, can be defined mathematically as a four-tuple $H=(S, V, A, B)$, where S is the set of hidden states, and V is the set of observable symbols. A represents the state transition matrix, where an entry in the matrix, a_{ij}, is the transition probability from state S_i to state S_j. Lastly, B represents the observable emission matrix, where b_{ij}, given the model is currently in state S_i, represents the emission probability for observable symbol V_j. Additionally, the sum of a_{ij} over j is equal to 1, and the sum of b_{ij} over j is also equal to 1 (Rabiner & Juang, 1986). Figure 2.1 illustrates an example of a three-state HMM.

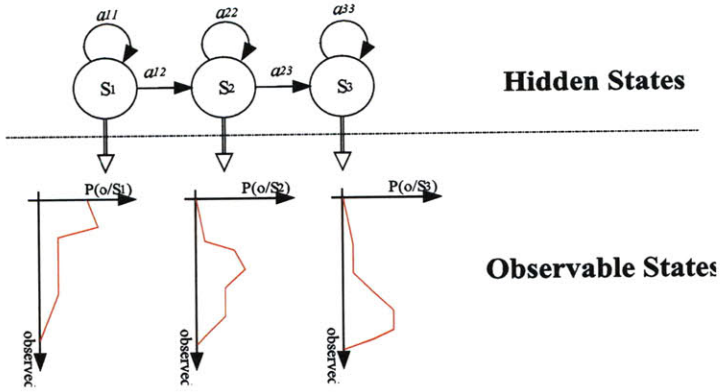


Figure 2.1: A three-state hidden Markov model (Boussemart, Fargeas, Cummings, & Roy, 2009)

An HMM can infer a sequence of state transitions from a sequence of observed symbols. For example, from a sequence of observed symbols, [V_1, V_3, V_2, V_2, V_2, V_4, V_5, V_5, V_1], from a three-state five-symbol HMM, the sequence of state transitions, [S_1, S_2, S_2, S_2, S_2, S_3, S_3, S_3, S_1], can be inferred if S_1 is known to emit V_1, S_2 is known to emit V_2 and V_3, and S_3 is known to emit V_4 and V_5. Moreover, it can be seen that all states transitioned back to themselves a number of times. This special transition is defined as self-transition. After the sequence of state transitions is obtained, the HMM can use the sequence of state transitions to predict future observable symbols. In the above example, S_1 is seen as the last transitioned state, and since S_1 is known to emit V_1 with high probability, the HMM can predict that the next observable symbol could be V_1 with high probability.

As will be discussed in more detail in the next section, a hidden semi-Markov model (HSMM) is a modification of the HMM, where self-transitions are no longer necessary due to each state having a duration property in addition to observables emission and state transition. Each state is given a probabilistic distribution of its duration. An HSMM is a specific sort of dynamic Bayesian network (Allanach, Tu, Singh, Willett, & Pattipati, 2004), and applications range from detecting terrorist networks to speech recognition (Rabiner & Juang, 1986). It was used in the context of this multiple operator, multi-UV supervisory control setting because of the critical temporal aspects, which are fundamental to event-based supervisory control systems.

2.2 Time Series Analyses

Single operator supervisory control interaction events are inherently sequential, so it is natural to use time series analysis techniques to model such data. Because this project makes use of Bayesian models in order to model operator behaviors and flag possible behavioral deviations as anomalies, given previously observed information in near real time, it is critical that the temporal component of state transitions be accounted for in the predictions.

To allow users of a decision support tool enough time to react to possible deviations of expected operator behavior, it is essential that the system generates predictions of more than one step ahead in time, i.e., supervisors need as much time as possible to intervene if operator behavior could lead to negative consequences. Thus, any useful decision support tool in time-critical supervisory control systems should be able to predict both that an anomalous behavior could occur, and also when.

There is a vast selection of time series analysis techniques, including but not limited to moving average models, autoregressive moving average models (ARMA), and dynamic Bayesian networks (DBN) (Sykacek & Roberts, 2002). While moving average models simply describe the smoothed average trend of the sequential data points, Bayesian network models mimic random processes and have the ability to predict future data points. For example, one study devised a Bayesian network-based algorithm to make financial market forecasts (Shiqing & Dawei, 2004).

Even though moving average models are widely used in making financial market predictions, they are too simplistic for modeling complex operator behavior. In contrast, HSMMs, subsets of DBNs, are specifically designed and tailored to handle complex but real-time sequential processes, i.e. time series data.

2.2.1 Time Series Analysis with HSMMs

HSMMs are similar to HMMs with the exception that each state has an associated probability density function of the state's duration. The state duration effectively eliminates the need for self-transition of the states. For a given state, instead of having self-transitions, a state can have its duration extended. Consequently, each state in an HSMM can emit a sequence of observations instead of emitting only a single observation, as in an HMM.

For example, a sequence of observed symbols of a three-state five-observable HSMM can also be $[V_1, V_3, V_2, V_2, V_2, V_4, V_5, V_5, V_1]$, the same as the HMM sequence of

observed symbols in the earlier section. However, the inferred sequence of state transitions is now [S_1, S_2, S_3, S_1] with no repeated state transitions. Instead of S_2 and S_3 experiencing self-transitions, the two states are defined to have longer durations.

An HSMM, H', can then be defined mathematically as a five-tuple $H'=(S, V, A, B, D)$, where S, V, A, and B are identical to those in an HMM, and D is the duration distribution matrix. An entry, d_{ij}, in the duration distribution matrix represents the probability of state, S_i, having duration of j units of time.

HSMMs can also be used to generate time-series predictions and in a computationally tractable way (Bulla & Bulla, 2006; Dong & He, 2007; Parlos, Rais, & Atiya, 2000). Most of the proposed methods generate predictions by using a dynamic programming technique called the forward-backward algorithm (Rabiner & Juang, 1986). Based on this algorithm, Guedon (2003) proposed the implementation of the Viterbi algorithm which efficiently uses back-pointers in order to track the most likely transition path in the state space. The Viterbi algorithm is therefore able to generate a series of predictions from an HSMM along with associated probabilities of each possible transition path.

Although there have been many studies on using HSMMs to analyze time series data in general (Bulla & Bulla, 2006; Hieronymus, McKelvie, & McInnes, 1992), there is virtually no previous research on the subject of using HSMMs to model human behavior. Moreover, since analysis and prediction results from an HSMM are represented as sequences of probabilistic values, which are difficult to comprehend by non-technical personnel (Tversky & Kahneman, 1974), they cannot be presented directly to the users. As a result, there is a need for the development of an abstraction layer that makes sense of these values for the users of an associated DSV.

2.3 Judgment under Uncertainty

Several experiments have shown that human perception of probabilistic values is often subjective and biased (Tversky & Kahneman, 1974). Moreover, people can have overly

optimistic or pessimistic view of a same probabilistic value, depending on the context in which it is presented (Tversky & Kahneman, 1981). For example, people tend to believe they are not at risk for some health conditions, often ignoring the high probability of getting a heart attack or diabetes, even in the presence of significant symptoms. This bias proves to be difficult to remove (Weinstein & Klein, 1995). These different biases will often have significant influence over a person's decision making process, so in time-pressured high-risk settings like those in command and control, it is critical that these biases be mitigated to greatest extent possible. Otherwise, a decision maker can inaccurately interpret a probabilistic value and make the wrong decision.

In particular, supervisors of UV operations are likely to lack the right mathematical training that allows them to interpret probabilistic information objectively. Displaying a table of probabilities will likely confuse the supervisor and may result in bad decision making. To avoid costly failures, any decision support tool should keep the amount of probabilistic information presented to the supervisor at a minimum, and presented in a manner that is intuitive which can be acted upon quickly. Hence, there is a need for an abstraction layer that is able to transform probabilistic information into equivalent certainty values or expected values.

2.3.1 Visualization of Anomaly Detections

In this effort, the supervisor of UV operators is notified when potentially abnormal behavior of an operator is detected. For an HSMM, given the training set of known patterns, there are two basic ways to detect an anomaly pattern: 1) show that the observed pattern is similar to a known adversary pattern and 2) show that the observed pattern is dissimilar to a known normal pattern (Singh, Tu, Donat, Pattipati, & Willet, 1996). However, it can be impractical at times to train the models to recognize an exhaustive list of adversary patterns because the number of them can grow indefinitely. Moreover, because the degrees of freedom are large in command and control settings, it is not practical (nor not possible) to identify every possible adversary state. As a result, only known normal patterns made available by trained HSMMs are assumed in this work.

Therefore, abnormal operator behaviors are detected when an observed behavioral pattern deviates from the norm (most expected behaviors).

The norm of operator behaviors is represented by all highly probable sequences of operator-system interactions with associated probabilities of occurrence. To generate a sequence of predictions, the HSMM infers the most likely future operator-system interactions from a sequence of past operator-system interactions. An example of a set of raw results from an HSMM is a matrix of probabilistic values, as shown in Figure 2.2.

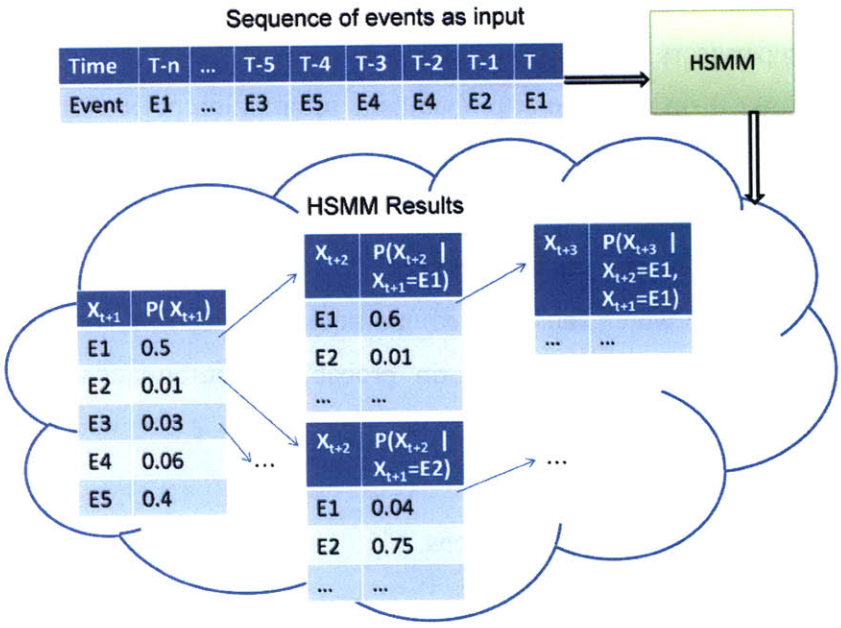


Figure 2.2: Examples of input to and output from an HSMM

In addition, deviation from the norm is also represented probabilistically. Without sufficient explanation, it will be challenging for non-technical personnel to understand these raw results. Indeed, it is often challenging for experts to interpret these results. One established effective way to represent anomalous states for time-series data is the use of trend graphs (Ware, 2000). Unfortunately, connecting HSMMs to trend graphs is difficult.

As can be seen in Figure 2.2, raw results from the HSMM contain many probabilistic values representing likelihoods of occurrences of different events at various time

intervals. Yet, it is unclear which values are to be plotted on the trend graphs. In addition, trend graphs assume continuous data and the probabilities in Figure 2.2, while interval in nature, are matched to discrete events. Thus, there is a need for an abstraction layer to transform discrete sets of probabilistic values into a continuous stream of date points.

2.4 Summary

This chapter established that an HSMM could be an appropriate time series analysis model for supervisory control anomaly detection. However, there is no previous literature on how to either use HSMMs to predict human behavior, or how to connect HSMMS to real-time decision support tools. This research proposes that for an HSMM-based predictive decision support system, there is a need for an abstraction layer that transforms raw HSMM results into meaningful values to be displayed on a graphical trend display. The next chapter will outline a proposed system architecture, and how the components of the system are connected.

3 Overview of System Architecture

As stated previously, the purpose of this thesis is to develop an abstraction layer between a hidden semi-Markov model (HSMM) and a graphical decision support tool that can aid a supervisor in real time to determine if personnel are behaving in an expected fashion. To this end, this chapter describes a representative UV environment that an HSMM models, as well as an actual decision support tool that provides alerts to a supervisor for off-nominal supervisee behavior. While the abstraction layer developed to link the HSMM with the decision support GUI is anchored in this specific case study, the methodology used is applicable to any other command and control application where a user interacts with a GUI in order to control automated remote systems, with a HSMM predicting possible future behaviors.

Previous research in single operator-multiple UV control led to the creation of the Research Environment for Supervisory Control of Heterogeneous Unmanned Vehicles (RESCHU) simulator, a software system that provides representative simulations of single operator control of multiple unmanned vehicles in an intelligence, surveillance, and reconnaissance (ISR) setting (Nehme, Crandall, & Cummings, 2008). The RESCHU environment serves as the representative single operator model upon which to build real-time HSMM predictions, and is described in more detail below. Since RESCHU was designed as a single operator model, the multiple representations in Figure 3.1 represent multiple operators, each independently controlling their own set of UVs.

Figure 3.1 illustrates the general system architecture required to link an HSMM via the abstraction layer to an actual system of humans controlling multiple UVs. This system is comprised of a number of operators (denoted by the word RESCHU in Figure 3.1 which represents the system they are controlling), a server application (SA) hosting the behavioral models (HSMMs), and a client application (CA), which is a graphical user interface (GUI).

For the purpose of this research, all the components in Figure 3.1 are designed to communicate with each other through a network connection. As a result, the SA is notified in real time when operators interact with RESCHU, and the SA then recognizes the type of interaction that occurred, and generates predictions to send as text messages to the CA over a network. Upon receiving the prediction message, the CA renders the graphics to be displayed for the supervisor.

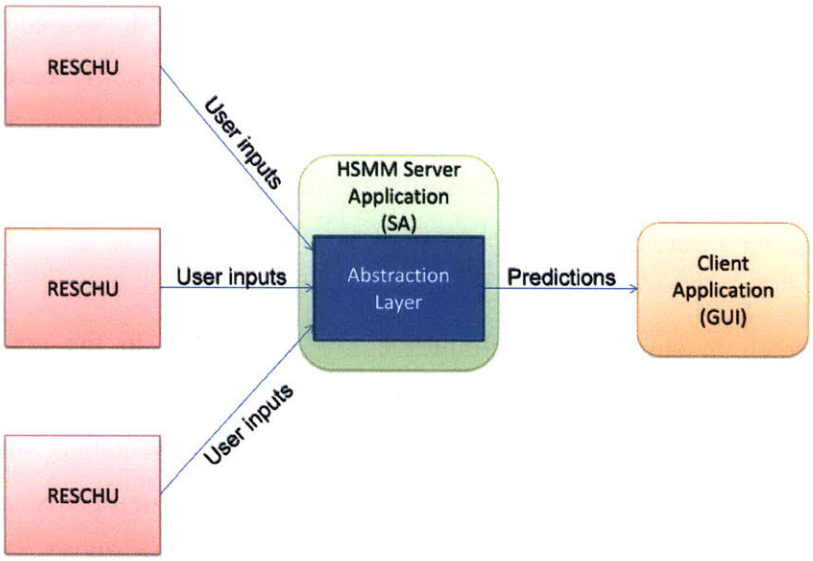


Figure 3.1: An overview of all the components the supervisor decision support system

3.1 RESCHU

In RESCHU, an operator controls multiple UVs to perform surveillance tasks with the ultimate goal of locating specific objects of interest in urban coastal and inland settings. There are three types of unmanned vehicles that the operator can control in RESCHU: Underwater UV (UUV), High Altitude Long Endurance Air UV (HALE), and Medium Altitude Long Endurance Air UV (MALE). Moreover, the RESCHU simulator can be loaded with different scenarios, where each scenario begins with five UVs of different combinations of the three types.

An operator's main objective is to conduct a visual search of as many target areas as possible to identify a possible target. Each operator is presented with five sub-interfaces in RESCHU: 1) Map Area, 2) Visual Task Panel, 3) Message Panel, 4) UV List, and 5) Time Line. A snapshot of the RESCHU application is shown in Figure 3.2.

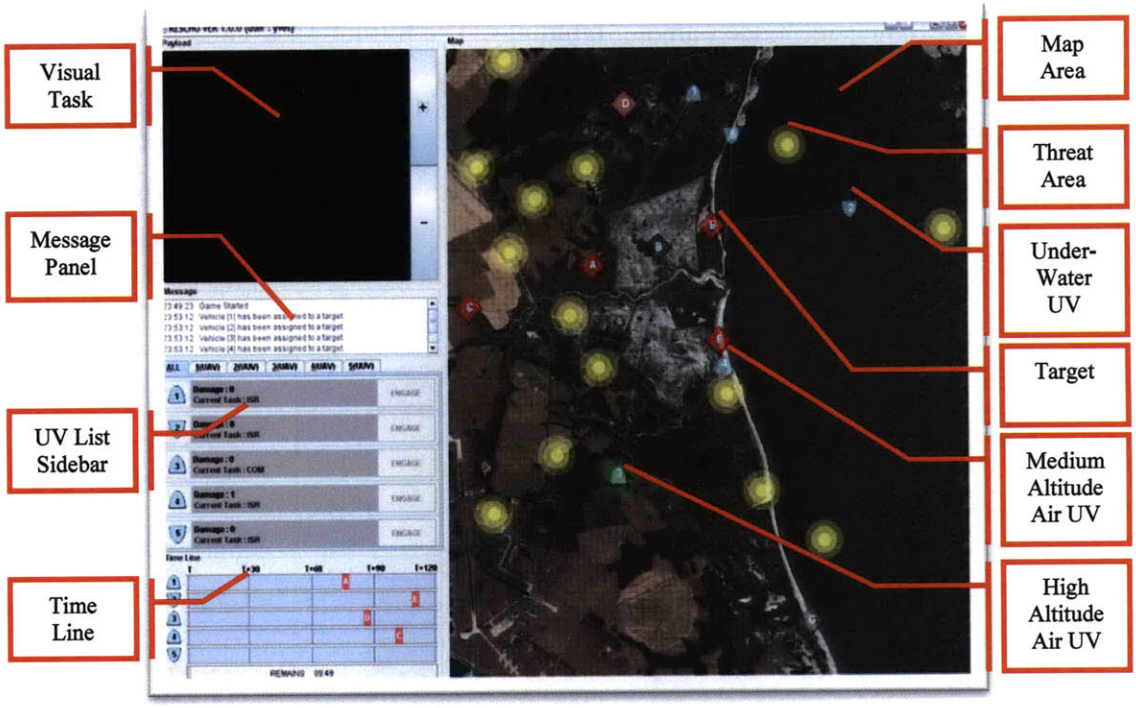


Figure 3.2: The RESCHU interface

3.1.1 Map Area

In the Map Area, the operator is free to interact with any of the five UVs. Threat areas on the map are marked by yellow circles. UVs need to avoid the threat areas, as they can be damaged while maneuvering through these areas. Operators can steer the UVs away from these areas by setting way points for each individual UV. The UVs are pre-assigned by the operator to various destinations or targets (i.e., goals) marked by red or gray diamonds. Upon a UV reaching its assigned target, the RESCHU system either urges the operator to reassign the UV to a different target or prompts the operator to engage in a visual task, which is described in the next section.

3.1.2 Visual Task Panel and Message Panel

The operator begins performing visual tasks in the Visual Task Panel when prompted by the system, which coincides with the arrival of a UV in a target area. A visual task involves the operator examining a photo to identify a specific object in the photo, such as searching for a fighter jet hidden among buildings. The Visual Task Panel provides the operator with tools to pan or zoom on the image. The RESCHU system provides the operator with the description of the object through displaying a message in the Message Panel. The Message Panel also signals the operator by displaying a short message when a UV reaches its target, as shown in Figure 3.3.

