Modeling Human Workload in Unmanned Aerial Systems

TJ Gledhill

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Michael A. Goodrich, Chair
Eric M. Mercer
Kevin Seppi

Department of Computer Science

Brigham Young University

August 2014

ABSTRACT

Modeling Human Workload in Unmanned Aerial Systems

TJ Gledhill
Department of Computer Science, BYU
Master of Science

Unmanned aerial systems (UASs) often require multiple human operators fulfilling diverse roles for safe and correct operation [11, 14, 24]. Reliably designing the human interaction, autonomy, and decision making aspects of these systems requires the use of modeling. We propose a conceptual model that models human machine interaction systems as a group of actors connected by a network of communication channels. We also propose a workload taxonomy derived from a review of the relevant literature, which we then apply to the conceptual model. We present a simulation framework implemented in Java, with an optional XML model parser, that can be analyzed using the Java Pathfinder (JPF) model checker. The simulator produces a workload profile over time for each human actor in the system. We conducted case studies by modeling two different UAS: Wilderness search and rescue using an unmanned aerial vehicle (WiSAR) and UAS integration into the national air space. The results of these case studies are consistent with known workload events and the simple workload metric presented by Wickens [32].

Keywords: human workload, unmanned aerial system, UAS, national air space, unmanned aerial vehicle, modeling human machine interaction

ACKNOWLEDGMENTS

I would like to thank my wife for her constant support and encouragement in pursuing this research during both the good and bad times. I would like to thank my adviser Dr. Goodrich and Dr. Mercer for truly taking on the roles of teachers and for being patient with me throughout this process. I would like to thank Jared Moore and Robert Ivie for the invaluable help they provided with the code and the publications. I would like to thank Neha Rungta of NASA Ames Intelligent Systems Division for her help with JPF and Brahms. I would also like to thank the NSF IUCRC Center for Unmanned Aerial Systems, and the participating industries and labs, for funding the work. Lastly I would like to thank my lab mates for their friendship and help.

Table of Contents

List of Figures

Chapter 1

Introduction and Overview

Most existing Unmanned Aerial Systems (UASs) require two or more human operators [14, 24]. Standard UAS practice is to have one human to control the aerial vehicle and another to control the camera or other payloads. In addition to this a third human is often responsible for overseeing task completion and interfacing with the command structure. Although some argue persuasively that this is a desirable organization [25], there is considerable interest in reducing the required number of humans and reducing human workload using improved autonomy and enhanced user interfaces [11, 17, 20].

UAV enabled wilderness search and rescue (WiSAR) has been a focus of the Human Centered Machine Intelligence (HCMI), Multiple Agent Intelligent Coordination and Control (MAGICC) and Computer Vision (CV) labs at Brigham Young University since 2005. In that time research has been conducted on human interaction with miniature unmanned aerial vehicles (mUAVs), improving target detection by enhancing video taken from a mUAV, integrating mUAVs into a search and rescue environment, and improving the mUAVs chance of getting video footage of the target.

The research results over this time period appear promising [9, 10, 13, 18, 19, 23, 27**?**], however, it remains unknown how these improvements will effect human workload within the context of a UAS. While it was possible to make changes to our existing WiSAR UAS and then perform user studies to view the effects of these changes, we felt that the more important scientific problem lay in discovering a way to measure changes in human workload as a result of changes to a UAS. To accomplish this we chose to use system modeling.

System modeling is not a new approach, though applying system modeling to human-machine systems requires extending most modeling languages. There are many different modeling languages, each of which is designed to perform specific types of validation [4]. For example, Brahms focuses on performing agent-based simulations, EOFM focuses on task analytics, and ACT-R focuses on human cognition. Due to the complexity of these modeling languages and our need to manipulate the internal workings of the language we decided that our shortest path forward was to create our own simple modeling framework. We call this modeling framework the Model Abstraction Framework.

The Model Abstraction Framework consists of the following core components: Models, Model Parser, Modeling Interface, Simulator, and Workload Viewer. A link to the source code is found in appendix B. The Model Abstraction Framework uses XML as the primary modeling language. The models consist of a set of Directed Role Graphs (DiRGs) representing the entities which are performing actions, and a Directed Team Graph (DiTG) representing the communication network between the different entities. The model is then parsed using the Model Parser. The Model Parser checks that the model syntax is correct and converts the XML into a set of Java classes that implement the Modeling Interface, a set of Java interfaces and abstract classes. The Modeling Interface allows the model to be expressed as a labeled state transition system to be run by the Simulator. The Simulator is a core set of Java classes which simulates the running of the model and gathers workload metrics. The simulation can be performed as a stand-alone application or it can be run inside of Java Pathfinder (JPF) to perform model checking. The workload viewer allows us to view human workload measures in real-time when the simulation is run as a stand-alone application.

We present two workload measurements, *Resource Workload Metric* and *Decision Workload Metric*, based off of multiple resource theory [32] with ties to queuing theory [26] and operator fan-out theory [17]. By relating these theories to the different operational components of the model we obtain a quantitative measure of a human's workload for each time-step in the system. Additionally we adapt the Wickens' computational model [32] to

the Model Abstraction Framework in an attempt to see how and where this metric differs from our own.

We perform a case study representing the introduction of a UAS into the National Air Space. We chose to model a UAS operating within the National Air Space for three reasons: *a)* it is similar to the work done in WiSAR, *b)* the goals outlined by the FAA are similar to our own [31], and *c)* the lack of modeling information, requiring the use of high levels of abstraction. As part of the case study we created four variations of the model to see how the workload metrics varied between models.

The verification of the metrics in this thesis is done using a *consistency* approach. Using known high and low workload scenarios obtained from WiSAR [2], we check that the workload is consistent with our expected results. While these results are inconslusive regarding actual human workload, we were pleased to find that the workload measures are consistent with known high workload scenarios. We were also pleased that our workload metric, while more expressive, was consistent with the Adapted Wickens' Model.

1.1 Overview and Papers

Chapters 2 consists of a paper published in the IEEE International Conference on Systems, Man, and Cybernetics, Manchester, England, 2013. It presents our core conceptual model, details of the Model Abstraction Framework, and a WiSAR model implemented in Java. It also presents how the Simulator is able to validate the model using JPF.

Chapter 3 presents our baseline workload metric, which is an adaptation of the Wickens' computational model for calculating cognitive resource load[32]. It then presents two new workload metrics, the*Resource Workload Metric* and the *Decision Workload Metric*.

Chapter 4 presents a case study which involves modeling the integration of a UAS into the NAS. It describes the model scenario, four model variations, and provides a detailed analysis and comparison of the results generated by these models.

Chapter 5 describes the similarities and differences with other work that is closely related to our approach of measuring human workload and modeling human-machine systems.

Chapter 5 contains a summary of the work, threats to validity, and future work needed to accomplish our goals.

Chapter 2

Modeling UASs for Role Fusion and Human Machine Interface Optimization

In relation to this thesis contributions were made to all aspects of this publication.

TJ Gledhill, Eric Mercer and Michael A. Goodrich. Modeling UASs for Role Fusion and Human Machine Interface Optimization. In Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, Manchester, England, 2013.

Abstract

Currently, a single Unmanned Aerial System (UAS) requires several humans managing different aspects of the problem. Human roles often include vehicle operators, payload experts, and mission managers [11, 15, 24]. As a step toward reducing the number of humans required, it is desirable to reduce operator workload through effective distributed control, augmented autonomy, and intelligent user interfaces. Reliably doing this requires various roles in the system to be modeled. These roles naturally include the roles of the humans, but they also include roles delegated to autonomy and software decision-making algorithms, meaning the GUI and the unmanned aerial vehicle. This paper presents a conceptual model which models the roles of complex systems as a collection of actors, running in parallel. Results from applying this model to the UAS-enabled Wilderness Search and Rescue (WiSAR) domain indicate (a) it is possible to model the entire WiSAR system at varying degrees of abstraction (b) that building and evaluating the model provides insight into the best practices of WiSAR teams and (c) a way to model human machine interactions that works directly with the Java Pathfinder model checker to detect errors.

2.1 Introduction

Most existing Unmanned Aerial Systems (UASs) require two or more human operators[15, 24]. Standard UAS practice is to have one human to control the aerial vehicle and another to control the camera or other payloads. In addition to this a third human is often responsible for overseeing task completion and interfacing with the command structure. Although some argue persuasively that this is a desirable organization [25], there is considerable interest in reducing the required number of humans and reducing human workload using improved autonomy and enhanced user interfaces [11, 17, 20].