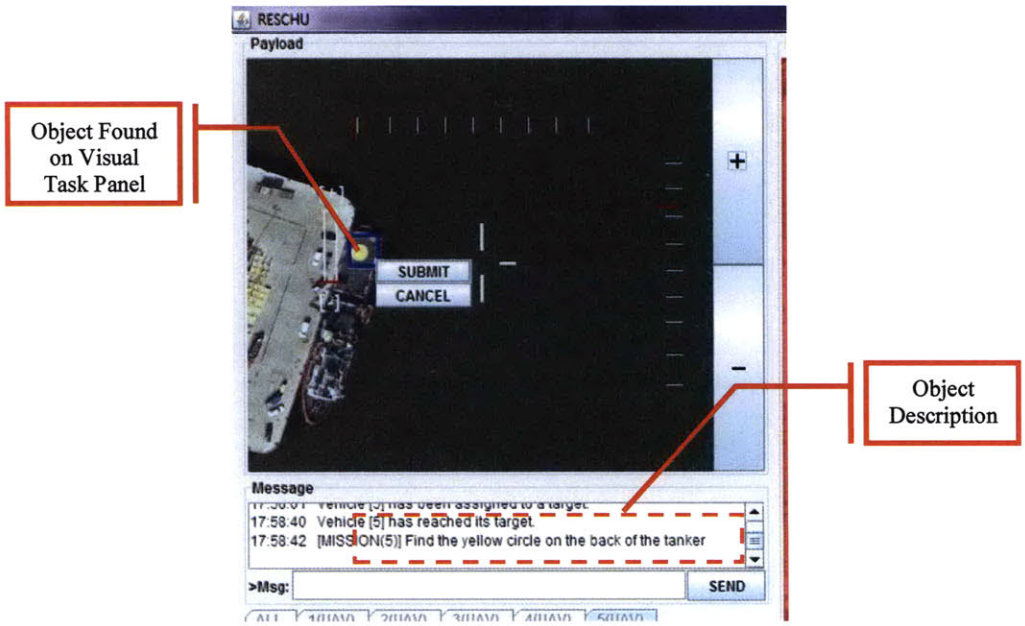


Figure 3.3: The Visual Task Panel

3.1.3 UV List

The UV List sidebar provides the operator with a quick overview of the status of the five UVs and serves as a shortcut for the operator to select a UV or engage a UV in visual task when appropriate. The operator can also monitor UVs' damage levels through this sidebar.

3.1.4 Time Line

The Time Line in Figure 3.2 presents a graphical view of the estimated remaining time to arrival of the UVs to their respective targets. This window also contains information on the total time remaining of the current RESCHU scenario.

3.1.5 Event Parsing and Communication

An operator's interaction with the RESCHU is encoded into various user input events by a grammatical parser (Boussemart & Cummings, 2008). The HSMM was trained to recognize patterns of these event encodings, and thus, the HSMM generates event predictions in the form of these event encodings as well. Each event is identified by a unique number (Table 3.1). These events are recorded to a remote online database upon each occurrence. For example, when the operator engages a HALE in visual task mode, the event id number 5 is tagged with a timestamp of occurrence and recorded in the database. For the purpose of this research, RESCHU is modified to also encode the events into messages and send to the server application through a local area network connection. There are a total of 19 possible events.

Table 3.1: Table of description for the 19 event types

Mode / UV Type	Select Sidebar	Select Map	Waypoint Edit	Waypoint Add/Delete	Goal	Engage Visual Task
HALE	0	1	2	3	4	5
MALE	6	7	8	9	10	11
UUV	12	13	14	15	16	17
ALL	18					

3.2 Server Application

The server application is capable of monitoring multiple instances of RESCHU simultaneously and generates predictions for multiple operators independently. Thus, the SA is required to be able to handle large amounts of complex calculations quickly to minimize the latency between receiving a user-event message from RESCHU and

sending a respective prediction message to the CA. The HSMMs are embedded in the SA for generating predictions. The detailed specifications of the SA and how the predictions are generated are discussed in the next chapter.

3.3 Client Application

The client application is a graphical user interface that displays the visualization of generated predictions. Upon receiving messages from the SA, the CA parses the messages and render the graphics accordingly. The GUI (Figure 3.4) has four main display regions: 1) a graphical trend display, 2) a bar graph, 3) a graphical history view, and 4) an integrated title and alert bar.

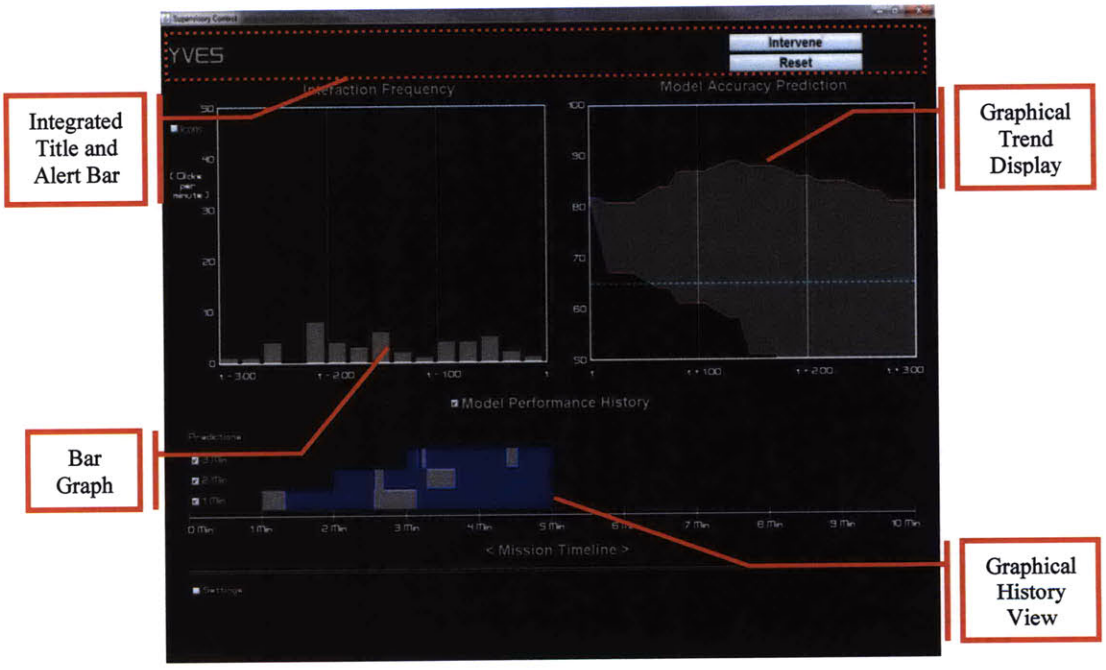


Figure 3.4: Annotated screenshot of the GUI

The graphical trend display provides a visualization for model accuracy prediction. The bar graph illustrates the interaction frequency between a RESCHU system and the operator. The graphical history view tracks the performances of model accuracy forecasts for three different time intervals: one-minute ahead, two-minutes ahead, and three-minutes ahead. The integrated title and alert bar is at the top of the display. The title bar identifies the operator the display monitors, while the alert bar displays alert messages

when they are made available by the system. The rendering technique for each of the four regions is discussed further in Chapter 5.

The graphical trend display is composed of two trend lines. The top trend line represents the model's expected accuracy trend for prediction generation, while the bottom trend represents the lower bound for the accuracy. The horizontal axis is labeled with time spanning from now to three minutes in the future. This time was chosen as the maximum future prediction window since typical user test sessions in RESCHU last 10-20 minutes. The vertical axis is labeled with the model's accuracy score from 50 to 100. This scoring scheme is a result of the abstraction layer design, and is explained in more detailed in Chapter 4.

Between the two trend lines for the Model Prediction Accuracy display is a shaded area that illustrates to the supervisor the uncertainty involved in generating predictions. As can be seen, trend lines are composed of segments that are smaller straight lines. Each trend segment represents a single step of prediction, and the width of the segment is determined by the expected duration of the prediction. In addition, each segment is color coded to represent the expected level of deviation of the actual event from the forecasts for that segment. The levels of expected deviation are marked high, medium, and low by red, gray, and blue respectively. Such trend graphs have been shown to be useful for supervisors of complex systems that need to understand possible system future states (Guerlain, Jamieson, Bullemer, & Blair, 2002; Woods, 1995).

The bar graph in the upper left of Figure 3.4 is composed of 15 bars, each representing the total number of mouse clicks performed by the operator in a 12-second interval. The 12-second was chosen as the time interval because previous data analysis suggested that the average idle time between two user-input events was 12 seconds. The vertical axis measures the number of clicks from 0 to 50, as previous data analysis for RESCHU simulations suggested that operators rarely click more than 50 times within 12 seconds during usual UV operations. The horizontal axis marks the time from three minutes ago to the current time in order to provide the supervisor with a sense of past operator

performance. This display was provided in addition to the model accuracy display since the underlying HSMM inherently relies on mouse-click data. Thus if an operator had a sharp increase in rate of mouse clicks, this could be reflected in a sudden change in model predictions.

The history view at the bottom of Figure 3.4 is composed of three horizontal bars. Each of the three bottom bars represents the models performance as it predicted 1, 2, and 3 minutes into the future. In this view, red signals low accuracy, gray signals medium accuracy, and blue signals high accuracy. The history view helps the supervisor gauge the past performance of the model, which indicates the credibility of the predictions. If the model's past predictions were mostly inaccurate, the supervisor might not want to rely on the model so much when making decisions during operation.

Model historical performance, the bottom of Figure 3.4, is measured by comparing the predicted accuracy score and the actual model accuracy score at the given time interval. For example, if at time T, the model predicted that its accuracy score would be 85 at time T+1, then at time T+1, this value is compared to the actual model accuracy. In this way, the supervisor can determine how effective the model is performing based on prior prediction accuracy. This display is distinctly different from the model perdiction accuracy display in the upper right that is only providing future forecasting information.

3.4 Summary

To summarize, the proposed HSMM-based decision support system assists supervisors in assessing operator performance by generating predictions on operator behavior and marking possible anomalies. The system includes a server-side application (the SA) that monitors multiple operators through sampling operator-RESCHU interactions, as well as a client-side application (the GUI) for displaying the visualization of decision support.

The SA generates predictions on operator behavior using the HSMM developed by Boussemart et al. (2009). Moreover, the SA relies on the abstraction layer to interpret and

transform raw probabilistic values into meaningful indicators on operator performance, before the GUI can display understandable results. The next two chapters present the implementation details of the SA and GUI. The abstraction layer is defined and discussed in detail, and usability tests results are discussed as well as model robustness results.

4 Server Application Design and Implementation

The main purpose of the server application (SA) is to generate predictions using hidden semi-Markov models (HSMMs), and connect them to a decision support tool for a human supervisor. The SA is designed to store user-event messages received from multiple RESCHU simulators to represent multiple operators controlling multiple vehicles. The SA also keeps the entire history of occurred events for each operator in order to generate predictions throughout the duration of each UV operation.

4.1 Design Requirements

Castonia (2009) conducted a cognitive task analysis (CTA) of a team supervisor utilizing a decision support tool during UV operations. This CTA resulted in a set of design requirements for two decision support visualization display (DSV) functions, Problem Identification and Problem Solving (Table 4.1).

Table 4.1: Design requirements for a decision support visualization display

Type	Requirement Description
Problem Identification	1. Allow supervisor to alter user-initiated notification specifications (both the value of prediction accuracy and within what time frame) 2. Alert supervisor to utilize DST when user-initiated notification specifications are met 3. Visualization of HMM prediction accuracy over time 4. Communication capability with operators (should be met with other displays)
Problem Solving	5. Future prediction boundaries (best/worst case scenario) and most-probable prediction accuracy 6. Ability for supervisor to alter time axis to obtain time-range specific data 7. Display automation's prediction of what is causing the drop in prediction accuracy 8. Alert supervisor if prediction accuracy drops at a steeper rate than a supervisor set value

The DSV implemented as a graphical user interface (GUI) according to the above requirements was detailed in Figure 3.4. Given the proposed interface design

requirements of the DSV (Table 4.1), additional system requirements need to be added to Table 4.1 that are specific to the server application. These are detailed in Table 4.2.

Table 4.2: Additional system design requirements for the server application

Requirement	Description
Req1	The SA should operate in real time
Req2	The SA should monitor one or more UV operators
Req3	The system should implements a Client/Server architecture
Req4	The SA should utilize the HSMMs developed by Boussemart et al. (2009) to perform calculations and generate predictions
Req5	The SA should communicate with RESCHU and DSV over a network

The requirements are discussed in more detail below.

Req. 1: The SA should operate in real time.

As a real time predictive decision support tool, the SA must be able to react quickly to each operator-RESCHU interaction in order to allow the supervisor to monitor the operators' most current behaviors. However, generating predictions is a computationally intensive process, and as a result, there is an inherent time delay between detecting an operator input and displaying predictions on the GUI. This time delay is defined as the latency of the system. Due to the time-critical nature of UV operations, the decision support tool must operate with minimal latency.

Previous research indicates that a system latency of under 500ms is imperceptible to humans (Claypool, 2005). However, when latency is above 500ms, system users have to make adjustments in their decision making process. In the case of this decision support system, unnecessary stress is added to the supervisors if latency becomes higher than 500ms. Hence, there is a need to keep system latency to less than 500ms. Moreover, system testing suggested that system latency grows linearly with the number of operators monitored. Since the interface was designed to support up to 10 operators under simultaneous supervision (Castonia, 2009), in order to maintain overall system latency under the limit, latency contributed by a single RESCHU instance is kept under 50ms.

Figure 4.1 shows the maximum amount of computation time per user if the system is to maintain sub-500ms system latency, and the figure is derived from dividing 500ms by the number of operators under supervision.

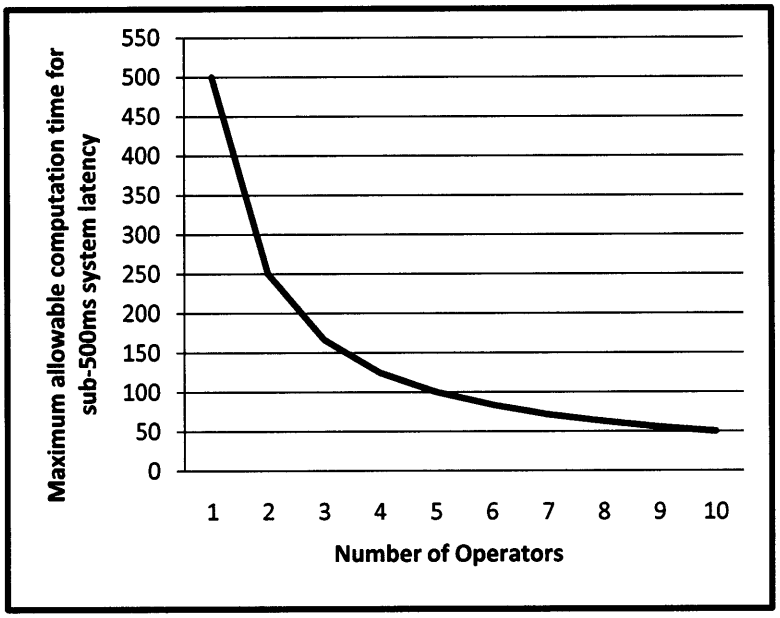


Figure 4.1: Operator vs. latency contributed by a single RESCHU simulation

Req. 2: The SA should monitor one or more UV operators

Since the system aims to assist the supervisor in monitoring multiple operators simultaneously, the SA needs to be able to monitor multiple RESCHU simulators at the same time. Consequently, the server application needs to perform multiple computationally intensive operations concurrently in order to generate predictions on the behaviors of multiple RESCHU operators and still maintain real-time operation (Req1). Possible challenges include keeping latencies under limit.

Req. 3: The system should implement the Client/Server architecture.

The Client/Server architecture is a common software system architecture that allows the system to be built from modular client and server software as well as work independently

and collaboratively. The level of independence between the components also allows the system to continue operating in case of a single point failure and makes recovery relatively simple. The architecture allows the system's components to be modified independently when future requirements emerge. This is particularly important in this design since the HSMM and GUI aspects of this system are also research projects, and therefore have not achieved stability. Since both of these system components will likely change with new experimental results, designing a modular and extensible system is critical for this research and development phase of system design.

Req. 4: The SA should utilize HSMMs to generate predictions.

The decision support system leverages HSMM-based predictions, and the predictions generated should maximize the supervisor's ability in assessing the performances of multiple operators to avoid costly failures in operation. A Java HSMM library was developed by Boussemart et al. (2009) for this project. The HSMM library is used to learn the parameters of HSMMs modeling the behavior of a RESCHU operator, given previously existing data sets. The resulting trained model is encoded into a text file. The library also provides a number of key functionalities including parsing an encoded text file into an HSMM Java object. The HSMM Java object can then be used to generate predictions. Therefore, the SA was implemented in Java in order to facilitate the integration with the Java HSMM library.

Req. 5: The SA should communicate with RESCHU and DSV over a network.

The only way to simultaneously monitor multiple UV operators on separate machines is to connect these machines over a network. Moreover, the networking approach provides the flexibility to scale the number of monitored RESCHU instances relatively easily. Given the Department of Defense's stated policy to move towards network-centric operations (Alberts, Garstka, & Stein, 2000), the use of a network for this kind of command and control environment is also an operational necessity.

In summary, the HSMM-based predictive decision support system's scalability depends on the speed and efficiency of the prediction-generating algorithm. Hence, during the development process, design decisions were made to optimize the system for speed since it is the only constraint to the system's scalability. How these requirements were fulfilled is discussed in the next section.

4.2 Overall System Architecture

The server application is composed of three main modules: 1) the connection module and 2) message dispatch module, and 3) the abstraction layer module. These modules work together to generate predictions and provide the client application with display data. An overview of system architecture is shown in Figure 4.2.

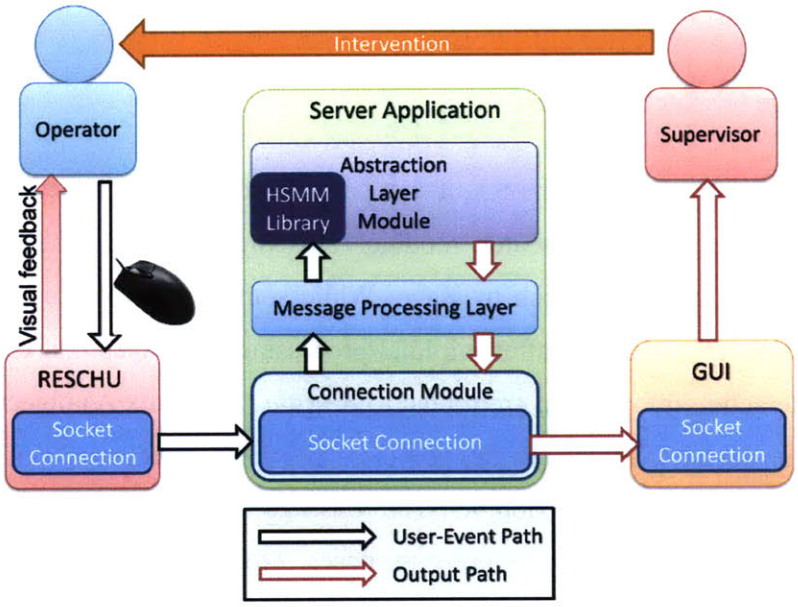


Figure 4.2: Overview of system architecture

4.2.1 Connection Module

The connection module handles connection logistics for the server application. It creates connections to the client application and to the RESCHU simulators by instantiating socket objects provided by the Java Networking API library (Sun Microsystems, 2008).

Socket objects create point-to-point access between two machines over a network. Connections are necessary because messages are passed from each RESCHU instantiation to the SA and from the SA to the GUI during operation (Figure 4.2).

Both RESCHU and the SA utilize the *DatagramSocket* class for sending messages. The SA and the GUI utilize the *MulticastSocket* class for receiving messages. The *DatagramSocket* class is part of the Java Networking API (Sun Microsystems, 2008), providing basic capabilities to send and receive data over a network. The *MulticastSocket* class is a subclass of the *DatagramSocket* class, providing additional capabilities for joining "groups" of other multicast hosts on the Internet. Joining a multicast group is similar to tuning into a radio channel, where many listeners can receive the data from a multicast group simultaneously. It allows multiple RESCHU programs to send messages to the same multicast group, while the SA is waiting to receive data.

4.2.2 Message Dispatch Module

When a message is received by the SA, it first passes through the connection module and then arrives at the message dispatch module. This module parses the messages and determines the destination of each message. While some messages are quickly passed back to the connection module after minimal processing, messages that carry operator-RESCHU interaction information are routed to *Session* (Appendix A) objects residing in the Abstraction Layer Module. The *Session* class is implemented as part of the SA API, which is discussed further in section 4.3. The Message Dispatch module keeps a map of usernames to *Session* objects for routing purposes. Moreover, Messages are formatted according to a communication protocol, i.e. the definition of how each type of messages should be formed, to ensure consistency.

4.2.2.1 Communication Protocol

Operators begin RESCHU simulations by first logging onto RESCHU with their unique usernames. Because the SA monitors multiple operators simultaneously, messages received from different RESCHU simulator must be routed individually according to the

operator's username, and thus, each message begins with the operator's username. There are four types of message sent from RESCHU to the SA:

- RESCHU-begin notice (RBEG)
- RESCHU-end notice (REND)
- Interaction notice (INT)
- User-event message (USER-EVT)

There are also four types of messages sent from the SA to the GUI as well:

- Session-begin notice (SBEG)
- Session-end notice (SEND)
- Click notice (CLK)
- Generated predictions (PREDICT)

The grammar of these messages is described in Table 4.3.

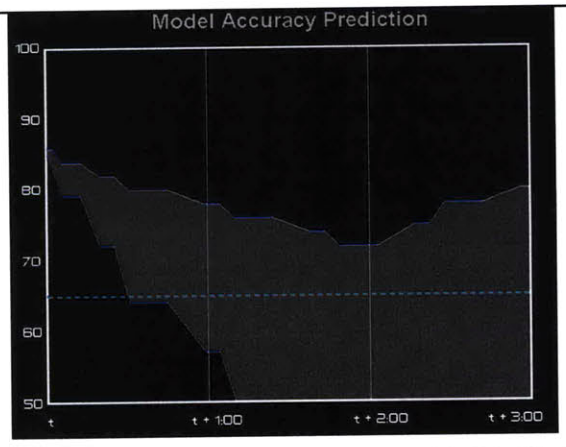
Table 4.3: Grammar of messages

Message	**Grammar**
RBEG	USER DELIM "BEGIN"
REND	USER DELIM "END"
INT	USER DELIM "CLICKED"
USER-EVT	USER DELIM TIME DELIM EVENT
SBEG	RBEG DELIM TIME
SEND	REND
CLK	INT DELIM TIME
PREDICT	HEADER SEP BODY
Variables	**Symbols**
HEADER	USER DELIM TIME DELIM ALRT
BODY	PRED [DELIM2 PRED]$^{+}$
PRED	TIME DELIM SCORE DELIM CONF DELIM LOW
USER	[A-Za-z]$^{+}$
DELIM	" : "
DELIM2	" ; "
EVENT	0 | 1 | 2 | ... | 18
SEP	" @ "
SCORE	50 | 51 | 52 | ... | 100
LOW	50 | 51 | 52 | ... | 100
CONF	[0-9]$^{+}$
TIME	[0-9]$^{+}$
ALRT	[0-9]$^{+}$

The RBEG signals the SA that the current RESCHU simulation has begun, while the REND marks the end of the current RESCHU session. The interaction notice includes a short message that signals the SA that a mouse-click occurred. The user-event message encrypts the time of occurrence, and the event identification number.

The SBEG signals the GUI that an operator session has been initiated on the server, and the SEND signals that the session had ended. The CLK notifies the GUI that the operator had clicked on RESCHU and the time it occurred. The prediction message (PREDICT) encodes the generated predictions.

The SA tags the first three types of messages from RESCHU with time information and then relays them to the GUI. Each prediction message is composed of a header and the body. The header includes the name of the operator and the time that the prediction was generated, while the body contains the generated predictions and added data to be displayed by the GUI. The body can be further separated into multiple segments delimited by semicolons, where each segment of the body encrypts the data required to populate a segment of the trend line described in Chapter 3. Below is an example of a PREDICT message and its corresponding trend graph. (Figure 4.3)




```
yves:1248894105224:0@14:86:0:86;14:84:128:79;11:82:128:72;2
8:80:128:64;11:78:128:57;28:76:128:50;11:74:128:42;28:72:12
8:35;11:75:128:33;28:78:128:30;11:80:128:27;28:80:128:27;
```

Figure 4.3: An example of a PREDICT message and its corresponding trend graph

As shown, `"yves:1248894105224:0"` is the header of the message, where `"yves"` is the operator's name. The first number is the difference, measured in milliseconds, between when the prediction was sent from the SA and midnight, January 1, 1970 UTC, which is Java's standard way of storing time. The second number encodes the alert type number, where 0 means no alerts to be displayed. Following the `"@"` is the body of the message. There are sets of four numbers separated by semicolons where each number is separated by colons. These numbers result from the abstraction layer interpreting HSMM results, and they represent the expected duration, expected accuracy, prediction confidence, and the lower bound in order. For example, the second segment of the body, `"14:84:128:79,"` represents that the first prediction is expected to be valid for 14 seconds with an accuracy score of 84, lower bound for accuracy score of 79, and a prediction confidence level of 128. These metrics will be explained in section 4.3.

While USER-EVT messages are routed to their corresponding *Session* objects, the other three messages are directly returned to the connection module to be sent to the GUI after slight transformations. More specifically, RBEG is transformed into SBEG by appending a time tag, and INT transforms to CLK in the same way, while REND is kept unchanged to form the SEND message. Figure 4.4 is a handshake diagram showing how messages are passed among two instantiations of RESCHU (representing two different operators), the SA and the GUI.

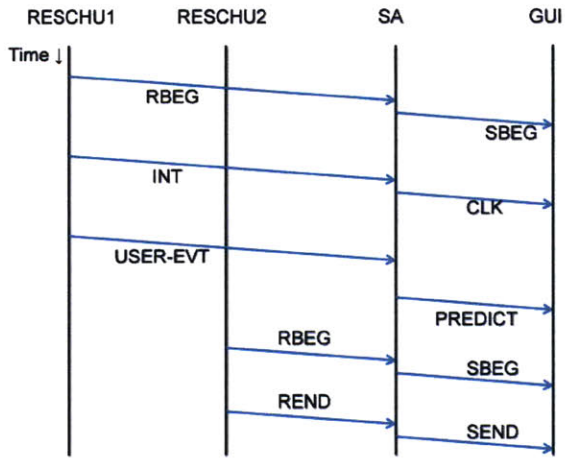


Figure 4.4: A handshake diagram showing the passing of messages

4.2.3 Abstraction Layer Module

The abstraction layer is the key module that generates predictions. In the prediction generation process, the HSMM library only provides the ability to calculate the probabilities of occurrences for sequences of user-events, while the abstraction layer interprets these probabilities to generate predictions and the PREDICT message. The prediction-generating (PG) algorithm is based on these HSMM-based probabilistic calculations. Examples of inputs and raw results of these probabilistic calculations are illustrated by Figure 2.2. As stated previously, these raw results are difficult to understand by non-technical personnel, and there is no obvious way to visualize these raw results for easier user comprehension. As a result, the abstraction layer is created to translate raw HSMM-generated prediction results into human-readable trend graph data.

Since the SA monitors multiple operators, the prediction-generation process for each operator needs to be separated. The module contains a collection of *Session* objects to keep track of the profiles of multiple operators. Each *Session* object is responsible for generating predictions for the operator it monitors and for storing information relevant to the operator. Moreover, the prediction generation process can be divided into three stages: 1) event aggregation, 2) probabilistic calculation, and 3) interpretation.

4.2.3.1 HSMM Library

The existing HSMM library (Appendix B) developed by Boussemart et al. (2009) provides the functions listed in Table 4.4, and the functions are described in detail below.

Table 4.4: HSMM Library functions

Function	**Input**	**Output**
mostLikelyStateSequence	int[] eventSequence	int[] stateSequence
logProbability	int[] eventSequence	double logProb

mostLikelyStateSequence

This function takes a sequence of event IDs in the form of an integer array. The function implements the Viterbi algorithm with forward-backward recursion and returns an integer sequence of the mostly likely state number sequence. For example, for a three-state HSMM, given a sequence of observable events [O_1, O_3, O_2, O_2, O_2, O_4, O_5, O_5, O_1], the method returns a most likely sequence of hidden state numbers, [S_1, S_2, S_2, S_2, S_2, S_3, S_3, S_3, S_1].

logProbability

This function takes a sequence of event IDs in the form of an integer array and calculates the probability of occurrence of that sequence of events. However, the probability of seeing a sequence of events decays rapidly as the sequence gets longer, and thus, the method returns the natural log of the probability. For example, given a sequence of events [1,1,1,2,2,8,8,8,9,9], the method returns the probability of the sequence, $P(1)P(1|1)P(1|[1,1]) \ldots P(9|[1,1,1,2,2,8,8,8,9,9) = P([1,1,1,2,2,8,8,8,9,9])$, which is a product of a series of conditional probabilities. Moreover, a longer chain of conditional probabilities could cause computational underflows (i.e. the probability becomes too small for a computer to store as a number[1]), which frequently occurs after just a few minutes in a RESCHU simulation. For example, consider the worst case scenario where each of the top five expected events is always equally likely to occur with probability 0.2 (i.e. $1 \div 5$), then $P([1,1,1,2,2,8,8,8,9,9]) \leq 1.024 \times 10^{-7}$. As the length of event sequence grows larger than 500, the probability of the entire sequence drops below 3.27×10^{-350}, a number that is equivalent to 0 to a computer. A common computational work-around is to use the natural log of the probability instead. So, in the previous example, $\ln(3.27 \times 10^{-350}) = -807.72$.

As previously stated, the HSMM includes a state transition matrix, A, an observable emission matrix, B, and a duration distribution matrix, D. Together, these provide state

[1] The smallest positive double for Java is 2.225e-308

transition probabilities, emission probabilities, and probability distribution of durations, for each state respectively. Additional Java functions are written to augment the HSMM library. These functions provide basic calculations involving the matrices (Table 4.5), and the functions are described in detail below.

Table 4.5: Additional functions supported by the HSMM library

Function	Input	Output
getNextState	int currentStateNumber	int nextStateNumber
getTopObservables	int currentStateNumber	List<EPPair> events
getExpectedDuration	int currentStateNumber	int expectedDuration

getNextState

This function takes an integer input, *currentStateNumber*, and returns an integer as an output, *nextStateNumber*, representing the most likely next state following the current state. This result is obtained from calculating the j index with the highest probability in the i^{th}column of the matrix A, where i is the *currentStateNumber*.

getTopObservables

Given the current state number, this function outputs the top five most likely observable events for this state. The function only outputs the top five because the top five most likely observables in combination usually cover more than 99% of the emission probabilities, and therefore, the rest of the observables can be neglected as they are rarely observed (on average, less than 0.01% probability each). This method accesses the observable emission matrix described in section 2.1 and groups the j indices (event numbers) with their probabilities in pairs. Only the top five events are recorded, and their probabilities are renormalized to sum up to one. These event number and probability pairs are stored as *EPPair* objects, which can be sorted by probabilities. The top five *EPPair* objects in descending probabilities are grouped into a List object and returned as output.

getExpectedDuration

Given the current state number, this function outputs the expected duration of the state, calculated from the duration distribution matrix of the HSMM.

4.2.3.2 Session Object and Event Aggregation

The *Session* class implements functions and data structures that are responsible for processing predictions and performance data of an operator. A *Session* object is instantiated on the SA when a RBEG message is received, and it is discarded when the SA receives the REND message. Upon receiving a USER-EVT message, the *Session* object parses the remaining information from the message and stores the TIME-EVENT pair in a List data structure. The collection of the TIME-EVENT pairs is used as input to the HSMM for generating predictions on operator behavior. After predictions are generated in the abstraction layer, the results are passed back to the Message Dispatch Module. The Message Dispatch Module then formats the results into a PREDICT message and passes the message to the Connection Module for transmission to the GUI.

As previously mentioned, the HSMM accepts inputs as an array of integers. For example, [1,1,1,2,2,8,8,8,8,8,8,8,9,9,9,9] represents a sequence of events where there is a 3-time-unit[2] delay between event 1 and 2, a 2-time-unit delay between event 2 and 8, and a 7-time-unit delay between event 8 and 9, while no new event occurred for 4 time units since event 9 arrived. However, events are added to the list data structure in the *Session* object only when a USER-EVT message is received. As a result, there are gaps between events. To determine the amount of time passed since the last received event, the SA can subtract the event time from the current system time. However, this adds unnecessary computational stress to the SA each time a prediction sequence needs to be generated.
As a solution, the *Session* object performs the auto-filling process, which automatically adds replicas of the last received event to the List data structure every second while not receiving new events, as illustrated in Figure 4.5. As a result of the auto-filling process,

[2] In this project, HSMMs were trained and built to use integer values of delays, which may represent sub-second resolution.

the list of received events can be easily transformed into the integer array of event numbers required by the HSMM library as input. Without the process, some events could be lost during the transformation. As seen in Figure 4.5, the incorrect sequence generated excludes the three most recent events (event 9's).

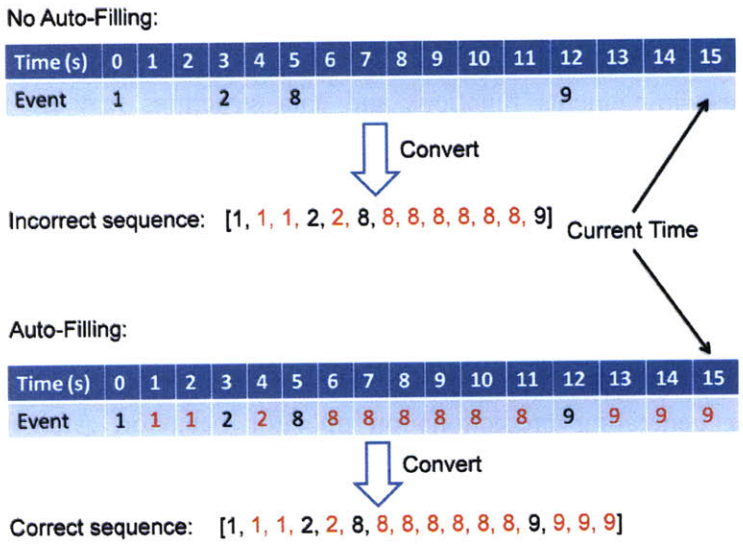


Figure 4.5: Auto-filling vs. no auto-filling

4.3 HSMM-to-GUI Translation

Because of the difficulty involved in non-technical personnel deciphering HSMM raw results, the abstraction layer is designed to translate the results for the supervisors. The abstraction layer bridges the gap between probabilistic data and the information that the supervisors need to assess operator behavior. Since the HSMM is trained to recognize normal operator behavior patterns, it is generally desirable that the operator's behavior follows the model's expected behavior closely.

In the original conception of the decision support visualization (Castonia, 2009), a GUI was proposed to display the HSMM's prediction accuracy on the trend display. The decision support GUI was originally designed to show a probability on the y axis, which represents the probability that the HSMM will make a correct prediction (Figure 4.6). However, this design illustrates a common disconnect between designers of interfaces

and the designers of algorithms. An HSMM does not actually output the probability of the model making a correct prediction in the way that designer of the GUI in Figure 4.6 anticipated. An HSMM, instead, outputs the probabilities of candidates for the next hidden state and next observable event, as well as the probability distribution of duration. However, it is unclear which probabilities should be displayed and how. Thus, one important aspect of the abstraction layer is to develop a methodology to make the desired interface work with the actual output of an HSMM.

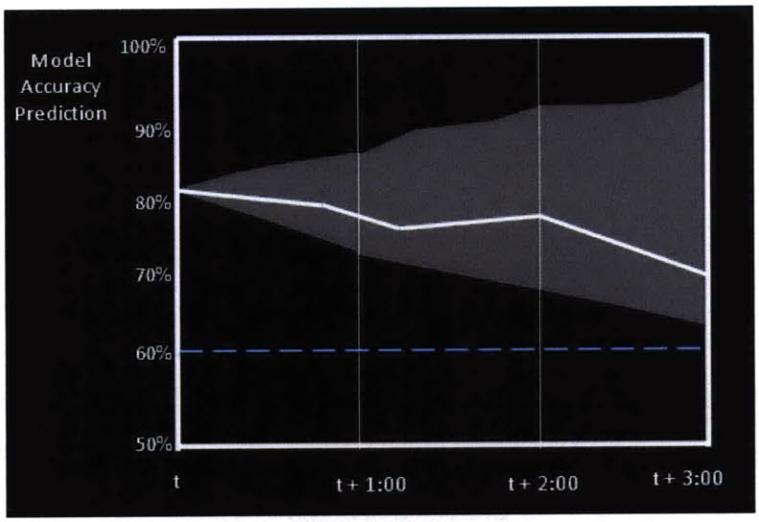


Figure 4.6: Original design of the trend display (Castonia, 2009)

As discussed previously, the HSMM can only be used to calculate the probability of occurrence for an event, given a history of previous events (Figure 2.2). Because there are different likelihoods for the different possible next events, there exist multiple possible probabilities for next possible observable event. In order to address this fact, an expected accuracy score can be calculated given that each predicted event is assigned an accuracy score.

One novel contribution of this SA effort is the development of an HSMM model accuracy scoring metric, which measures the HSMM's ability to predict operator behavior. This metric, the Model Accuracy Score (MAS), ranges from 50 to 100 and is derived by evaluating the moving average of the previous 10 sub-scores. Sub-score is a weighted

average of two factors, 1) the quality of the last HSMM prediction, and 2) the timing of the prediction, and is given in (1).

$$\text{Subscore} = 50 + \alpha \times Quality + (1 - \alpha) \times Timing \quad (1)$$

Both quality score and timing score range from 0 to 50 to ensure that the sub-score ranges from 50 to 100. Sub-scores are always rounded to the nearest integer, as integer scale is required by the trend display. The moving average of 10 sub-scores is used to smooth the accuracy trend and allows the abstraction layer to be more forgiving if a few predictions do not perform as well. The moving average window is chosen to be 10 sub-scores because it maintains a good balance sensitivity and tolerance to both good and bad predictions.

4.3.1 Quality

Instead of prediction accuracy, a quality of prediction score is proposed to more accurately represent an HSMM output. The quality of the prediction is measured by the rank of the incoming event among the top five expected events in descending order of their occurrence probabilities, and varies from 0 to 50, as illustrated by Table 4.6. For example, if the incoming message encodes event 12 as the third expected event, then the sub-score for this prediction is 40. This rubric is skewed to apply heuristic clustering (Davis, 2003) for the top three expected events. While the top five expected events cover more than 99% of the emission probabilities, the top three expected events often cover more than 80% of the emission probabilities, and thus should be considered more heavily than the bottom two in a supervisor's decision making process. To enforce this bias, the top three scores are grouped within 5 points away from each other, while a more significant drop in scoring is present between the third and forth ranks of scores (Table 4.2). It is important to note that the range of prediction accuracy scores is kept as 50 to 100, mimicking the 50% to 100%, as seen in Figure 4.6.

Table 4.6: Rubric for quality sub-scores

Event Rank	Score
1st	50
2nd	45
3rd	40
4th	20
5th	10
Not in rank	0

4.3.2 Timing

When the list of top five expected events is generated, the HSMM also is expecting to receive one of these five events within a certain amount of time. This time interval is defined by the expected duration of the prediction. As a result, the timing of the received event is also used to measure the accuracy of a set of predictions.

The prediction receives a timing score based on the number of standard deviations (SD) away from the expected duration when the actual event arrives. If the actual event arrives within one SD of the expected duration, the prediction receives full marks on timing. If the actual event arrives between 1 and 2 SD away from the expected duration, the prediction receives a 14 (i.e. $50 \times 0.136 \times 2$) as the timing score, where 0.272 (i.e., 0.136×2) is the probability of being between 1 and 2 SD in a Gaussian distribution (Figure 4.7). Likewise, if the event arrives after 2 SD from the expected duration, the prediction receives a 2 (i.e. $50 \times 0.021 \times 2$). If the event arrives more than 3 SD away from the expected duration, a timing score of 0 (i.e. $50 \times 0.001 \times 2 \cong 0$) is assigned.