The broad research context driving this paper takes a multistep approach: (a) model the roles for a specific UAS and a well-defined set of tasks, (b) delimit assumptions and abstractions used in the model, (c) verify properties of the model, (d) use the model to explore ways of combining roles in such a way that operator workload and the number of humans is minimized, and (e) design vehicle autonomy and user interface support to allow a real UAS team to operate more efficiently. The focus of this paper is on the important lessons learned in the first two steps.

The modeling used in this paper could be applied to a number of tasks, but we focus on Wilderness Search and Rescue for two reasons. First, the authors have done prior work on UAS-enabled Wilderness Search and Rescue (WiSAR) [16]; and second, there is a host of modeling information about how WiSAR is currently performed [**?**]. The UAS-enabled WiSAR systems produced by this research requires three humans, two GUIs, and a single UAV.

To gain insight into WiSAR we have chosen to model the system as a group of directed role graphs (DiRGs). Each human, GUI, and UAV represent a DiRG, allowing evaluation of potential conflicts between and opportunities for unification of the various roles. These DiRGs are referred to as *Actors* in the models below.

Modeling is essentially a process of abstraction, choosing which elements of a system are essential and which are not [7]. Since one of the goals of this paper is to use model-checking

to evaluate models, we choose a model class that is simple enough that it allows us to clearly delineate between what is modeled and what is not. Thus, we use DiRGs, which can be expressed as Mealy state machines, to model different WiSAR roles. These models explicitly encode key aspects of the various actors, and collectively form a group of Mealy machines that run in parallel. The model is encoded in Java using a custom set of interfaces designed to simulate a discrete time environment, facilitate input/output between roles, and provide non-deterministic event handling.

Model checking is performed using Java Pathfinder (JPF). This is convenient because JPF runs the model checking on the compiled Java code generated by the modeling exercise.

The results of this modeling exercise indicate (a) it is possible to model the entire WiSAR system using varying degrees of abstraction and (b) that building and evaluating the model provides insight into the best practices of WiSAR teams and (c) a way to model human machine interactions that works directly with the Java Pathfinder model checker to detect errors.

2.2 Related Work

NASA Ames Research Center (NASA ARC) is using Brahms, a complex and robust language, to model interactions between operators and their aerial equipment [28]. To study the Uberlingen collision, an in air collision of two commercial passenger planes, Rungta and her colleagues produced a model entirely in the Brahms language. This model correctly predicts the collision and also reveals some of the difficulties intrinsic to this type of system.

One critical aspect of NASA ARCs work carries over into our own: variable task duration [1]. Task duration directly influences whether the situation ends safely, barely avoids a crash, or crashes. The biggest advantage Brahms has over Java comes from its strict grammar. However, Brahms must be translated into Java before using JPF, a step we avoid by implementing our model in Java directly.

Bolton and Bass used the Enhanced Operator Function Model (EOFM) language to create a model consisting of the Air Traffic Controller, the pilot flying the plane, and the pilot monitoring the equipment, as well as the interfaces they used [5]. As they increased the number of allowable miscommunications, their system had an exponential increase of errors. EOFM facilitates the division of goals into multiple levels of activities. These activities can then be broken into atomic actions [6]. The main difference between this model and our own is that EOFM is expressed in XML while ours is expressed in Java. This allows us to perform model checking directly using JPF.

Wilderness Search and Rescue is primarily concerned with finding people who have become lost in rugged terrain. Research has shown that UASs could potentially be used to facilitate this work. Goodrich et al. tested the effectiveness of these types of operations [14]. A key outcome of these field tests is the speculation that effectiveness could be enhanced if the roles of the UAV operator and video operator were combined.

In prior work, a goal-directed task analysis, a work domain analysis, and a control task analysis were performed. [**?**]. These analyses modeled WiSAR as a collection of goals, work domains, and tasks. While these studies proved valuable for understanding the WiSAR processes they were less helpful in suggesting improvements to the WiSAR processes. Indeed, the limitations of such tools for informing the design of technology to support existing processes has lead to new methods for performing such analyses [18]; the work in this paper complements such work, using model-checking to perform analyses on problems that do not lend themselves to answers using other approaches.