As a result of this timing consideration in the sub-score, the Model Accuracy Score is penalized if events do not arrive in a timely fashion as predicted by the model. For this project, the SA expects an event up to 1 SD (inclusive) before or after the expected time of arrival (ETA). Beginning at 2 SD after ETA, the model is penalized at every SD interval, and a new sub-score is assigned to reflect the penalty. The quality score of that sub-score is always zero because no event is received, which is worse than receiving a

non-predicted event, while the timing score varies with the number of SD in terms of time passed.

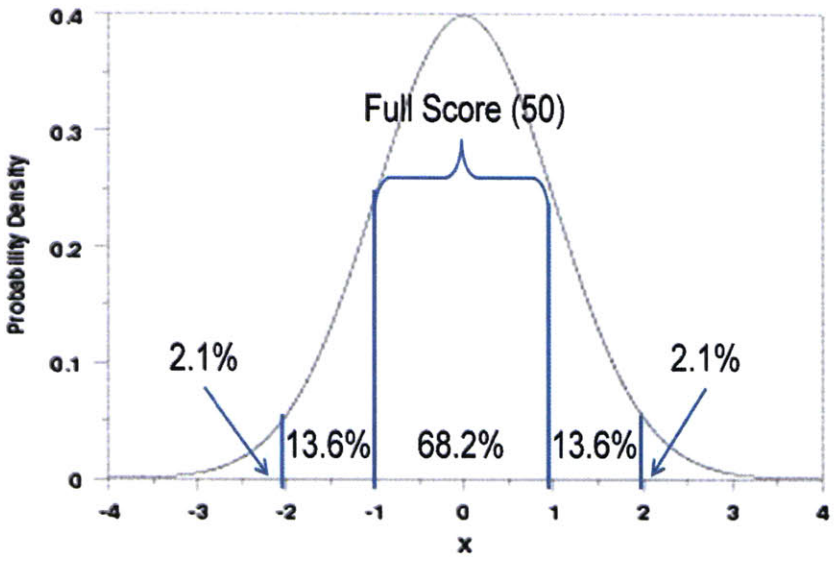


Figure 4.7: Gaussian distribution (Heckert & Filliben, 2003)

Every time a new sub-score is assigned, the MAS is recalculated, and a new PREDICT message is generated to notify the GUI client of the change in MAS. The *Session* object is responsible for the timing of these penalties for every prediction of each operator. These penalties begin at 2 SDs after ETA time, and are assessed at every 1 SD until a new event is received. ETA is measured by the expected duration of the current state (i.e. if the expected duration of the current state is 16 seconds, ETA is t+16 seconds).

In (1), the value of α, the weighing factor, is dependent on the time sensitivity of an HSMM and the time sensitivity of the timing metric itself, and an α of 0.9 was found to be appropriate for this work because the timing metric was designed to be more punishing than the quality metric. While the quality metric has five tiers of scoring, the timing metric has only three (based on the three standard deviations). Since time is critical in these supervisory control domains, even if the next predicted event occurs, but at the wrong time, this is counted as essentially a poor prediction.

4.3.3 Expected MAS, Lower Bound, & Prediction Confidence

The expected MAS represents a prediction on the future Model Accuracy Score (MAS), and the lower bound represents the expected minimum predicted MAS. The prediction confidence indicates the credibility of a given prediction. These numbers form the trend graph in the GUI.

The Model Accuracy Scoring metric is also applied to generated event predictions so as to compute accuracy score predictions. The abstraction layer uses the scoring metric to calculate the expected MAS from the generated predictions. The MAS and the expected MAS form the trend lines that can be visualized on a graphical display and more easily understood by the supervisors (Figure 4.8).

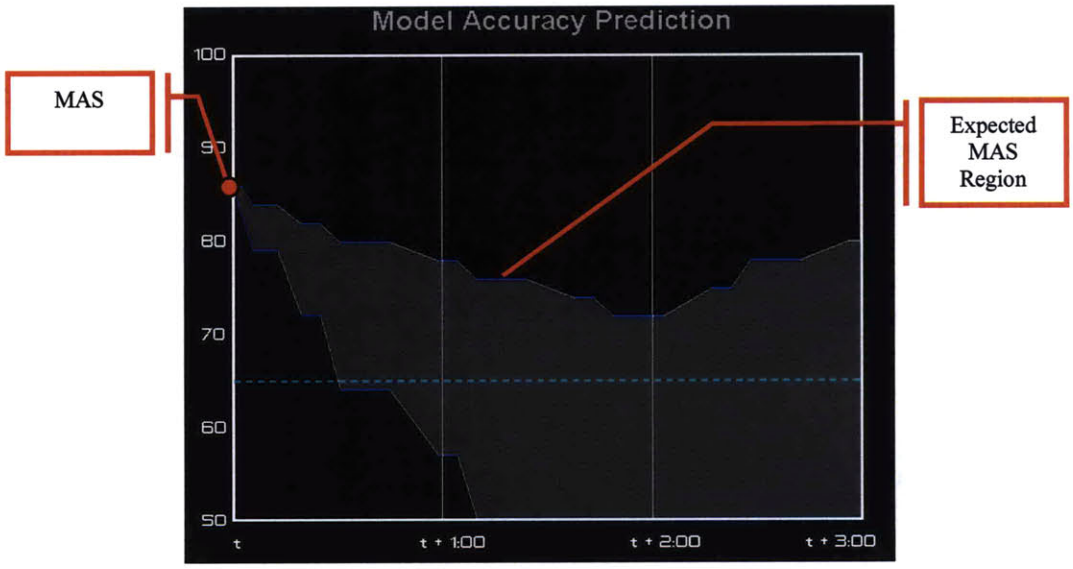


Figure 4.8: MAS and expected MAS on trend lines

The expected MAS represents the expected predicted accuracy, and is calculated from 10 expected sub-scores, while the lower bound is calculated from 10 expected low-scores. Heuristic clustering is also applied in that the expected sub-score is the expected value of the top three prediction scores, and the lower bound is the expected value of the bottom three prediction scores. Accordingly, both are derived from a list of expected events and their probabilities. Moreover, expected sub-scores are calculated with the assumption that

the prediction receives perfect timing score (i.e., 50), and the opposite is assumed for expected low-score (i.e. timing is 0). For example, given a list of five expected events [11, 10, 7, 4, 9] with probabilities [0.34, 0.28, 0.18, 0.11, 0.09], the expected sub-score is 88, and the expected low-score is 60, calculated as below.

$$EV[subScore] = 0.9 \times (0.34 \times 50 + 0.28 \times 45 + 0.18 \times 40) + 0.1 \times 50 + 50 \cong 88$$
$$EV[lowScore] = 0.9 \times (0.18 \times 40 + 0.11 \times 20 + 0.09 \times 10) + 0.1 \times 0 + 50 \cong 60$$

The sub-score is then added to the current list of sub-scores, from which the new expected MAS is calculated. Likewise, the low-score is added to the current list of low-scores to calculate the lower bound.

Each prediction's confidence is derived from the top five most likely events. Confidence in the events prediction is high if less variability is present in the prediction. Hence, the confidence level is derived from the variability in the predictions, which is measured by the variance of the prediction. Variance of the prediction is calculated as the expected value of the squares minus the square of the expected value. In addition, only the variance of the quality of the predictions is measured, and therefore, the timing score of the predictions is set at 50, the maximum timing score, to be consistent. For the same example as above, the resulting five possible scores are listed in Table 4.7.

Table 4.7: Sub-score table assuming perfect timing score

Event Rank	Quality Score	Sub-score
1st	50	$50 + 0.9 \times 50 + 0.1 \times 50 = 100$
2nd	45	$50 + 0.9 \times 45 + 0.1 \times 50 = 95.5$
3rd	40	$\ldots = 91$
4th	20	$\ldots = 73$
5th	10	$\ldots = 64$

Moreover, the value is always rounded to the nearest integer for simplicity, and the variance of the example above is calculated as in (2).

$$Var(X) = E[X^2] - E[X]^2$$

$$= (0.34 \times 100^2 + 0.28 \times 95.5^2 + 0.18 \times 91^2 + 0.11 \times 73^2 + 0.09 \times 64^2) - 90.91^2$$
$$= 8399.08 - 8264.6281$$
$$= 134.4519 \approx 134 \quad (2)$$

$E[X]$ is calculated as in (3).

$$E[X] = 0.34 \times 100 + 0.28 \times 95.5 + 0.18 \times 91 + 0.11 \times 73 + 0.09 \times 64 = 90.91 \quad (3)$$

4.3.4 Prediction Sequence Length

The DSV displays predictions up to three minutes (180 seconds) into the future on the trend display. However, since the decision support tool operates in real-time, the trend display is dynamic. The trend lines continuously shift to the left of the screen each second by the width of one second (Figure 4.9).

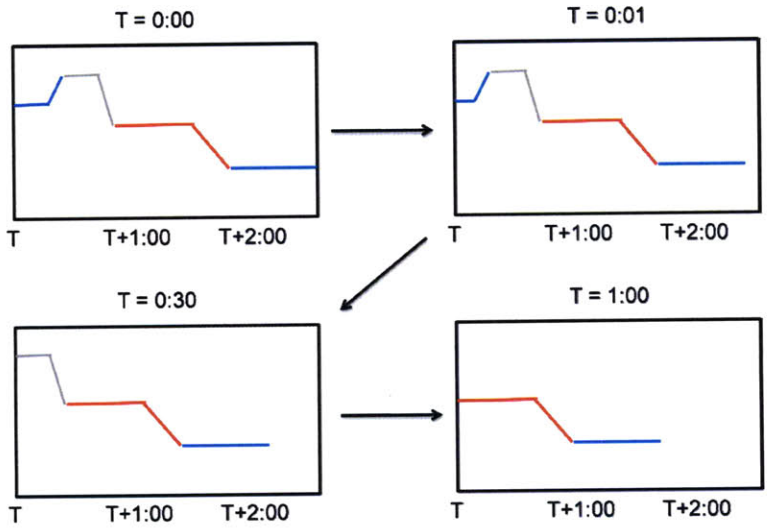


Figure 4.9: Shifting trend lines

To ensure the trend display remains populated the entire time between event arrivals, the abstraction layer module generates 210-second long prediction sequences to provide a 30-second buffer for the trend graph, while the SA is waiting for the next event to occur in RESCHU. Thirty seconds is chosen to be the length of the buffer because either the

timing penalties begin at most after 48 seconds or on average, the next event arrives after 12 seconds, as this is the average time of user interactions (see section 3.3). Thus, the average of 48 and 12 is 30.

4.3.5 Prediction-Generating Algorithms

The heart of the abstraction layer is the prediction-generating (PG) algorithm that interprets raw HSMM outputs and combines these outputs to form the data required to populate the trend graph. As introduced in section 4.2.2.1, a sequence of sets of four numbers (expected duration, expected MAS, prediction confidence, and lower bound) is required to populate each trend graph, which is updated when a new event occurs in RESCHU.

Both a simple method and a more exhaustive method that generate the trend graph sequence were investigated when developing the PG algorithm to connect the HSMM to the GUI, and are discussed below.

4.3.5.1 Simple Method

The simplest way of generating predictions is simply to traverse the HSMM structure by following state transitions and obtaining emitted observables:

1. Obtain the current state by using the *mostLikelyStateSequence* method given the sequence of all of the received events.
2. Obtain the expected duration associated with the current state.
3. Obtain the top five most likely expected events by using the *getTopObservables* method, with the current state number as input.
4. Calculate expected MAS given the top three of the five most likely events.
5. Calculate the lower bound given the bottom three of the five events.
6. Calculate the expected deviation from the five events.
7. Encode the four numbers into a PRED segment.
8. Append the top event to the sequence of received events.
9. Repeat 1-8 until desired number of predictions has been generated.

10. Append all PRED segments and format them into a PREDICT message.

To generate 210-second long sequences of predictions, steps 1-8 would be repeated until the sum of the expected durations is greater than or equal to 210. The simple method generates a 210-second long sequence of predictions in less than 10ms. However, its simplicity limits its predicting power: the method makes cumulative assumptions with every predictions step, creating additional error. These assumptions are:

1) Most probable next state is always the next achieved state
2) The five most expected events of that state are the five most expected events in general.

For example, during each iteration of prediction generation, once a most likely state is obtained, that state is assumed as the next state with high certainty. As a result, the method also assumes that the most expected events of that state are also the most expected events among all of the candidate states for the next step. However, the second most likely state could be almost as probable. Moreover, the list of most likely events is generated using the assumed current state; however, the second most likely state could also generate a list of events that are equally likely as the first list, as illustrated in Figure 4.10.

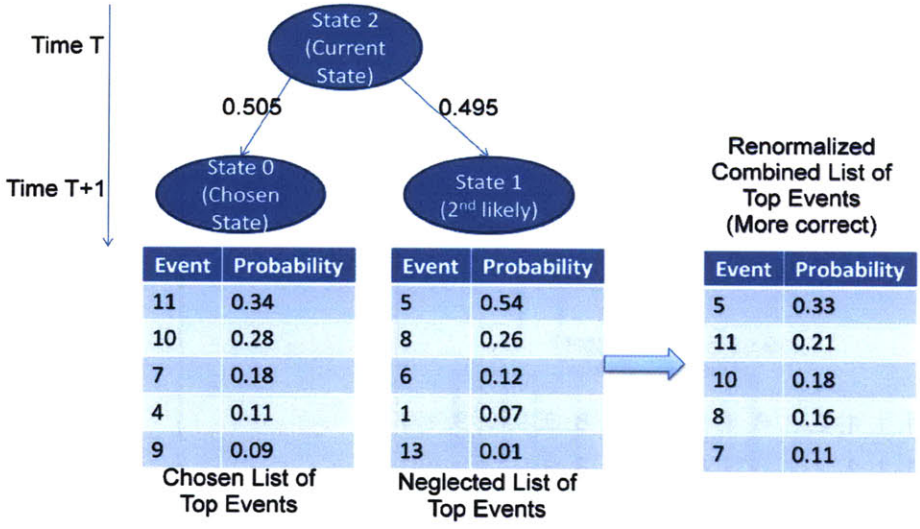


Figure 4.10: Scenario where a combined list is more correct than the chosen list

4.3.5.2 *Exhaustive Method*

The more rigorous method for generating predictions makes no assumptions about certain states or events during the process. Instead, it considers the full extent of the probabilistic distributions across hidden and observable states.

For this method, the top five most expected events are obtained by the following procedures.

1. For event 0 to 18, append each event to the current list of occurred events, and calculate the probability of seeing the new sequence of events using *logProbability*.
2. From the 19 probabilities calculated, sort and record the top five events with the highest probabilities.
3. Convert the logs of probabilities to normal probabilities.
4. Normalize these five events so that their probabilities sum to one.

This set of procedures ensures that every event is considered as the next expected event without designating a current state beforehand, and thus each event is considered without bias. As illustrated in Figure 4.11, each possible event is appended to the current sequence of events, and *logProbability* method is invoked for a total of 19 times on these new sequences of events to obtain the log probabilities of these new events.

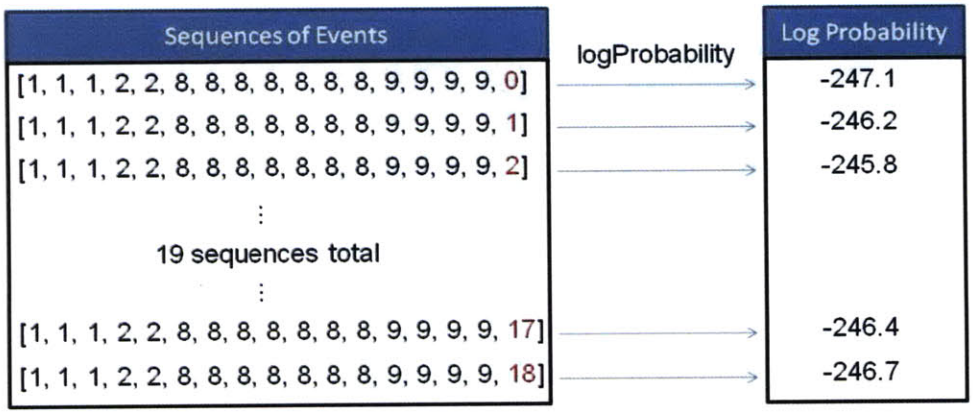


Figure 4.11: *logProbability* is invoked 19 times per step of prediction generation.

These events are then sorted by their probabilities, and the top 5 events are recorded as the five most expected events. Moreover, the probabilities of the top 5 events are renormalized to sum up to one. The expected MAS and the lower bound are then calculated the same way using these five events. Prediction confidence can be derived using the same technique stated previously. This method for generating predictions leaves the element of probability distribution embedded in the predictions and therefore, is more rigorous than the previous naïve method.

Because this method does not assume a single state as the current state, the method calculates the expected duration of a given prediction step by weighing every state's expected duration by the state's probability, as illustrated in Figure 4.12. To obtain the probabilities for all candidates for the current state, a variation to the *mostLikelyStateSequence* method returns an array of probabilities indexed by the state numbers for the last state in the state sequence, instead of returning the most likely state sequence.

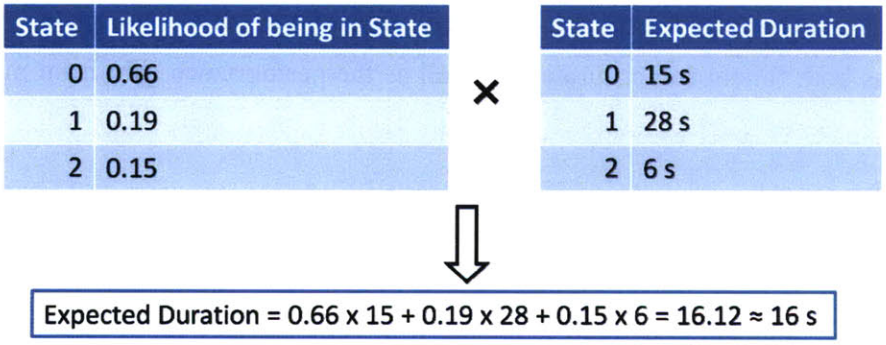


Figure 4.12: Calculating expected duration from probability and duration of each state

This method is more rigorous than the simple method because fewer assumptions are made during prediction generation. As a result, every hidden state and every event is considered and treated equally at each step of the prediction generation. Thus, this method should in theory has a better predictive ability, and is more favorable than the simple method. However, it requires a significant amount of computation due to the fact

that *logProbability* method is called 19 times at every single step of prediction generation. Moreover, the length of the input to the method grows at each subsequent step of the process. It costs the abstraction layer much more time to generate a PREDICT message than using the naïve method. On average, a PREDICT message that predicts up to 210 seconds into the future takes more than five seconds to generate using the rigorous method, and is unacceptable given the system requirements (see section 4.1).

Computational enhancements were implemented to solve the latency issue with this method (discussed in Chapter 6). After implementing the enhancement, the rigorous method is able to generate a 210-second long sequence of predictions in 50ms. Thus, this method was chosen as the final prediction-generating method for its correctness without sacrificing system performance (Appendix A).

To satisfy all of the system requirements described earlier while keeping the prediction ability of the model high, compromises have to be made. As in all complex, multivariate time pressured settings, there are inevitable trade-offs between speed and correctness of the predictions. In order to promote the rapid computations needed for a real-time decision support system, Chapter 6 will detail those trades that were made to generate predictions both timely and accurately as well as the performance gain from making the trades.

4.4 Alerts

The PREDICT messages have the capability to encode the type of alerts to be displayed by the GUI client, discussed in the previous chapter. Although the library of alerts has only been proposed but not developed (Castonia, 2009), the SA has the capability of discerning the nature of an operator behavior anomaly. The SA can analyze the received events to infer the exact event type or behavior pattern that caused the anomaly and can alert the supervisor of the finding through the GUI display.

The alert library can be easily developed since each type of alert message is assigned an alert code. This alert code is encoded into the PREDICT message so the SA can notify the supervisor of a particular type of anomaly. Upon receiving the PREDICT message, the GUI can extract the alert code from the PREDICT message and display the appropriate alert message according to the code. While the GUI maps alert codes to alert messages, the SA links the alert codes to various anomalous scenarios. Currently, alert code 0 is reserved for the case of no alert to be displayed.