2.3 WiSAR UAS Domain

Wilderness search and rescue often occurs in remote, varying, and dangerous terrains. According to [30], there are four core elements of a WiSAR operation: *Locate*, *Reach*, *Stabilize*, and *Evacuate*. The WiSAR UAS operates within this first element so it is the focus of this paper.

During the *Locate* element, the incident commander (IC) develops a strategy to obtain information. This strategy makes use of the available tactics to obtain this information. The WiSAR UAS is one of the tactics that the IC may choose to use. A WiSAR UAS technical search team consists of three humans: Mission Manager, Vehicle Operator, and Video Operator. These constitute the three human roles in the team. Supporting these human roles are two intelligent user interfaces, the Vehicle Operator GUI and the Video Operator GUI; these constitute two other roles that must be modeled. The final role is the aerial vehicle itself, which is equipped with sensors and controllers that enable it to make decisions. Since the WiSAR UAS technical search team must coordinate its efforts with other members of the search team via the IC, we embed the five UAS roles within a Parent Search model. The parent search model represents the entire command structure for the search and rescue operation.

In the next section, we model the WiSAR roles and the interactions between these roles. Naturally, these roles will need to take input from the environment, so we present a simple model of the environment that emphasizes the key environmental elements, probabilistic events and varying task durations. Note that this model of the environment exists at a higher level of abstraction than what is typically considered an environment model in literature; typical models tend to focus on environmental realism, encoding things like terrain, wind, etc, but our model emphasizes events that affect the behavior of the WiSAR roles.

2.4 Conceptual Model

We have chosen to conceptualize the WiSAR UAS as a group of DiRGs. A DiRG represents a sequence of tasks for a single role. By conceptualizing WiSAR as a collection of DiRGs running in parallel we hope to gain more insight into the WiSAR processes with a goal of improving these processes. In prior work, a goal-directed task analysis, a work domain analysis, and a control task analysis were performed. [**?**]. These analyses modeled WiSAR as a collection of goals, work domains, and tasks. While these studies proved valuable for

understanding the WiSAR processes they were less helpful in suggesting improvements to the WiSAR processes. Indeed, the limitations of such tools for informing the design of technology to support existing processes occurs because they discover conditions which may result in problems rather than discovering system problems themselves. Performing system-level task modeling, such as we are, is capable of discovering such problems [4].

While we are using this technique to find such problems we expand on it in several ways. First, this technique is most commonly used for analyzing a single human using an interface. Our models involve a team of humans simultaneously using multiple interfaces which naturally increases the state space of the model, decreasing scalability. Second, we are using this technique to analyze the workload of the system. Our goal is the combining of human roles and interfaces. We hope to gain insight into decreasing the system workload, and possibly combining roles, by establishing metrics associated with the task model and model simulation. These metrics can then be used to determine if changes to the model represent a decrease in operator workload.

As is common when modeling human-automation interaction we have decided to model the DiRGs usings Mealy state machines [4] This allows us to abstract the different WiSAR roles into individual state machines that we call Actors. Actors do not correspond to a single aspect of the WiSAR domain, anything can be an Actor, thus providing the freedom to flexibly model as many aspects of the domain as necessary and at various levels of resolution. This freedom is important and represents our primary method of reducing the state space to manage scalability. Actors transition between states by receiving inputs generated by other Actors and Events. Events are also modeled as Mealy machines whose transitions are triggered by a combination of simulation and Actor inputs. Because Actors and Events may receive input from other Actors and Events the combination of their transition matrices define the wiring of the different DiRGs. A single wire is when an output from one Actor is an input on another Actor. This implies that the set of all inputs is the same as the set of all outputs, however, in practice we do not treat these sets as the same since unhandled input

represents transitions returning to the current state. We ignore these looping transitions except when their behavior tells us something interesting.

Formally, the models are the following mathematical structures:

$$Actor = (S, s_0, \Sigma_A \cup \Sigma, \Lambda_A, T) \tag{2.1}$$

$$Event = (S, S_0, \Sigma_A \cup \Sigma_S, \Lambda_A, T) \tag{2.2}$$

$$T : S \times \Sigma \Rightarrow S \times \Lambda_A \tag{2.3}$$

where S is a set of states, s_0 the start state, Σ_A the set of all Actor inputs, Σ_S the set of all Simulator inputs, Λ_A the set of all Actor outputs, and T a transition matrix which specifies the outputs for any state transition. T may have multiple inputs and multiple outputs.