For example, a possible alert can be "High Interaction Frequency" with alert code 1. The SA can trigger this alert on the GUI when a high number of clicks are detected. To trigger the alert, the SA encodes the alert code 1 into the new PREDICT message and sends it to the GUI. When the PREDICT message is parsed by the GUI, the GUI displays "High Interaction Frequency" on the Title and Alert Bar (Figure 3.4).

4.5 Summary

This chapter described in detail how the SA was designed and implemented to link HSMM results to the decision support visualization GUI. The disconnect between the actual HSMM results and the designer's graphical representation of anticipated results was pinpointed and addressed by the development of the abstraction layer. While the abstraction layer module remains as the heart of the server application, the connection module and the message dispatch module serve as supporting cast to allow timely and consistent communication between the SA and other components of the decision support system (RESCHU and GUI). In the next chapter, the implementation details of the GUI will be explained, and the GUI usability testing results will be presented.

5 Display Rendering and Usability Results

This chapter describes the techniques implemented to render the four regions of the GUI client (Figure 5.1): the Model Accuracy Prediction Panel, the Model Performance History Panel, the Interaction Frequency Panel, and the Title and Alert Bar. The GUI is implemented using the Java Swing library, while the Java Graphics API is extensively applied (Sun Microsystems, 2008). The GUI is designed to fit on a screen with resolution of 1280 pixels (px) by 1024px.

Figure 5.1: Screenshot of the GUI

5.1 Model Accuracy Prediction Panel

The Model Accuracy Panel is on the top right of the GUI (Figure 5.1), displaying trend lines representing future trends for the expected MAS, spanning the next 3 minutes. More specifically, the top trend line represents the expected MAS if the top three predictions match, and the bottom trend line represents the expected MAS if the bottom three

predictions match the actual occurred event, as discussed in Chapter 4. The shaded gray area between the two lines represents the uncertainty between the two (Figure 5.2).

It should be noted that the GUI designer anticipated both an upper and a lower bound of prediction accuracy, as well as the most probable prediction accuracy (Table 4.1, Requirement 5). However, the most probable prediction accuracy is the upper bound, by definition. As a result, the original design (Figure 4.6) was modified to include only the most probable prediction accuracy and the lower bound, cutting the number of trend lines from three to two.

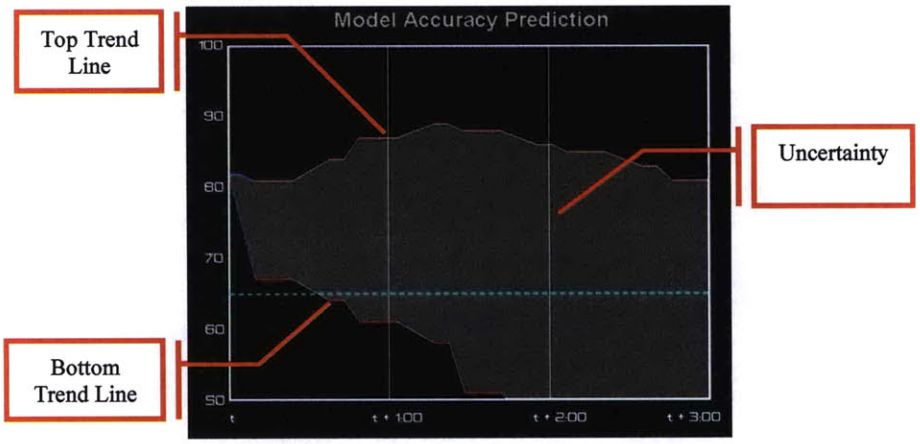


Figure 5.2: Model Accuracy Prediction Panel

The Model Accuracy Prediction Panel is implemented as a subclass of the standard Java *JComponent* class. The *paintComponent* method is automatically invoked by the *JComponent* class whenever the computer screen needs updating. It can also be triggered by other methods to redraw the display. Hence, this method is overridden to render the trend lines, which are updated every second.

The Model Accuracy Prediction Panel is designed to occupy an area of 630x460px on the screen, 540x400px of which are designated as the trend line area. Thus, the axes are drawn as a 540x400px rectangle with a thickness of 3px. Since the window displays three minutes (180 seconds) of predictions ranging from 50 to 100 points, one second spans the width of 3px (540px divided by 180), while a point occupies a height of 8px (400px divided by 50).

Trend lines are rendered from parsed PREDICT messages (Appendix C). As defined previously, a PRED clause in a PREDICT message contains information on expected duration (ED), expected MAS (EMAS), prediction confidence (PC), and lower bound (LB). Each PRED clause forms a single segment for both top and bottom trend line. The widths of both trend line segments are determined by ED. The height of the top trend line is determined by EMAS, while LB determines the height of the bottom trend line. PC determines the color. For example, a PC value below the high confidence threshold (set by the user) is represented in blue, a PC value above the low confidence threshold causes the color to turn red, and a PC value between the two threshold values sets the color to gray. However, further study is needed to investigate the appropriate values for the high and low threshold, which is beyond the scope of this research project.

Direct rendering of the HSMM results in a number of disconnected horizontal lines. However, an experiment showed that users are more confident in the model when the trend lines are smooth (Castonia, 2009). To create smooth trend lines from disconnected horizontal segments requires connecting the segments with diagonal lines. A graphical illustration is shown in Figure 5.3.

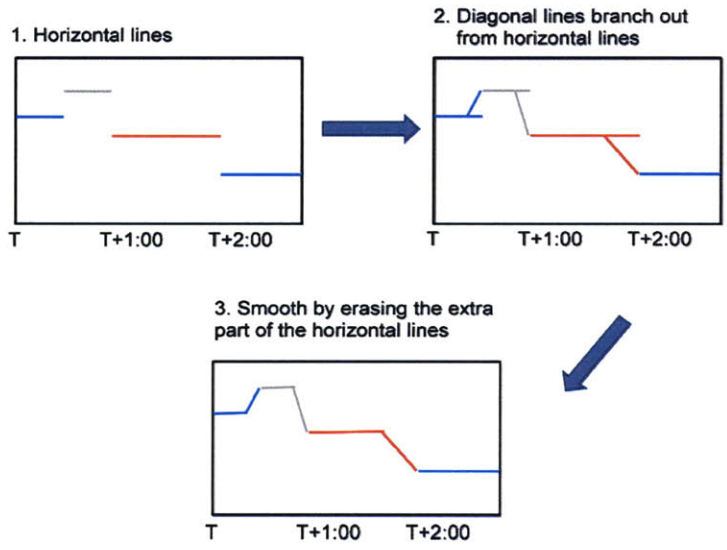


Figure 5.3: Trend line smoothing

The diagonal lines branch out at 70% width of the original horizontal lines, and 70% was chosen to balance the smoothness of the trend line and the truthfulness of the line's information.

The shaded area is filled every time both a top and bottom segment of the trend lines are drawn. The shaded area is composed of many translucent gray polygons bounded by the top and bottom segments and two vertical lines connecting the beginning points and the end points of the two segments (Figure 5.4).

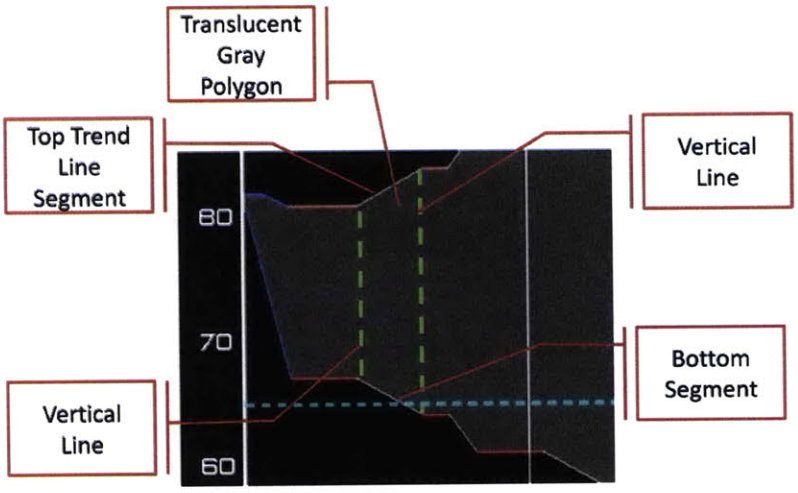


Figure 5.4: Rendering of shaded area of uncertainty

To summarize, to render the Model Accuracy Prediction Panel, a series of segments are first drawn along with the gray polygons from the parsed PREDICT message, and then the axes and labels are filled.

5.2 Model Performance History Panel

The Model Performance History Panel, the lower section of the decision support tool in Figure 5.1, is a subclass of *JPanel* class. The panel is designed to occupy 1280x300px of screen space on the bottom of the GUI display. It contains three *HistoryBar* objects, each occupying 1200px by 36px. *HistoryBar* is a subclass of *JComponent*, and similar to

Model Accuracy Prediction Panel, the *paintComponent* method is overridden to render the bar. Each bar is composed of a series of rectangles of different colors. Blue represents high model performance, while gray represents medium performance, and red signals low performance, given the previously discussed user-determined performance thresholds.

The colors were chosen to promote visual discrimination. These differ from the typical red-orange-green color-set in order to accommodate most color-blind personnel. Approximately 10% of the male population is color-blind, and 95% of all variations in human color vision involve the red and green receptor in human eyes (Gegenfurtner & Sharpe, 2001). Therefore, the significant number of people affected makes it necessary for the interface's color scheme to be color-blind friendly, and the blue-gray-red color-scheme was specifically chosen so as to be accessible to red-green color-blind personnel who represent the vast majority of color-vision deficient personnel. Figure 5.5 shows how the interface would be seen by red-green color-blind personnel (Gegenfurtner & Sharpe, 2001).

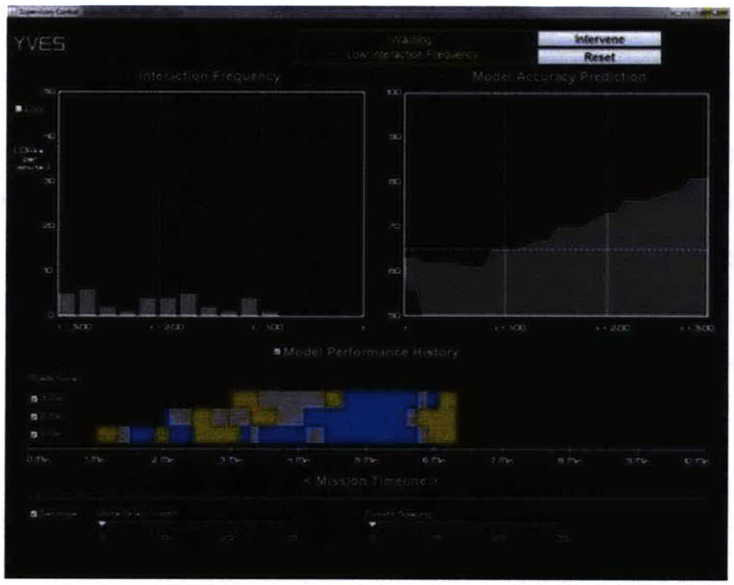


Figure 5.5: The GUI as seen by color-blind personnel (Dougherty & Wade, 2009)

As detailed in Chapter 3, the bottom three bars visualize the historical performance of the model's predictions at 1, 2, and 3 minutes into the future, as shown in Figure 5.6.

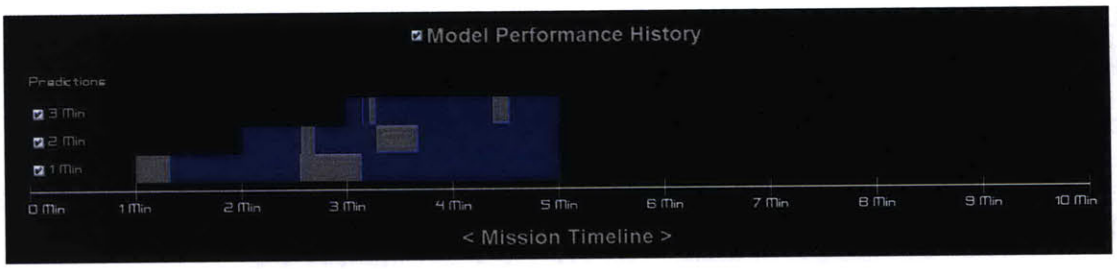


Figure 5.6: Model Performance History Panel

The history bar for the one-minute prediction is rendered black before the 1 minute mark on the mission timeline because the mission begins at time 0, and prediction history cannot begin until 1 minute into the operation. Similarly, the history bars for two and three-minute predictions are black before the 2 and 3 minute marks on the time line, respectively.

The three bars are updated per second simultaneously. The GUI records the EMAS values at the 1, 2, and 3-minute intervals when rendering the Model Accuracy Prediction Panel. At time T, the 1-minute EMAS value is recorded and tagged with timestamp T+60s, while 2 and 3-minute values are tagged with T+120s and T+180s respectively. These values are compared to the MAS value when the time matches their timestamps. The comparison results determine the color of the rectangles, and at each second, a 2x36px rectangle of the designated color is drawn and added to the end of each history bar. The width of the new rectangle is 2px because that is the width of 1 second on the mission timeline (1200px divided by 600 seconds).

The absolute difference between the predicted EMAS value and the actual MAS value determines the color of the rectangle. For example, a difference below the high confidence threshold sets the color blue, a difference above low confidence threshold sets the color red, and a difference in between the two thresholds sets the color gray. Figures 5.7 and 5.8 illustrate how different threshold values affect the color rendering of the bars.

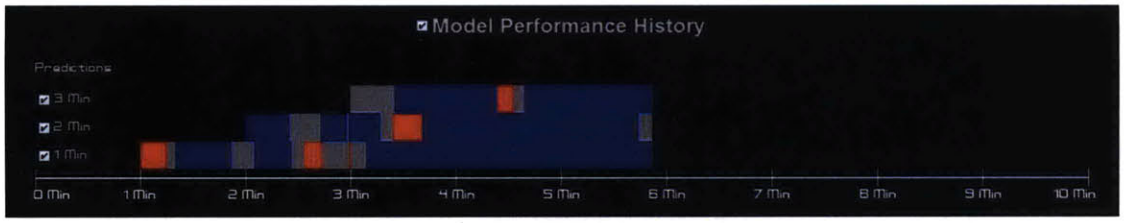


Figure 5.7: Lower threshold values as compared to Figure 5.6

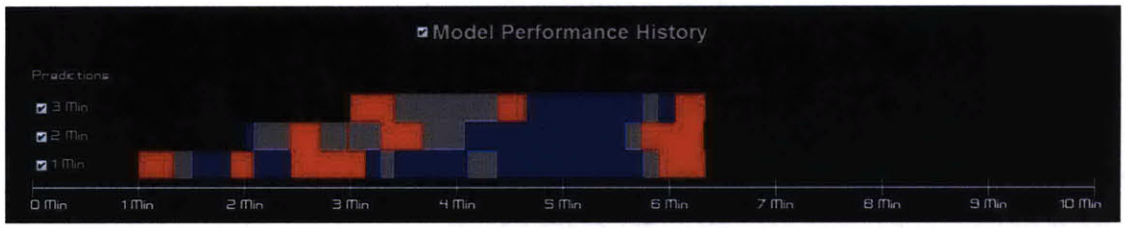


Figure 5.8: Even lower threshold values render more red and gray rectangles

5.3 Interaction Frequency Panel

The Interaction Frequency Panel, the upper left panel in Figure 5.1, displays the interaction frequency of the operator and RESCHU for the past three minutes. The panel occupies 630x460px of screen space, and displays up to 15 histogram-style bars, each representing the total number of mouse clicks within a 12-second interval. The height of each bar is determined by the number of clicks, where one click amounts to 8px. The width of each bar is 30px. It is less than 36px, which is the width of 12 seconds, because it allows for spacing between bars, as shown in Figure 5.9.

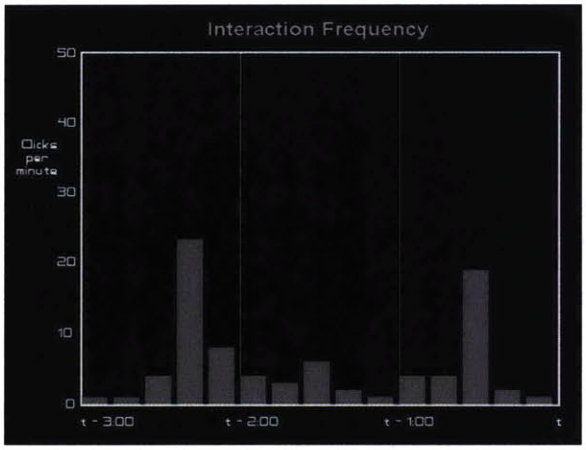


Figure 5.9: Interaction Frequency Panel

In order to render this window, the GUI client aggregates all CLK messages received and sums up the total number of messages for each 12-second interval beginning at 3 minutes prior to the current time. The timestamp included in each CLK message indicates the timing of the click. This window is updated every 12 seconds to keep consistent with the 12-second interval.

5.4 Title and Alert Bar

The Title and Alert Bar displays the identity of the operator being monitored and alerts the supervisor of possible operator failures, as shown in Figure 5.10.

Figure 5.10: Title and Alert Bar

The Title and Alert Bar resides on the top portion of the GUI client and occupies 1280x70px of screen space. At the beginning of each RESCHU operation, the SA receives a RBEG message and in turn, sends the GUI a SBEG message to signal the beginning of a session. Upon receiving the SBEG message, the GUI client displays the operator's username on the left side of the Title and Alert Bar. As shown in Figure 5.10, the operator being monitored is Yves.

As stated previously, the HEADER of each PREDICT message also encodes the type of alert message to be displayed. While 0 indicates no alert, other numbers can trigger different alerts. In the example shown in Figure 5.10, a Low Interaction Frequency alert is triggered and displayed in the center portion of the Title and Alert Bar. The two buttons on the left of the alert message allow the supervisor to acknowledge the alert message in two ways. The Intervene button signals the GUI that the supervisor may wish to take some action as a result of the predictions. The Reset button signals the GUI that the supervisor sees the message but has no plan to communicate with an operator in the

near term. Both buttons erase the message from the screen, while the Reset button allows the message to reappear after some delay, preset by the supervisor.

5.5 Usability Testing Results

A pilot study was conducted to evaluate the usability of the system. A total of 16 subjects were asked to evaluate various aspects of the decision support display during and after a 10-minute testing session. During the testing session, the subjects were presented with a 10-minute long video clip of a RESCHU instance along side an instance of the DSV display, both connected to the SA, while a simulated operator interacted with RESCHU. The video clip was displayed across two computer monitors.

Four alert messages of various content were triggered and displayed in the video clip, and the subjects were asked to make hypothetical decisions either to intervene with the operator or to ignore the alert. Testing showed that subjects made decisions based on both operator's actions in the RESCHU interface and the information presented on the decision support display. Subjects were able to make the correct decision 60% of the time in terms of whether intervention was needed.

Subjective comments included a few complaints about the layout of the DSV display, and some subjects did not understand the Model Performance History Panel nor the Model Accuracy Prediction Panel. Some subjects also reported that they ignored the DSV display most of the time and focused only on RESCHU. As a result, the alert messages were easily missed by these subjects. It is recommended that the designer re-design the layout of the alert message area so as to increase the visability of the alert messages. The color-scheme of the confidence level representation also seemed vague to some subjects. These design recommendations are further discussed in Chapter 7.

No technical issues were revealed during testing, and subjects did not notice any lag caused by system latency. Moreover, the system worked smoothly as a whole, and the

pilot testing proved that the concept of an HSMM-based decision support tool is feasible, albeit with further refinements.

5.6 Summary

This chapter discussed how each region of the DSV display is populated. Detailed implementation specifications of the DSV display were explained. Usability testing results from a pilot study were also presented and discussed. The next chapter will describe the system testing results and the computational enhancements implemented to allow the system to work smoothly.

6 Software Testing Results and Discussion

This chapter documents the tests conducted to explore the boundary conditions of the server application. The prediction-generating (PG) algorithm creates latencies, and several parameters of the algorithm can be adjusted to increase or decrease the latency of the system. Software testing was conducted to explore the upper limit for each parameter while keeping system latency under 500ms. This chapter also discusses various performance enhancements put in place to increase the speed of the system.

6.1 Testing Results

As previously stated, latency of the system is an important factor that governs both usability and scalability of the system, and thus, it was a major area of testing. Adjustable system parameters that affect latency for a single RESCHU operator are:

- Length of prediction sequence generated
- Length of event sequence input for the PG algorithm (or Event Window Width (EWW))

The length of prediction sequence affects the latency of the system because the longer the sequence of predictions, the more prediction steps need to be generated, and thus more calculations are needed. However, as discussed in section 4.3.4, a minimum length of 210 seconds is required by the system to ensure the DSV trend graph remains populated for enough time.

The length of event sequence also affects the latency of the system because the longer the event sequence, more calculations are needed to compute the probability of the sequence, and thus, more time is spent by the PG algorithm. Since the PG algorithm is based on the Viterbi algorithm, the run-time of the PG algorithm is mainly governed by the run-time of the Viterbi algorithm. Moreover, the run-time of the Viterbi algorithm is $O(NK^2T^2)$, where N is the length of the event sequence, K is the number of hidden states for the HSMM, and T is the maximum duration of the hidden states (Mitchell, 1995). For a

particular HSMM, where K and T are both fixed, the Viterbi algorithm run-time scales linearly with the length of the input event sequence, which is determined by the Event Window Width (EWW). Thus, testing validates whether the PG algorithm preserves the linearity property of the Viterbi algorithm.

When multiple user-event messages reach the SA one immediately following another, they are processed sequentially (order is determined by the TCP/IP networking layer) by the PG algorithm to generate predictions. Moreover, processing a single message results in latency, and the overall maximum system latency is the sum of the latencies from processing multiple user-event messages. Thus, the number of operators monitored determines the overall maximum system latency. Hence, tests conducted include:

1) Single operator latency vs. length of prediction sequence generated
2) Single operator latency vs. EWW
3) Maximum overall system latency vs. number of operators monitored

Testing was conducted using computers with identical hardware specifications running Windows Vista 64-bit operating system and Java Runtime Environment version 1.6.0 update 14. Hardware specifications of the computers are listed below.

- Intel Core™ 2 Duo CPU E6750 @ 2.66GHz (over-clocked to 3.00GHz)
- 4GB DDR2 RAM at 800MHz
- 1TB 7200rpm SATA Hard Drive

6.1.1 Single Operator Latency vs. EWW

The HSMM can provide the most likely future events predicated on a sequence of previous events. The EWW determines the length of the sequence of events from which the inferences of future states are made. A longer sequence of events requires more computation to calculate the probability of that sequence of events. Testing was conducted to investigate how system latency varies with the EWW when a single RESCHU operator is under supervision.

For the RESCHU test bed used in this experiment, a testing session lasted 10 minutes. When testing, the length of prediction sequence was kept constant at 210 seconds, which is the required minimum temporal length of prediction sequence, while the EWW varied. Figure 6.2 shows a corresponding graph to the testing result.

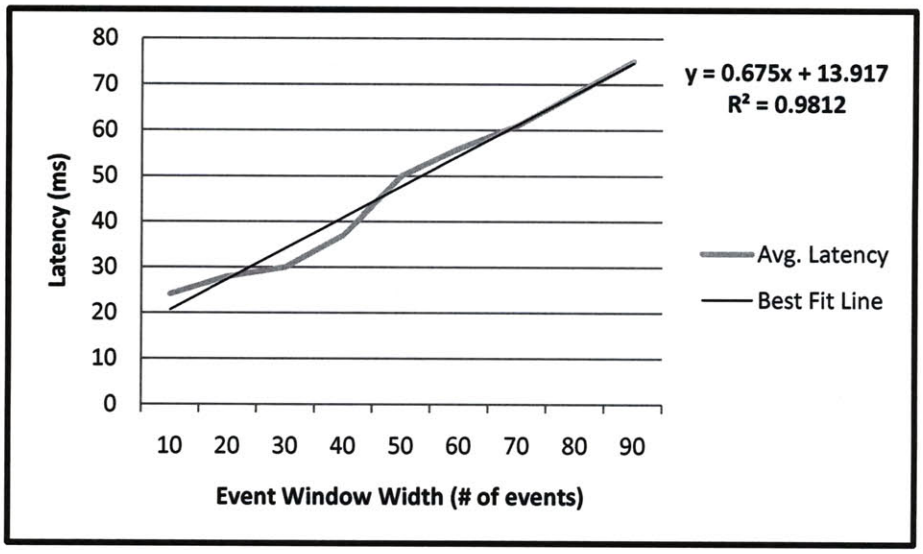


Figure 6.1: EWW vs. single operator latency

Results show that the latency is monotonically increasing with the EWW. A linear best fit line was generated with R^2 greater than 0.98, and the linear coefficient is significant ($p < 0.001$). Moreover, when the window width is 50, the average latency reaches 50 ms for a single RESCHU operator. Fifty milliseconds was previously shown to be the maximum allowable lag time for a system designed to support up to 10 operators. Hence, the size of the largest window the system can handle is 50 events and was set to be the default EWW for this project.

6.1.2 Single Operator Latency vs. Length of Prediction Sequence

The Model Accuracy Prediction panel displays predictions up to three minutes into the future. As a result, at least a 180-second long sequence of predictions needs to be generated each time a new event occurs. Moreover, since each prediction is time-

sensitive, the panel updates every second by shifting the trend lines to the left by one second. Hence, a 1-second part of the prediction sequence becomes invalid every second and disappears off the screen (Figure 4.9).

To ensure the trend graph remains fully populated for a reasonable period of time, a prediction sequence of longer than 180 seconds must be generated each time. As discussed in section 4.3.4, a minimum length of 210 seconds is required for all generated sequences of predictions because a buffer of 30 seconds is needed to ensure the trend line region stays populated throughout operation. However, generating a longer sequence increases single operator latency. Hence, tests were conducted to investigate how latency varies with the length of prediction sequence. During testing, the EWW was fixed at 50 (see section 6.1.1), while the prediction sequence length varied. Testing results are presented in Figure 6.1.

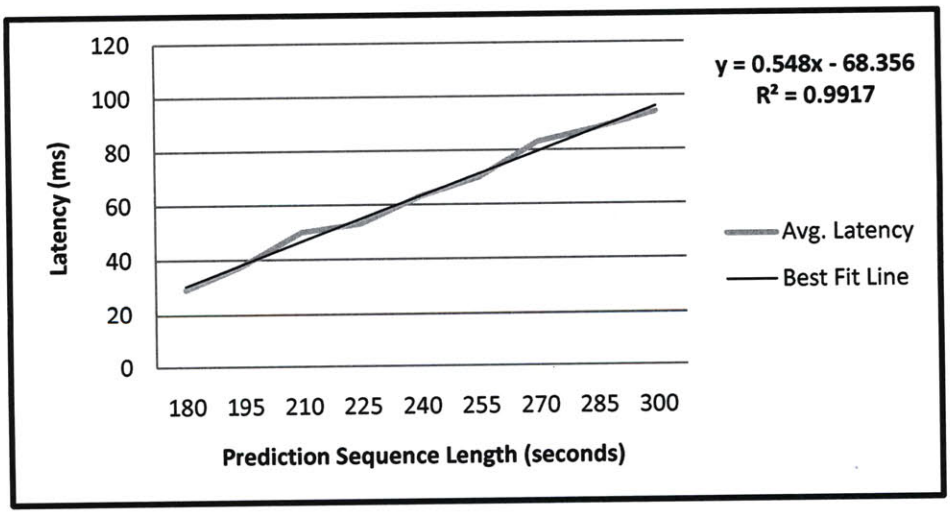


Figure 6.2: Prediction sequence length vs. single operator latency

Testing results show that latency increases linearly with the length of prediction sequence (Figure 6.1). Moreover, when the length of prediction sequence is 210 seconds, the single operator latency reaches 50ms. A linear best fit line was generated with R^2 greater than 0.99, and the linear coefficient is significant ($p < 0.001$).

6.1.3 Overall System Latency vs. Number of Operators Monitored

The interface designer envisioned that the decision support visualization discussed previously would assist supervisors to monitor a team of maximally 10 operators simultaneously. Testing was performed to investigate how the overall system latency scales with each additional operator monitored.

During testing, the EWW was set to 50, while length of prediction sequence was set to 210 seconds. The system test evaluated system performance under the worst case scenario by sending N (determined by number of operators monitored) user-event messages to the SA at random time intervals over the RESCHU session course of 10 minutes (Poisson process with lambda of 12 seconds, which is the average interaction time interval). The system was tested with the number of operators ranging from 1 to 15. Figure 6.3 presents the testing results.

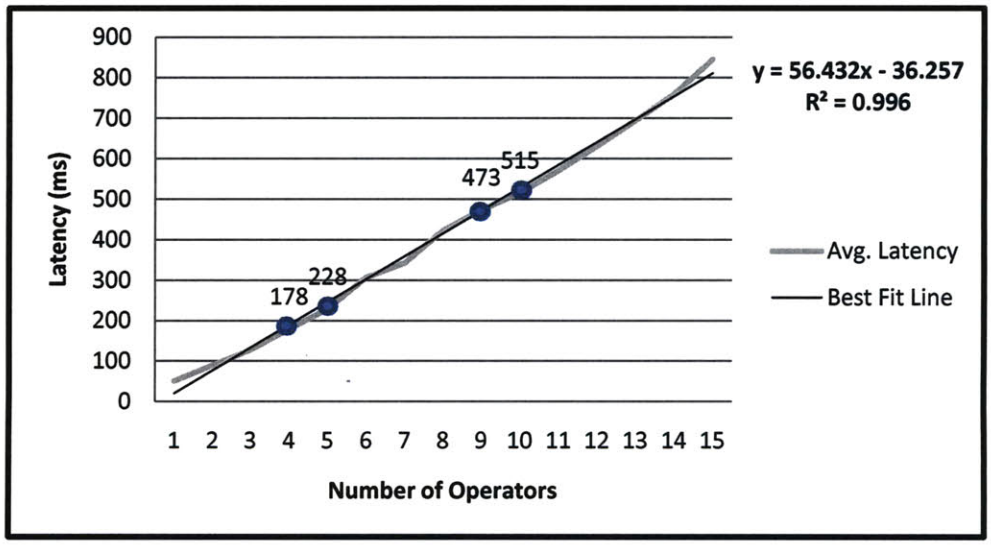


Figure 6.3: Number of operators vs. system latency

Results show that system latency increases linearly with the number of operators monitored and that when 10 operators are under simultaneous supervision, average system latency begins to exceed 500 ms slightly (Figure 6.3). A linear best fit line was generated with R^2 greater than 0.99, and the linear coefficient is significant ($p < 0.001$).

Due to the time-critical nature of UV operations, the system needs to be conservative in terms of allowing latency. Thus, the maximum number of operators supported by the representative RESCHU system should be reduced to 9 given the 500ms requirement. However, while 500ms is an acceptable latency to most people (Claypool, 2005), other research has shown that humans prefer to see latencies less than 200ms (Card, Moran, & Newell, 1983). Should this lower latency be required, the maximum number of operators supported by a single server application may need to be reduced to 4. In this situation, one possible way to support up to 10 operators simultaneously would be to have multiple SAs working in parallel in order to share and balance the computational load. Although this feature is currently not implemented, the modular nature of the system architecture supports such modification with minimal effort.

6.1.4 Testing Results Summary

The testing results provide insight on how single operator latency and overall system latency vary with different system parameters. As stated in the original requirements, an overall system latency of less than 500ms must be achieved to ensure system responsiveness. Moreover, other research suggests that it might be more favorable to keep the system latency under 200ms at all times. This limits the maximum number of operators monitored by a single server application to 4, given that the EWW is 50 and length of prediction sequence is 210 seconds.

However, future system designers have the freedom to make trade-offs between length of prediction sequence and the EWW. A Pareto front of the system parameters was generated to investigate how a future system designer could choose a different set of system parameter values so as to minimize system latency while maximizing the number of operators supported (Figure 6.4). Data points were generated by varying the EWW from 30 events to 70 events with 10-event intervals, the length of prediction sequence from 180 seconds to 300 seconds with 30-second intervals, and the number of operators from 2 to 12. The y-axis denotes latency, and the x-axis denotes the inverse of the

number of operators so that the maximum number of operators is on the left-most end of the axis in order to populate a classical view of the Pareto front graph.

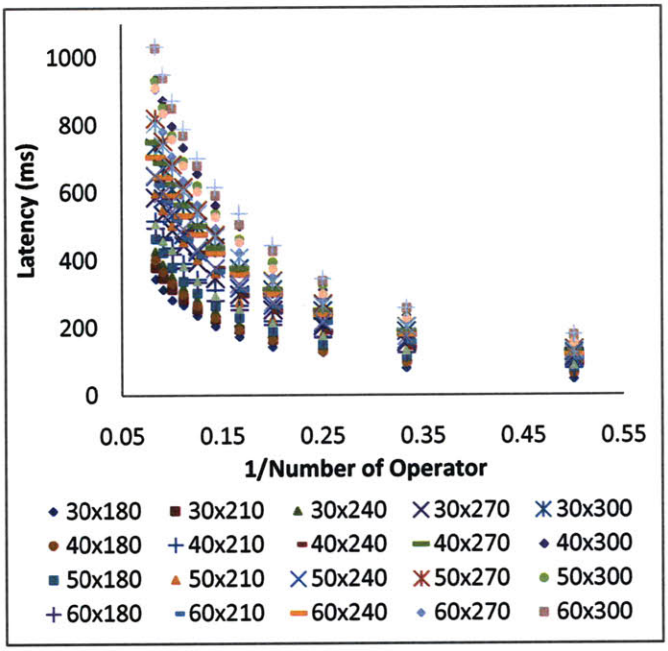


Figure 6.4: Pareto front graph of the system parameters

The left-most data points represent 12 operators, while the right-most data points represent 2 operators monitored. Figure 6.5 shows a magnified view of the Pareto front.

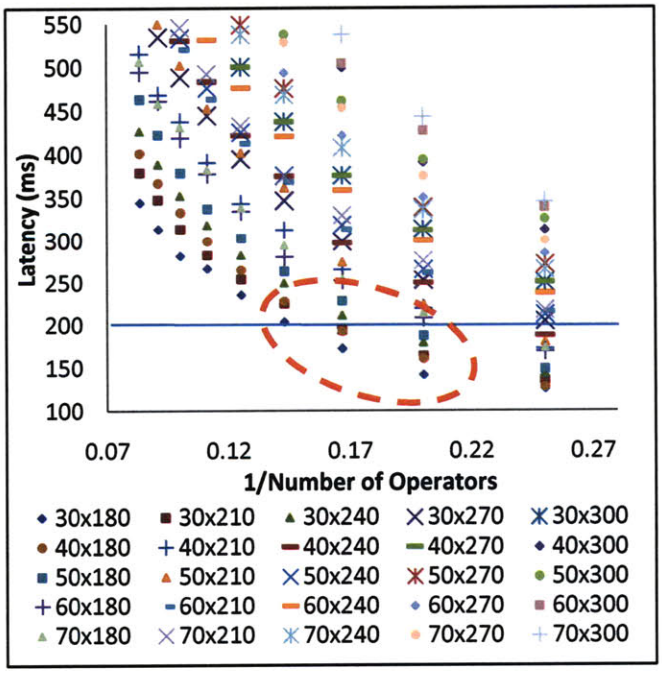


Figure 6.5: Magnified view of Pareto front

Given a maximum latency limit of 200ms, the current system can support up to a maximum number of 6 operators by lowering the EWW and the length of prediction sequence. Moreover, the set of Pareto-efficient system parameters is listed in Table 6.1.

Table 6.1: Pareto-efficient system parameters

EWW	Length of Prediction Sequence (s)	Operators Supported	Latency (ms)
30	180	7	204
30	210	6	194
40	180	6	193
30	180	6	173
50	180	5	188
30	240	5	180
30	210	5	164
40	180	5	161
30	180	5	142

The set of Pareto-efficient system parameters provides insights for the future system designers on how system parameters can be varied in order to increase the maximum number of operators supported while keeping latency acceptable. However, if a system designer wishes to keep EWW at 50 events and the length of prediction sequence at 210 seconds, then future work is needed in order to support more than 4 operators. For example, the server application can be improved so as to allow multiple server applications working together, each monitoring up to 4 operators independently (given a system similar to RESCHU).

6.2 Computational Enhancements

Over the course of the project, a number of computational issues emerged and were resolved during the development phase of the PG algorithm. This section documents these issues and the corresponding solutions.

As stated in Chapter 4, the rigorous prediction generation method computes the probability of an event sequence with the *logProbability* function 19 times for each prediction step. In addition, the top five most probable events out of the 19 are chosen as the five only candidates for the next expected event, and their probabilities are normalized to sum up to one. Two issues emerged during development: 1) the *logProbability* function is computationally intensive, and 2) renormalizing small log probabilities directly is challenging in Java.

6.2.1 Fast computation of the probability of sequences

The computational enhancement that had the largest impact on the system is the implementation of a faster variant of the method used to compute the probability of a sequence, *logProbability.* A new function was created, *logProbability2,* in order to achieve a significant speed increase as compared to the original function. The original *logProbability* function takes a sequence of events as input and calculates the log probability of the sequence. To calculate the probability of observing each of the 19 events as the next event, this function must be called 19 times. The new variation of *logProbability* takes a sequence of events and calculates the 19 log probabilities of seeing all 19 event types as the next event. This variation eliminates redundancy in invoking the original *logProbability* method 19 times on inputs that are, for the most part, identical (Figure 6.5).

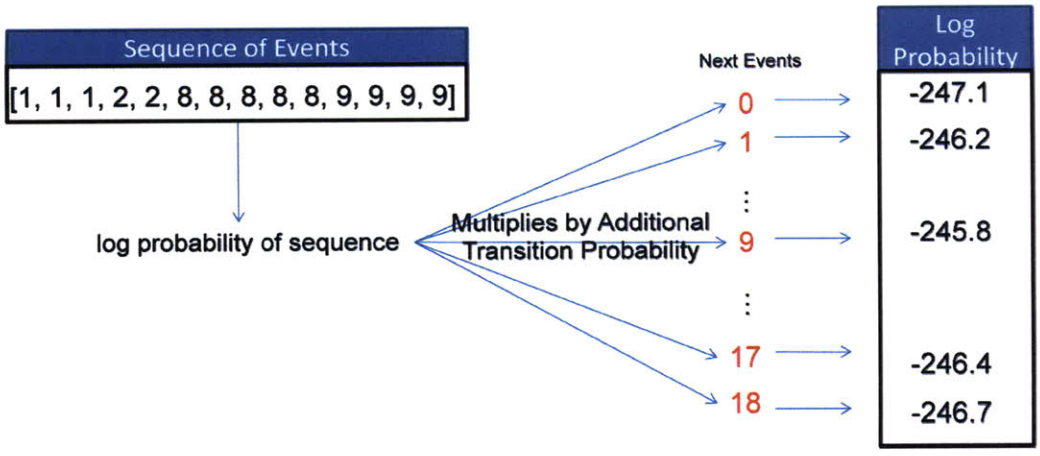


Figure 6.6: New variation of logProbability

The new variation of *logProbability* calculates the probabilities in two steps:

1. Calculate the probability of the event sequence
2. Calculate the additional transitional probability from the sequence to the next event for all 19 event types.

As previously discussed, the Viterbi algorithm has run-time $O(N)$ given a particular HSMM. Previously, the PG algorithm invokes the original *logProbability* 19 times, which is based on the Viterbi algorithm. Given an input sequence of length, N, the total time to compute the log probability of the input sequence plus the next event is $N + 1$ steps. Therefore, the total time spent in computation for the PG algorithm per step of prediction is 19 $(N + 1)$ steps. In the new variation of *logProbability*, the log probability of the input sequence is first computed in N steps. Then, 19 lookups are performed to calculate the log probabilities of the 19 candidates of next event all at once. Assuming each lookup has computation cost c, the total time spent to compute the log probabilities of the 19 sequences of input sequence plus next event is $N + 19c$. In addition, since c is the time spent in looking up the probability for a single event instead of for a sequence of events, c is negligible compared to N. Moreover, overhead between each subsequent call of *logProbability* was avoided, and thus, the new method allows the prediction generation process to run at least 2034% faster than the original method (Table 6.1).