At this point our conceptual model is implementation agnostic. Indeed, the abstraction allows us to group Actors, break Actors into sub-Actors, and use Actors to validate specific behaviors. The model also allows for complex transitions between Actor states and the ability to enter normally unreachable state spaces using Events. The model is also easily adapted to code which can be verified using model checking tools such as Java Pathfinder (JPF) which we will show in the following sections.

2.5 Simulating the WiSAR UAS

Real WiSAR environments and UAV dynamics are complex so full models of the environment and UAV can also become extremely complex. However, many of the complexities are not relevant to the decisions made by the various WiSAR actors. Consequently, we propose a model that "abstracts away" many unessential details and encodes key aspects of the

environment. In order to simulate critical aspects of the WiSAR UAS model it is necessary to represent communication between Actors, concurrency, and task duration, concepts which are outside the scope of a standard state machine. To do this we constructed a basic simulation framework. The simulation framework is encapsulated into a single Java class, called *Simulator*. This section discusses the key components of this simulation framework.

2.5.1 Core Simulator Objects

The Simulator is made up of the following objects: Team, Actors, Events, States, Transitions, and Unique Data Objects (UDO). We organize these objects in the following way. A Team is a wrapper class representing the entire model which contains a collection of Actors, Events, and shared UDOs. Each Actor contains a set of private States. Each of these States contains a set of Transitions. Each Transition contains a set of input UDOs, a set of output UDOs, the outgoing Actor State, and a reference to the Actors current State. This structure is convenient because it naturally encapsulates the different aspects of a Mealy machine.

Each UDO represents a unique piece of data, input or output, and is flagged as active or inactive. A UDO is temporarily set to active after it is sent as output. Each Transition can easily determine if it is possible by checking to see that all of its input UDOs are active. Each State can then return a list of possible transitions. An Actor evaluates the list of possible Transitions to determine how it should transition. If it is empty then the Actor does not transition, otherwise the Actor chooses a single Transition to occur. An Actor chooses a Transition at the Simulators request. The Simulator tracks when this Transition should occur, at which point the Transition will set each output UDO to active and change the Actors current state to the Transitions outgoing State. Because the UDOs must exist before Actors can be initialized we have created the Team class. The Team class wraps all of the Actors, Events, and UDOs into single entity. This class first initializes each UDO, afterwards each Actor and Event is initialized with a list of input and output UDO references which it will use in its transitions. This structure offers a few benefits. Code wise it allows us to simulate

the transfer of data without actually transferring data which drastically simplifies the code. It also forces us to explicitly define our model wiring in two places, first at the Team level and second at the Actor Transition level as mentioned above. Although the Transition set of inputs is not limited to the set of global UDOs we can still compare the set of global inputs and outputs an Actor receives with the set of inputs and outputs defined by its transitions. Through this we can validate that the set of Actor transitions is complete, as defined in our Team, which is an important step in validating the model. A simple example of the initialization of the Team and its components is shown in the following example:

```
Team {
    UDO A1Output = new UDO()
    UDO E1Output = new UDO()

    UDO Inputs[] = [E1Output]
    UDO Outputs[] = [A1Output]
    Actor A1 = new Actor(Inputs, Outputs)

    UDO Inputs[] = [A1Output]
    Actor A2 = new Actor(Inputs, [])

    UDO Outputs[] = [E1Output]
    Event E1 = new Event([], Outputs)
}

A1(Inputs, Outputs) {
    State S1 = new State()
    State S2 = new State()
```

```
    S1.addTransition(this,
        Inputs.E1_Output,
        Outputs.A1Output,
        S2)
}

A2(Inputs, Outputs) {
    State S1 = new State()
    State S2 = new State()

    S1.addTransition(this,
         Inputs.A1Output,
         null,
         S2)
}

E1(Inputs, Outputs) {
    State S1 = new State()
    State S2 = new State()

    S1.addTransition(this,
        null,
        Outputs.E1Output,
        S2)
}
```

While this is only a basic example it clearly shows how the models have been implemented as code. The example also illustrates the Event class. Events differ from Actors in

that Event transitions require an express command from the Simulator before processing. This allows the Simulator to non-deterministically trigger events which, when run in JPF, can be setup to process events at different intervals or when the system changes state. From this we can determine the effects events have on the system in a very robust manner.