Table 6.2: logProbability vs. logProbability2

Algorithm Runtime	**Using *logProbability***	**Using *logProbability2***	**Performance Gain**
Maximum (ms)	1546	76	2034%
Average (ms)	1124	50	2248%

6.2.2 Normalizing log probabilities

When generating predictions, the algorithm records the top five most probable events as the candidates for the next event. The rationale behind this procedure is that the top five

most probable events in combination cover more than 99% of the total probabilities of the 19 events. Since only 5 out of the 19 events are selected as the exclusive candidates for next event, their probabilities should also be normalized to sum up to one. The normalized probability of an event is calculated as its probability divided by the sum of the five probabilities, given in (4).

$$Normalize(P_i) = \frac{P_i}{\sum_i P_i} \tag{4}$$

The probabilities output by *logProbability2* are log probabilities, and in order to normalize these probabilities, they must be converted into natural probabilities. However, these probabilities are stored in log probability form because they may be too small for Java, and converting them will result in zeroes. To resolve these issues, the normalization procedure relies on simple mathematical rules listed below.

$$Log(xy) = Log(x) + Log(y) \tag{5}$$

$$\frac{b}{c} = \frac{a \times b}{a \times c} \tag{6}$$

These rules allow the normalization procedure to increase the log probability values by a common factor before normalizing them, which is actually multiplying the probabilities by a common factor. Once these log probability values are large enough for Java, they are normalized using (4). The common factor is determined by the integer value of the absolute value of the largest log probability value. The common factor ensures that at least one log probability will be large enough for Java after adding the common factor.

The following proof shows that the distribution of the probabilities is preserved after applying the common factor.

Proof:

Let x_i be the probability of a sequence, and let $log(a)$ be the constant common factor

$$We\ want\ to\ show\ that\ \frac{e^{log\,(x_i)+log\,(a)}}{\sum_i e^{log\,(x_i)+log\,(a)}} = \frac{x_i}{\sum_i x_i} = Normalize(x_i)$$

$$\log(x_i) + \log(a) = \log(\, ax_i) \tag{7}$$

$$e^{log\,(x_i)+log\,(a)} = e^{\log(\,ax_i)} = ax_i \tag{8}$$

$$\frac{e^{log\,(x_i)+log\,(a)}}{\sum_i e^{log\,(x_i)+log\,(a)}} = \frac{ax_i}{\sum_i ax_i} = \frac{ax_i}{a \times \sum_i x_i} = \frac{x_i}{\sum_i x_i} \tag{9}$$

$$Thus, \frac{e^{log\,(x_i)+log\,(a)}}{\sum_i e^{log\,(x_i)+log\,(a)}} = \frac{x_i}{\sum_i x_i} = Normalize(x_i), \qquad Q.E.D. \tag{10}$$

For example, consider five events of the following log probability: [-1245.9, -1246.57, -1247.5, -1248.03, -1249.17]. The common factor is 1245. Table 6.2 shows the step-by-step calculations for normalizing these log probabilities.

Table 6.3: Intermediate results of normalization of a typical case

Event Rank	Log Probability	After adding common factor	Probability value in Java	Normalized
1	-1245.9	-0.9	0.406569	0.53
2	-2046.57	-1.57	0.208045	0.27
3	-2047.5	-2.5	0.082085	0.11
4	-2048.03	-3.03	0.048316	0.06
5	-2049.17	-4.17	0.015452	0.03

6.3 Summary

System testing results showed that system latency scales linearly with system parameters of Event Window Width (EWW) and length of prediction sequence, as well as the number of operators monitored. The Pareto front of the system parameters showed that future system designers have the freedom to select other values for the system parameters to increase the maximum supported number of operators, while keeping latency low. Given the 500ms latency requirement and the default system parameter setting (EWW = 50 and length of prediction sequence at 210 seconds), up to a maximum of 9 operators can be supported for the given RESCHU system. However, the maximum latency

requirement might need to be reduced to 200ms according to other research, in which case, a maximum of 4 operators may be supported per server application. Computational enhancements that produced performance gains were also discussed.

7 Conclusion

This project connected a hidden semi-Markov model that predicts real-time operator behavior in a command and control setting with a graphical decision support visualization display. The resulting proof-of-concept system operates in real-time, providing a team supervisor with behavior predictions for up to 4 UV operators simultaneously with maximum latency of 200ms, and possibly up to 9 users if the latency requirement is relaxed to 500ms. The heart of the system is a server-side application. It houses a connection module to connect the application to RESCHU instances and the DSV display, a message dispatch module that sends and receives messages, and an abstraction layer module that transforms HSMM prediction results into data readable by the DSV display. Through the use of this system, it is envisioned that supervisors could improve their performances in monitoring teams of operators.

Although the system implemented in this project is not mature enough for real-world UV operation, this research provides valuable insight into potential difficulties that might arise during future related research and development; especially in situations where the output of complex algorithms must be interpreted under operational time pressure by non-technical personnel. Moreover, the research reveals the important disconnect between actual mathematical formulations and an interface designer's anticipated output. As the implementation of the current interface resulted in a set of new design recommendations, this work further emphasizes the need for a feedback mechanism between the design of the interface and the operation of the underlying model.

While the current system enables future research on the subject of using HSMMs to aid supervisors of supervisory control operators in real-time, there remain many areas for future works of improvements.

7.1 Design Recommendation

Two main issues involving the interface design were raised both during the development phase of the project and after the pilot study for system usability. They are discussed below.

7.1.1 Interface Labeling

As discussed in Chapter 4, the output of the HSMM cannot be directly displayed on the trend display in its original design. An abstraction layer was created to transform the output of the HSMM into data that can be rendered as trend lines. As a result, a model accuracy scoring metric was devised to perform the transformation, which was not part of the original intended design. This metric assigns various scores to the model according to various level of its prediction performance as documented in Chapter 4. As a result, the trend display no longer presents future predictions on the accuracy of the model, but rather the projected performance score of the model. Thus, it is recommended that the designer change the title of the trend display from Model Accuracy Prediction to Performance Forecast. Moreover, the designer should assess whether this change fundamentally changes supervisor understanding of the graph.

7.1.2 Interface Layout

As suggested by several subjects during the pilot study, both the intervene button and reset button (Figure 5.10) should be larger. Subjects also suggested that the alert message area should be resized to be larger and needed a change of color scheme to make the messages more salient. Audio alerts should also be investigated.

7.2 Future Work

There remains room for improvement for several aspects of the server application: 1) the model accuracy scoring metric could be dynamic, and 2) the alert library should be implemented to complete the system.

7.2.1 Model Accuracy Scoring Metric

The model accuracy scoring metric currently considers only the five most probable events regardless of their probability distribution. Although the top five events cover more than 99% of the probability distribution of all of the 19 events most of the time, there are still instances where less than or more than five events should be considered. Thus, one way to improve the model accuracy metric would be to make it dynamically adjust the number of events considered. For example, when the top two events have probabilities 0.65 and 0.34, while the remaining 0.01 is evenly distributed among the remaining 17 events, it may be more accurate to consider only the top two events. However, when the top seven events are almost equally probable with probabilities of around 0.14, then all seven events should be considered as candidates for next event instead of just five.

As the metric changes from static to dynamic, the scoring of each rank of event should also change. Currently, the ranks receive 50, 45, 40, 20, and 10 in order. A new scoring system should be devised for the dynamic metric, while keeping the heuristic clustering property of the current scoring system in play. The changes would necessitate further human-in-the-loop testing.

7.2.2 Alert Library

An important feature of the system is its ability to display alerts to the supervisor. The system is designed to display different alerts for different scenarios. Currently, the system triggers an alert and sends the alert code in the PREDICT message, which then sends an alert message to the GUI. However, only a default alert message and its corresponding alert code are used by the SA. The library of the alert messages should be implemented by the interface designer and the model builder to assess the full span of possible problematic states. Moreover, the server application should also be augmented to detect the exact event that triggers an alert.

The thresholds for triggering alerts can be set by the user. Further research conducted by the HSMM designer and the interface designer should assess possible optimal threshold schemes (set either by humans or automated agents) in order to alert the supervisor of the exact cause for potential failures when anomalies arise.

7.3 Summary

Advances in technology will eventually allow a team of operators to manage fleets of multiple heterogeneous UVs in a supervisory control setting (Cummings et al., 2007). Under such scenarios, it is important that the team supervisor is provided with the most updated status of each operator's performance, as well as possible predictions of anomalous behaviors, so that costly mistakes can be avoided. This research explored the possible implementation of such a decision tool powered by hidden semi-Markov models, and successfully developed a proof-of-concept prototype. This project enables further researches in the area of real-time supervisory control decision support, specifically for unmanned vehicle systems. The project's major contribution is connecting an HSMM with a decision support display in real-time via a server application with an embedded abstraction hierarchy, which with more research, may one day be perfected to aid supervisors in actual UV operations.

References

Alberts, D. S., Garstka, J. J., & Stein, F. P. (2000). *Network Centric Warfare: Developing and Leveraging Information Superiority* (2nd ed.). Washington, DC: Command and Control Research Program (CCRP).

Allanach, J., Tu, H., Singh, S., Willett, P., & Pattipati, K. (2004). Detecting, Tracking, and Counteracting Terrorist Networks via Hidden Markov Models. *IEEE Aerospace Conference Proceedings*, 3246-3257.

Baum, L. W., & Petrie, T. (1966). Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics*, 1554-1563.

Boussemart, Y., & Cummings, M. L. (2008). *Behavioral Recognition and Prediction of an Operator Supervising Multiple Heterogeneous Unmanned Vehicles.* Paper presented at the Humans Operating Unmanned Systems, HUMOUS'08, Brest, France.

Boussemart, Y., Fargeas, J. L., Cummings, M. L., & Roy, N. (2009). Comparing Learning Techinques for Hidden Markov Models of Human Supervisory Control Behavior. *AIAA Infotech*, 1-5.

Bulla, J., & Bulla, I. (2006). Stylized Facts of Financial Time Series and Hidden Semi-Markov Models. *Computational Statistics & Data Analysis, 51*(4), 2192-2209.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Castonia, R. (2009). *The Design of a HSMM-based Operator State Monitoring Display* (No. HAL2009-04). Cambridge: MIT.

Claypool, M. (2005). The effect of latency on user performance in Real-Time Strategy games. *Computer Networks, 49*, 52-70.

Cummings, M. L., Bruni, S., Mercier, S., & Mitchell, P. J. (2007). Automation Architecture for Single Operator Multiple UAV Command and Control. *The International Command and Control Journal*, 1-24.

Davis, S. F. (2003). *Handbook of research methods in experimental psychology.* Malden, MA: Blackwell Publishing.

Dong, M., & He, D. (2007). A Segmental Hidden Semi-Markov Model (HSMM)-based Diagnostics and Prognostics Framework and Methodology. *Mechanical Systems and Signal Processing, 21*(5), 2248-2266.

Dougherty, B., & Wade, A. (2009). *Vischeck.* Retrieved 07/24/2009, from www.vischeck.com

Gegenfurtner, K. R., & Sharpe, L. T. (2001). *Color vision: from genes to perception*: Cambridge University Press.

Guedon, Y. (2003). Estimating Hidden Semi-Markov Chains From Discrete Sequences *Journal of Computational & Graphical Statistics, 12*(3), 604-639.

Guerlain, S., Jamieson, G., Bullemer, P., & Blair, R. (2002). The MPC Elucidator: A case study in the design of representational aids. *IEEE Journal of Systems, Man, and Cybernetics, 32*(1), 25-40.

Hart, D. (1995). *Air Traffic Control Center, Longmont, Colorado*. Retrieved 07/22/2009, from http://ails.arc.nasa.gov/images/newimages/jpegs/lowres/AC95-0408-2.jpg

Heckert, A., & Filliben, J. J. (2003, 06/01/2003). *Normal PDF*. Retrieved 07/25/2009, from http://www.itl.nist.gov/div898/handbook/eda/section3/eda364.htm

Hieronymus, J. L., McKelvie, D., & McInnes, F. (1992). Use of Accoustic Sentence Level and Lexical Stress in HSMM Speech Recognition. *Accoustics, Speech, and Signal Processing, 1*, 225-227.

Mitchell, C. H., M; Jamieson, L. (1995). On the complexity of explicit duration HMMs. *IEEE Transactions on Speech and Audio Processing, 3*(3), 213-217.

Nehme, C., Crandall, J. W., & Cummings, M. L. (2008). *Using Discrete Event Simulation to Model Situational Awareness of Unmanned-Vehicle Operators.* Paper presented at the Modeling, Analysis and Simulation Center Capstone Conference, Norfolk, VA.

Parlos, A. G., Rais, O. T., & Atiya, A. F. (2000). Multi-step-ahead Prediction Using Dynamic Recurrent Neural Networks. *Neural Networks, 13*(7), 765-786.

Rabiner, L., & Juang, B. (1986). An introduction to hidden Markov models. *ASSP Magazine*, 4-16.

Shiqing, W., & Dawei, M. (2004). A Market Forecast Algorithm Based on Bayesian Network. *Journal-ZhengZhou University Natural Science Edition*, 9-12.

Singh, S., Tu, H., Donat, W., Pattipati, K., & Willet, P. (1996). Anomaly Detection via Feature-Aided Tracking and Hidden Markov Models. *IEEE Transactions on Systems, Man, and Cybernetics*.

Sun Microsystems, Inc. (2008). *Java™ Platform, Standard Edition 6 API Specification*. Retrieved 07/21/2009, from http://java.sun.com/javase/6/docs/api/

Sykacek, P., & Roberts, S. (2002). Bayesian Time Series Classification. *Advances in Neural Information Processing Systems, 2*(14), 937-944.

Tversky, A., & Kahneman, D. (1974). Judgment under Uncertainty: Heuristics and Biases. *Science, 185*(4157), 1124-1131.

Tversky, A., & Kahneman, D. (1981). The Framing of Decisions and the Psychology of Choice. *Science*, 453-458.

Ware, C. (2000). *Information Visualization: Perception for design*. San Francisco: Morgan Kaufman.

Weinstein, N., & Klein, W. (1995). Resistance of personal risk perceptions to debiasing interventions. *Health Psychology*, 132-140.

Woods, D. D. (1995). Toward a Theoretical Base for Representation Design in the Computer Medium: Ecological Perception and Aiding Human Cognition. In J. M. Flach, P. A. Hancock, J. Caird & K. J. Vicente (Eds.), *Global Perpsectives on the Ecology of Human-Machine Systems*. Mahwah, NJ: Erlbaum.

Appendix A: Excerpts of Code from the Server Application

This appendix presents the important code excerpts that implement the server application. The *ConnectionModule* class implements the Connection Module. The *MessageDispatcher* class implements the Message Dispatch Module. The *Session* class implements the Abstraction Layer Module with the PG algorithm.

```
/**
 * This class implements the connection module of the SA. The
connection module provides
 * the capability to receive data from RESCHU and send data to the GUI.
 *
 * @author Hank
 *
 */
public class ConnectionModule {
	private int RECEIVE_PORT = 4446;
	private int BROADCAST_PORT = 4447;
	private String ADDRESS = "230.0.0.1";

	// receiver socket
	private MulticastSocket r_socket;
	// sender socket
	private DatagramSocket s_socket;
	// server address
	private InetAddress address;

	/**
	 * Constructs a ConnectionModule with default listen and send IP
addresses and ports.
	 *
	 * Default IP address is 230.0.0.1.
	 * Send port is 4447.
	 * Receive port is 4446.
	 */
	public ConnectionModule() {
		boolean testing = true;
		setAddress(testing);

		try {
			r_socket = new MulticastSocket(RECEIVE_PORT);
			address = InetAddress.getByName(ADDRESS);
			r_socket.joinGroup(address);

			System.out.println("Listening to port
"+RECEIVE_PORT+" at "+ADDRESS+" ...");
			s_socket = new DatagramSocket();
			System.out.println("Waiting to send to port
"+BROADCAST_PORT+" at "+ADDRESS+" ...");
		} catch (IOException e) {
		}
	}
```

```
	/**
	 * Prompts user for the IP addresses and the ports,
	 * if not in testing mode, i.e. testing=false.
	 *
	 * @param testing - true if testing
	 */
	public void setAddress(boolean testing) {
		if (testing) return;
		try {
			String port = JOptionPane.showInputDialog("Please
enter server's listening port number","4446");
			RECEIVE_PORT = Integer.parseInt(port);
			port = JOptionPane.showInputDialog("Please enter
server's outgoing port number","4447");
			BROADCAST_PORT = Integer.parseInt(port);
		} catch (Exception e) {
			e.printStackTrace();
		}
		ADDRESS = JOptionPane.showInputDialog("Please enter server
IP address","230.0.0.1");
	}

	/**
	 * Closes all the connections.
	 */
	public void close() {

		s_socket.close();
	}

	/**
	 * Listens to incoming datagram packet data,
	 * and fills the empty packet with data if data arrives.
	 *
	 * @param rPacket - an empty datagram packet waiting to be filled
with incoming data.
	 */
	public void receive(DatagramPacket rPacket) {
		try {
			r_socket.receive(rPacket);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	/**
	 * Sends the message string
	 *
	 * @param msg - the message to be sent
	 */
	public void send(String msg) {
		byte[] buf = msg.getBytes();
		try {
			InetAddress group = InetAddress.getByName(ADDRESS);
			DatagramPacket s_packet = new DatagramPacket(buf,
buf.length, group, BROADCAST_PORT);
			s_socket.send(s_packet);
```

```
            } catch (UnknownHostException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}

/**
 * This class implements the Message Dispatch Module of the SA. This
class provides
 * basic parsing of the incoming messages and the routing of the
messages.
 *
 * @author Hank
 *
 */
public class MessageDispatcher {

      private HashMap<String,Session> sessions;

      private final String BEGIN_TAG = "BEGIN";
      private final String CLICK_TAG = "CLICKED";
      private final String END_TAG = "CLICKED";

      /**
       * Constructs a MessageDispatcher.
       */
      public MessageDispatcher() {
            sessions = new HashMap<String,Session>();

      }

      /**
       * Processes incoming messages and routes them to proper
destinations.
       * Handles outgoing messages and insert them on the server
outgoing queue.
       *
       * @param incomingMessage
       * @throws IOException
       */
      public void processMessage(String incomingMessage) throws
IOException {
            String msg = null;

            String[] tokens = incomingMessage.split(":");
            String user = tokens[0];
            if (tokens[1].equals(BEGIN_TAG)) {
                  // initialize model
                  if (sessions.containsKey(user))
sessions.get(user).kill();
                  Session session = new
Session(user,Server.getServer().getHSMM());
                  sessions.put(user, session);
                  msg = session.createBeginMessage();
```

```
            } else if (tokens[1].equals(CLICK_TAG)) {
                  // send click message with timetag
                  msg = sessions.get(user).createClickMessage();
            } else if (tokens[1].equals(END_TAG)){

            } else {
                  msg = processUserEvent(user,incomingMessage);
            }

            if (msg!=null) Server.getServer().queueMessage(msg);
      }

      private String processUserEvent(String user, String s) {
            if (!sessions.containsKey(user)) {
                  Session session = new
Session(user,Server.getServer().getHSMM());
                  sessions.put(user, session);
            }
            Session currentSession = sessions.get(user);

            // if event not recognized return null
            if (!currentSession.addUserEvent(s))
                  return null;

//          long start = currentSession.getMessageSentTime();
            String result = currentSession.generatePredictions();
//          long end = System.currentTimeMillis();
//          long latency = end-start;
//          currentSession.addLatency(latency);
//          System.err.println(latency+" ms --" + user + "--avg--" +
currentSession.getAvgLatency() + " ms");
//          System.out.println(result);
            return result;
      }
}


/**
 * The Session class stores information associated with one RESCHU
operator during a RESCHU
 * simulation. This class also provides the ability to generate
predictions.
 *
 * @author Hank
 *
 */
public class Session {

      private static final int SCORING_WINDOW_WIDTH = 10;
      private static final int EVENT_SEQUENCE_LENGTH = 50;

      private static final int PREDICTION_LENGTH = 210;

      private static final int[] subScores = {50,45,40,20,10};

      static double A = 0.9;
      static double B = 1 - A;
```

```
	// holds events received from PESCHU
	private ArrayList<UserEvent> receivedEvents;

	// holds events the model expects to observe
	private List<EPPair> expectedEvents;

	// holds past n scores
	ArrayList<Integer> scores;

	...

	/**
	 * Constructs a Session object from a username and the HSMM
	 * object from server
	 */
	public Session(String username, MyHSMM model) {
		start_time = System.currentTimeMillis();
		this.username = username;
		this.model = model;

		receivedEvents = new ArrayList<UserEvent>();
		expectedEvents = new ArrayList<EPPair>();
		scores = new ArrayList<Integer>();
		for (int i = 0; i<SCORING_WINDOW_WIDTH; i++) {
			addScoreToWindow(100,scores);
		}

		currentScore = 100;

		currentState = this.model.getInitialState();
		expectedEvents = model.getTopEvents(currentState);
		updateStateData();
		currentStateBeginTime = -1;

		timer = new Timer(true);
	}

/**
 * Returns the BODY of a PREDICT message composed of series of
predictions.
 *
 * @return predications - the PREDICT message of length
predictionLength seconds
 */
public String generatePredictions() {
	String result = "";
	String states = "";

	ArrayList<Integer> scores = new ArrayList<Integer>(this.scores);
	ArrayList<Integer> lowscores = new
ArrayList<Integer>(this.scores);

	int[] evts = this.getEventSequence(EVENT_SEQUENCE_LENGTH);

	int state = this.currentState;
```

```
        int tcount = 0;
        int score = 0;
        int lowscore = 0;
        int dev = 0;
        int dt = 0;

        dt = model.getExpectedDuration(state);
        score = this.currentScore;
        lowscore = this.currentScore;
        dev = 100;
        tcount += dt;

        // patch up the current events sequence
        int eventduration = dt*model.timeResolution-1; // in 0.5 seconds.
        int[] temp = new int[evts.length+eventduration];

        for (int i = 0; i<eventduration; i++)
              temp[evts.length+i] = evts[evts.length-1];

        evts = temp;

        result+=String.valueOf(dt)+":"+String.valueOf(score)
        +":"+String.valueOf(0)+":"+String.valueOf(score)+";";

        while (tcount<PREDICTION_LENGTH) {
              states +=state+"->";

              // add the expected score to the running total
              List<EPPair> eps = model.getTopObservables(evts);
              addScoreToWindow(calculateBestScore(eps), scores);
              addScoreToWindow(calculateLowScore(eps), lowscores);

              dt = model.getExpectedDuration(state);
              score = calculateScore(scores);
              lowscore = calculateScore(lowscores);
              dev = calculateConfidence(eps);

              eventduration = dt*2; // in 0.5 seconds.
              temp = new int[evts.length+eventduration];

              System.arraycopy(evts, 0, temp, 0, evts.length);

              for (int i = 0; i<eventduration; i++)
                    temp[evts.length+i] = eps.get(0).eventtype;

              evts = temp;

              // update the state using the new sequence
              state = model.getNextState(state);

              tcount += dt;
              result+=String.valueOf(dt)+":"+String.valueOf(score)
              +":"+String.valueOf(dev)+":"+String.valueOf(lowscore)+";";
        }
        states+=state;
        return username + ":" + System.currentTimeMillis()+"@"+result;
}
```