2.5.2 Communication Between Actors

To simulate a team of Actors working together it is necessary to establish a communication medium within the model and simulation which can represent the different forms of communication. In the model this communication medium is represented as inputs, outputs, and transitions.

Our initial attempts to simulate this communication resulted in a PostOffice class attached to the Simulator. Actors sent data along with the name of the recipient to the PostOffice. Actors could then retrieve their input from the PostOffice, much the way PO boxes work. Actors also had the ability to make certain output observable through the PostOffice. This meant that an Actor could place data into the PostOffice for other Actors to observe, a public PO box. When we added sub-Actors to the model code it became necessary to allow Actors and sub-Actors to share both private and public PO boxes. It was also necessary to store current and future output separately to achieve concurrency which we explain in the next section. Although this achieved the desired goal the results were less than satisfactory. In addition to the added complexity the design used implicit input and output connections making it much harder to validate that the code represented the desired model.

Our next iteration of the Simulator simplified this communication medium with the use of the previously defined UDOs. By initializing these UDOs and passing them as references to the Actors and Events we greatly simplified inter-Actor communication. This new design also requires explicit declarations for each UDO connection allowing us to validate the model code with the model and again with the transition matrixes. The UDO is also capable of representing both direct and observable communications which naturally allow sub-Actors to

link inputs with parents. Indeed, Actor relationships are now irrelevant in regard to sharing input and output since Actors only depend on the status of the UDOs. In the case where it does matter which Actor generated the output then a new UDO can be created representing that relationship thus preserving Actor independence through explicit connections.

The Team initialization example above demonstrates the use of UDOs. The example defines two UDOs, A1Output and E1Output. Once defined these UDOs are passed by reference to specific Actors and Events as inputs, explicitly defining the connectivity implied by the UDOs. The Actors use the UDO references for constructing their transition matrices which results in the connecting of A2 to A1 and A1 to E1. If we desired to change our model and allow A2 to transition on E1Output the UDO would be added to the A2 input and A2 would declare a transition for that input resulting in the connectivity of A2 to E1.

2.5.3 Simulating Time

To simulate task duration the simulator uses the delta time algorithm. Each Transition has a specified duration range defined by the Actor or Event. The simulator has five different duration settings for choosing a value within a range: MIN, the minimum; MAX, the maximum; MIN_OR_MAX, a random choice between one of these settings; $MIN\_MEAN\_OR\_MAX$, another random choice.. When an Actor begins a Transition the Simulator chooses a duration, the value of that duration is then converted to the Simulator delta time. Basically if a transition is to finish in 30 time steps but another Transition finishes at 25 time steps then the first Transition is placed after the second Transition and is given a count of 5 which means it happens 5 time steps after the prior Transition. Thus as the simulation progresses time remains relative.

Slightly different from the use of transition durations is the notion of simulating Events. The time range over which an Event may occur is often much larger than the ranges defined by tasks, also, Events are only possible in specific state spaces thus preventing us from predicting the available time range of the Event. This prevents us from simulating

Event timing in the same way as task durations because such large time ranges cannot be accurately represented with with only 2 to 3 choices and we cannot select a min, max, or mean if the range is unknown. We solve this problem in two ways. One method is to trigger the Event at regular time intervals while the Event is possible. Depending on the interval size this can cause a dramatic increase in state space. While this increases the possibility of exploring the possible effects an Event can generate on the system it offers no guarantees. Another method for triggering Events is to trigger the Event on each state change within the system while the Event is possible. This guarantees that the Event will be explored in each state space that is presented during the simulation, however, this is also likely dramatically increase the state space and is much more difficult to implement. Since triggering an Event represents a transition within the Event it is included in the delta time algorithm used for progressing simulation time.

2.5.4 The Simulation Loop

It is now possible to describe the actual simulation. After initialization we enter the simulation phase. This phase is used to transition the Actors and trigger Events. One challenge with transitioning the Actors is the need for concurrency. The simulation must allow multiple Actors to transition without interfering with one another. Previous versions of our Simulator placed each Actor within its own thread. We found that threads complicate the conversion into JPF so instead we chose to use transactions. In each transaction we allow each Actor to make the changes required by its transition. These transitions only modify a temporary value on the UDOs. After all transitions are completed we finish the transaction by moving each temporary UDO value into the actual value. The entire simulation phase can be described thus:

```
Begin Transaction:
    Foreach Actor
        if ( Transition duration reached )
```

```
            Process Transition

    End Foreach
End Transaction

Process Transaction

Foreach Actor
    Transition = Get Next Transition
    If Transition is not null
        Convert Transition duration to delta time.
        Update necessary Actor delta times.
End Foreach
```

When there are no longer any pending Transitions the loop ends and the simulation is terminated.

2.6 WiSAR UAS Model

This section describes the models produced for the UAS-enabled WiSAR process. We first discuss Actors and Events, followed by a brief discussion of Java asserts and a case study drawn from WiSAR.

This conceptual model uses Mealy state machines. This allows us to abstract the different WiSAR roles into individual state machines that we call Actors. Actor states do not correspond to a single aspect of the WiSAR domain, thus providing the freedom to flexibly model as many aspects of the domain as necessary and at various levels of resolution. Actors transition between states are triggered by inputs generated by Events and by other Actors. Events are also modeled as Mealy machines whose transitions are triggered by a combination of simulator and Actor inputs. A benefit of this state machine-based conceptual model is the

ability to convert the model into code. The coded model can be verified using model checking tools such as Java Pathfinder (JPF) to gain further insight into the model. In the interest of space we will not describe the Java simulation framework developed for JPF model checking.

2.6.1 Actors

Choosing the core Actors is critical since modeling UAS-enabled WiSAR requires a level of abstraction that gives useful results without adding unnecessary complexity. After exploring several levels of abstraction, we selected a model that treats as an Actor any core decision-making element of the team, yielding the following Actors: parent search (PS), mission manager (MM), UAV operator (VeOp), video operator (VidOp), UAV operator GUI (VeGUI), video operator GUI (VidGUI), and the UAV. We deliberately chose to not model ground searchers, leaving this to future work.

The models of the human roles use specific states for communication. We describe these states once and then refer to them as a single communication state. Typically before a human communicates he or she receives some signal that the communication is being received. We model this as a POKE state. When communicating, an Actor model of a human enters the POKE state where it waits until it receives an acknowledgement. If the acknowledgement is not received then the communication does not occur. After the acknowledgement the human moves into a transmit (TX) state whose duration is based on the data being transferred. At the end of this transfer the human enters an end (END) state and outputs the transferred data to the receiver.

If the Actor model of the human receives a poke, then it responds with a busy or an acknowledge. If the human acknowledges the poke, then it enters the receive (RX) state. The human will not leave this state until the end communication input is received or until it decides to leave on its own. If one of these communications is interrupted before completion, then we consider that the data was not transferred. To better facilitate communication interruptions we only transfer a single piece of information per communication.

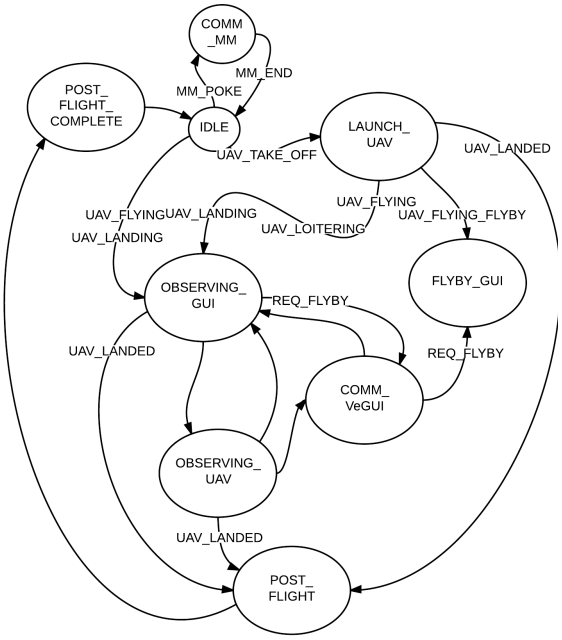

Figure 2.1: UAV Operator DiRG. Excludes transition output.

The next sections are dedicated to describing the Actor state machines with a generalized description of their relative transition matrixes. We omit several of the previously mentioned Actors in the interest of space.