Appendix B: Excerpts of Code from the HSMM Library

This appendix presents important code excerpts from the HSMM library, including the five Java functions described in section 4.2.3.1, as well as the Viterbi algorithm implementation.

```
/**
 * mostLikelyStateSequence is a method that calculates the most likely
 * state sequence given an observation sequence using the Viterbi
 * algorithm described in Guedon's paper.
 *
 * @param oseq - an observation sequence
 * @return stateSequence - the most likely state sequence to be seen
 * across this observation sequence for this HSMM
 */
public int[] mostLikelyStateSequence(int[] oseq)
{
      //calculate the most likely state sequence using viterbi
      Viterbi_HSMM vh = new Viterbi_HSMM();
      return vh.viterbiCalc(oseq, this.clone());
}

/**
 * viterbiCalc calculates the most likely state sequence given an
 * observation sequence using the Viterbi method as described by Guedon
 *
 * @param x observation sequence
 * @param HSMM HSMM
 * @return most likely state sequence
 */
public int[] viterbiCalc(int[] x, HSMM HSMM)
{
      //need as input x, b, p, d
      //need to store: Fj(t), StateIn_j(t+1),N_t

      /*these constants should be set by input*/
      double[][] b = HSMM.b;
      double[][] p = HSMM.p;//need values for->transition probabilities
      double[][] d = HSMM.d;//need values for->occupancy distribution
      double[] pi = HSMM.pi;
      int tau = x.length;//length of observation sequence
      int J = p.length;//number of states
      int[] M = new int[J];//need values for-> represents upper bound
of the time spent in state j
      for(int j = 0;j<J;j++)
            M[j] = d[j].length-1;
      /*end constants being set by input*/

      /*initialize certain variables*/
      double[][] F = new double[J][tau];
      double Observ;
      double[] N = new double[tau];
      double[][] D = Dfromd(d,tau);
```

```
      double[][] StateIn = new double[J][tau];//should be able to
calculate from something
      /*end initialize certain variables*/

      int[] stateSequence = new int[x.length];
      double[] likelihoods =  new double[x.length];
      boolean Bfix = true;

      for(int t = 0;t<=tau-1;t++)
      {
            N[t] = 0;
            for(int j = 0;j<J;j++)
            {
                  F[j][t] = 0;
                  Observ = b[j][x[t]];
                  if(t<tau-1)
                  {
                        for(int u =1;u<=Math.min(t+1, M[j]);u++)
                        {
                              if(u<t+1)
                              {
                                    double curVar =
Observ*d[j][u]*StateIn[j][t-u+1];
                                    F[j][t] += curVar;
                                    if(Bfix)
                                          curVar *= b[j][x[t]];
                                    if(curVar>likelihoods[t])
                                    {
                                          likelihoods[t] = curVar;
                                          stateSequence[t] = j;
                                    }
                                    N[t] +=
Observ*D[j][u]*StateIn[j][t-u+1];
                                    Observ *= b[j][x[t-u]]/N[t-u];
                              }
                              else// if(u==tau+1)
                              {
                                    double curVar =
Observ*d[j][t+1]*pi[j];
                                    F[j][t] += curVar;
                                    if(Bfix)
                                          curVar *= b[j][x[t]];
                                    if(curVar>likelihoods[t])
                                    {
                                          likelihoods[t] = curVar;
                                          stateSequence[t] = j;
                                    }
                                    N[t] += Observ*D[j][t+1]*pi[j];
                              }
                        }
                  }
                  else// if(t==tau-1)
                  {
                        for(int u = 1;u<=Math.min(tau, M[j]);u++)
                        {
                              if(u<tau)
```

```
                                {
                                        double curVar =
Observ*D[j][u]*StateIn[j][tau-u];
                                        F[j][tau-1] += curVar;
                                        if(Bfix)
                                              curVar *= b[j][x[t]];
                                        if(curVar>likelihoods[t])
                                        {
                                              likelihoods[t] = curVar;
                                              stateSequence[t] = j;
                                        }
                                Observ *= b[j][x[tau-1-u]]/N[tau-1-];
                                }
                                else// if(u==tau)
                                {
                                double curVar = Observ*D[j][tau]*pi[j];
                                F[j][tau-1] += curVar;
                                        if(Bfix)
                                              curVar *= b[j][x[t]];
                                        if(curVar>likelihoods[t])
                                        {
                                              likelihoods[t] = curVar;
                                              stateSequence[t] = j;
                                        }
                                }
                        }
                  }
                  N[tau-1] += F[j][tau-1];
            }

            for(int j = 0;j<J;j++)
                  F[j][t] /= N[t];

            if(t<tau-1)
            {
                  for(int j=0;j<J;j++)
                  {
                        StateIn[j][t+1] = 0;
                        for(int i =0;i<J;i++)
                              StateIn[j][t+1] += p[i][j]*F[i][t];
                  }
            }
      }
      return stateSequence;
}

private double[][] Dfromd(double[][] d, int tau)/*get survivor
occupancy distribution from original occupancy distribution*/
{
      int J = d.length;;
      double[][] D = new double[J][tau+1];//survivor function of
occupancy
      for(int i = 0;i<J;i++)
            for(int j = 0;j<=tau;j++)
            {
                  D[i][j] = 0;
                  for(int k = 0;k<d[0].length;k++)
```

```
                {
                        if(k>=j)
                                D[i][j] += d[i][k];
                }
        }
    return D;
}
/**
 * Returns an array log probs corresponding to the number of different
 * possible final events
 * @param x - the sequence of events
 * @return logprobs - the array of log probs of reaching the final
 * events from the sequence of events x
 */
public double[] logProbability2(int[] x)
{
    HSMM initHSMM = this;
    /*these constants should be set by input*/
    double[][] b = initHSMM.b;
    double[][] p = initHSMM.p;//need values for->transition
probabilities
    double[][] d = initHSMM.d;//need values for->occupancy
distribution
    double[] pi = initHSMM.pi;
    int tau = x.length;//length of observation sequence
    int J = p.length;//number of states
    int O = b[0].length;//number of observables
    int d_len = d[0].length-1;//need values for-> represents upper
bound of the time spent in state j
    /*end constants being set by input*/

    /*initialize certain variables*/
    double[][] F = new double[J][tau+1];
    double Observ;
    double[] N = new double[tau+1];
    double[][] D = forwardBackward.Dfromd(d,tau+1);

    double[][] StateIn = new double[J][tau+1];//should be able to
calculate from something
    /*end initialize certain variables*/

    double[] logProbs = new double[19];
    Arrays.fill(logProbs, 0.0);

    Arrays.fill(N,0.0);
    //doing calculations as described in Guedon's paper
    for(int t = 0;t<=tau-1;t++)
    {
        for(int j = 0;j<J;j++)
        {
            F[j][t] = 0;
            Observ = b[j][x[t]];
            for(int u =1;u<=Math.min(t+1, d_len);u++)
            {
                if(u<t+1)
                {
```

```
                                    F[j][t] += Observ*d[j][u]*StateIn[j][t-
u+1];
                                    N[t] += Observ*D[j][u]*StateIn[j][t-u+1];
                                    Observ *= b[j][x[t-u]]/N[t-u];
                              }
                              else// if(u==tau+1)
                              {
                                    F[j][t] += Observ*d[j][t+1]*pi[j];
                                    N[t] += Observ*D[j][t+1]*pi[j];
                              }
                        }
                  }

                  for(int j = 0;j<J;j++)
                        F[j][t] /= N[t];

                  for(int j=0;j<J;j++)
                  {
                        StateIn[j][t+1] = 0;
                        for(int i =0;i<J;i++)
                              StateIn[j][t+1] += p[i][j]*F[i][t];
                  }

                  for(int o = 0;o<O;o++)
                        if(!Double.isNaN(N[t]) && !(N[t] <= 0))
                              logProbs[o] += Math.log(N[t]);
                        else
                              logProbs[o] += -100000000;

            }

            double[] Nlast = new double[O];
            //for last step
            int t = tau;
            for(int j = 0;j<J;j++)
            {
                  F[j][t] = 0;
                  Observ = 1;
                  for(int u = 1;u<=Math.min(tau+1, d_len);u++)
                  {
                        if(u<tau+1)
                        {
                             F[j][tau] += Observ*D[j][u]*StateIn[j][tau+1-u];
                              Observ *= b[j][x[tau-u]]/N[tau-u];
                        }
                        else// if(u==tau)
                        {
                              F[j][tau] += Observ*D[j][tau+1]*pi[j];
                        }
                  }
                  for(int o = 0;o<O;o++)
                  {
                        Nlast[o] += F[j][tau]*b[j][o];
                  }
            }
            for(int o = 0;o<O;o++)
                  if(!Double.isNaN(Nlast[o]) && !(Nlast[o] <= 0))
```

```
                    logProbs[o] += Math.log(Nlast[o]);
              else
                    logProbs[o] += -100000000;
        //end for last step
        return logProbs;
}

/**
 * Returns the next most likely state from n
 *
 * @param n - the current state
 * @return nextState - next most likely state from n
 */
public int getNextState(int n) {
        double max = -1;
        int nstate = -1;

        for (int i=0; i<p[n].length; i++) {
              if (p[n][i]>max) {
                    max = p[n][i];
                    nstate = i;
              }
        }

        return nstate;
}

/**
 * Returns the expected duration of state n
 *
 * @param n - the state
 * @return expectedDuration - state n's expected duration in seconds
 */
public Integer getExpectedDuration(int n) {
        double[] timeDistr = d[n];
        double dt = 0;
        for (int i = 0; i<timeDistr.length; i++) {
              int x = i+1;
              dt += x*timeDistr[i];
        }
        return (int)Math.round(dt/timeResolution);
}

/**
 * Returns a list of 5 EPPairs representing the top 5 most
 * likely events and the associated normalized probability
 *
 * @param evts - an array of the sequence of observed events
 * @return topObs - a list of 5 EPPairs representing the top 5 most
 * likely events and the associated normalized probability
 */
public List<EPPair> getTopObservables(int[] evts) {
        List<EPPair> results = new ArrayList<EPPair>();

        List<EPPair> temp = new ArrayList<EPPair>();
```

```
        int[] tempevts = evts.clone();

        double[] logprobs = this.logProbability2(tempevts);

        for (int evt = 0; evt<logprobs.length; evt++) {
              double logprob = logprobs[evt];
              int offset = (int)Math.ceil(logprob);
              logprob -= offset - offset%500;
              double prob = Math.exp(logprob);
              temp.add(new EPPair(evt,prob));
        }

        Collections.sort(temp);
        Collections.reverse(temp);

        double benchmark = 0;
        for (int i = 0; i<5; i++) {
              EPPair ep = temp.get(i);
              benchmark += ep.prob;
              results.add(ep);
        }

        this.normalize(benchmark, results);
        return results;
}
```

Appendix C: Excerpts of Code from the GUI

This appendix presents important code excerpts that implement the GUI, including a *HistoryBar* that draws a single performance history bar on the Model Performance History Panel and the *drawStraightPaths* function that draws the trend lines and the shaded gray area on the Model Accuracy Prediction Panel.

```
/**
 * This class draws a single Model Performance History Bar used to
populate
 * the Model Performance History Panel, given the underlying
performance
 * history data.
 *
 * @author Hank
 *
 */
public class HistoryBar extends JComponent implements DataListener {

	private static int PIXEL_PER_SECOND = 2;

	private PerformanceHistory history;

	/**
	 * Constructs a HistoryBar object.
	 *
	 * @param width
	 */
	public HistoryBar(int width) {
		setPreferredSize(new Dimension(width, -1));
	}

	/**
	 * Sets the PerformanceHistory data that this HistoryBar draws
	 * @param performanceHistory
	 */
	public void setPerformanceHistory(PerformanceHistory
performanceHistory) {
		this.history = performanceHistory;
		performanceHistory.setHistoryListener(this);
		performanceHistory.start();
	}

	@Override
	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		Graphics2D g2 = (Graphics2D) g.create();
		g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
				RenderingHints.VALUE_ANTIALIAS_ON);


		g.setColor(Color.BLACK);
```

```
            g.fillRect(0, 0, this.getWidth(), this.getHeight());

            if (history==null||history.isEmpty()) return;

            int x = 0;
            int y = 0;
            int w = 0;
            int h = 40;

            for (int i=0; i<history.getLevels().size(); i++) {
                  int t = history.getCLevel(i).duration;
                  int l = history.getCLevel(i).devLevel;


                  w = t*PIXEL_PER_SECOND;

                  setColor(g,l);
                  g.fillRect(x, y, w, h);

                  x = x + w;
            }
      }

      private void setColor(Graphics g, int level) {
            if (level <= SystemConstants.HIGH_MID_CUTOFF)
g.setColor(SystemConstants.HIGH_CONFIDENCE_COLOR);
            else if (level <=
SystemConstants.HIGH_MID_CUTOFF+SystemConstants.CUTOFF_SPACING)
g.setColor(SystemConstants.MEDIUM_CONFIDENCE_COLOR);
            else g.setColor(SystemConstants.LOW_CONFIDENCE_COLOR);
      }

      @Override
      public void updateData() {
            try {
                  SwingUtilities.invokeAndWait(new Runnable(){
                        public void run() {repaint();}
                  });
            } catch (InterruptedException e) {
                  e.printStackTrace();
            } catch (InvocationTargetException e) {
                  e.printStackTrace();
            }
      }
}


/*
 * Draws the trend lines and the shaded gray area for the
 * Model Accuracy Panel, given a parsed PREDICT message
 */
private synchronized void drawStraightPaths(Graphics g) throws
NullPointerException {
      if (path==null || path.getEdges()==null) return;

      int base_score = path.getBaseScore();
      int dwell_time = path.getFirstDwelltime();
```

```
        g.setColor(SystemConstants.HIGH_CONFIDENCE_COLOR);

        int start_x = 60;
        int next_x = 60;

        int start_y = (100-base_score)*pixelPerPoint+60;
        int next_y = start_y;

        int low_start_y = start_y;
        int low_next_y = start_y;

        int dx = dwell_time*pixelPerSecond;
        int decay_x = start_x + (int)(dx*decayPoint);

        // draw initial line to decay point
        drawThickLine(g, start_x, start_y, decay_x, next_y, true);

        next_x += dx;

        for (int i = 0; i<path.getEdges().size(); i++) {
              PredictionStep pe = path.getEdges().get(i);
              int p = pe.getExpectedMAS();
              int l = pe.getLowerBound();
              int conf = pe.getPredictionConfidence();

              next_y = (100-p)*pixelPerPoint+60;
              low_next_y = (100-l)*pixelPerPoint+60;

              // draw decay lines
              drawThickLine(g,decay_x, start_y, next_x, next_y, true);

              // if it's the first lower bound, start from start_x
              if (i==0) {
                    drawThickLine(g,start_x, start_y, next_x,
low_next_y,false);

                    fillGrayArea(g, decay_x, start_y, next_x, next_y,
                                start_x, start_y, next_x, low_next_y);
              } else {
                    drawThickLine(g,decay_x, low_start_y, next_x,
low_next_y,false);

                    fillGrayArea(g, decay_x, start_y, next_x, next_y,
                                decay_x, low_start_y, next_x, low_next_y);
              }

              dwell_time = pe.getExpectedDuration();

              // update start_y's
              start_y = next_y;
              low_start_y = low_next_y;

              if (start_y > 460) return;

              // update decay_x
              dx = dwell_time*pixelPerSecond;
```

```
            decay_x = next_x + (int)(dx*decayPoint);

            if (decay_x > 600) {
                  // draw last segment within box.
                  drawThickLine(g, next_x, start_y, 600, start_y,true);
                  drawThickLine(g, next_x, low_start_y, 600,
low_start_y,false);

                  fillGrayArea(g, next_x, start_y, 600, start_y,
                              next_x, low_start_y, 600, low_start_y);
                  return;
            }

            // draw the next flat segment
            setColor(g,conf);

            drawThickLine(g, next_x, start_y, decay_x, start_y, true);
            drawThickLine(g, next_x, low_start_y, decay_x, low_start_y,
false);

            fillGrayArea(g, next_x, start_y, decay_x, start_y,
                                    next_x, low_start_y, decay_x,
low_start_y);

            start_x = next_x;
            next_x += dx;
      }
      drawThickLine(g, decay_x, start_y, 600, start_y, true);
      drawThickLine(g, decay_x, low_start_y, 600, low_start_y, false);
      fillGrayArea(g, decay_x, start_y, 600, start_y,
                              decay_x, low_start_y, 600, low_start_y);
}
```