UAV Operator (VeOp)

As illustrated in Figure 2.1, the VeOp Actor has the following states: *IDLE*, *OBSERVING_VeGUI*, *FLYBY_VeGUI*, *OBSERVING_UAV*, *LAUNCH_UAV*, *POST_FLIGHT*, and three communication states: *MM*, *VeGUI*, and *VidOp*. Initially the VeOp is idle. After receiving a new search command from the MM the VeOp constructs a flight plan using the VeGUI. When this is complete the VeOp will then launch the UAV. While in the launch state the VeOp is observing the UAV, when the UAV completes its take off the VeOp moves to observing the VeGUI. While the UAV is airborne the VeOp will continually move between observing the UAV and observing the VeGUI. The VeOp will respond to any problems that are noticed while observing the VeGUI or UAV.

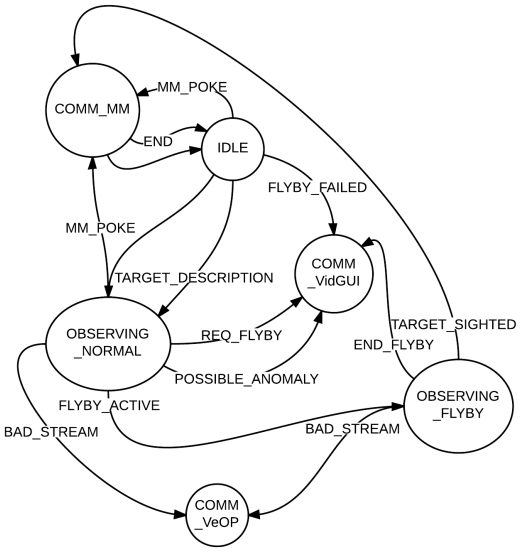

Figure 2.2: Video Operator DiRG. Excludes transition output.

During the flight the VeOp listens for input from the PS and VidOp. If there are flyby requests on the VeGUI, then the VeOp may choose to enter a flyby mode. This implies a high cognitive load on the VeOp while positioning the UAV over the specified anomaly. The VeOp will remain in flyby mode until the VidOp specifies, through the GUI, that the flyby is finished. During an operation the UAV will often land and take off multiple times. The post flight state represents the work necessary to get the UAV ready for flight, such as changing the battery.

Video Operator (VidOp)

As illustrated in Figue 2.2, the VidOp Actor has the following states: *IDLE*, *OBSERVING_VidGUI_NORMAL*, *OBSERVING_VidGUI_FLYBY*, and three communication states: *MM*, *VeOp*, and *VidGUI*. Initially the VidOp is idle. After receiving a target description and the search information the VidOp moves to normal GUI observation. While observing the VidGUI the VidOp watches for anomalies, each time an anomaly is visible the VidOp decides if the anomaly is seen. If it is seen the VidOp decides if it is an unlikely, possible, or likely sighting. This is done with probabilities related to the type of anomaly, true positive or false

positive. If the anomaly is classified as possible then the VidOp makes a validate sighting request for the MM. If the anomaly is a likely sighting then the VidOp requests a flyby from the VeOp.

When the VeOp begins a flyby request the VidGUI signals the VidOp to enter the flyby state. In this state the VidOp watches for the anomaly. Due to the nature of the flyby the VidOp can now make an informed decision about the nature of the anomaly, gaining a much higher probability of being correct. After deciding if it is the target the VidOp signals through the VidGUI that the flyby is finished. If the sighting is confirmed the VidOp reports to the MM, otherwise the VidOp returns to normal GUI observation.

Operator GUI (VeGUI)

The VeGUI Actor has two states: *NORMAL* and *ALARM*. The VeGUI communicates directly with the UAV and VidGUI Actors. The default function of the VeGUI is to observe the UAV. The VeGUI keeps internal variables of all the UAV and VidGUI data that it tracks, all of this data is available through observation of the VeGUI. If it detects an error with the UAV outputs such as low battery, no flight plan, low height above ground, or lost signal the VeGUI will enter the alarm state. This state indicates that there are visible warnings on the screen to alert the VeOp of the problem. The VeGUI listens for VeOp input or changes in the UAV output to signal that the problem has been dealt with before moving into the normal state.

UAV

The UAV Actor has the following states: *READY*, *TAKE_OFF*, *FLYING*, *LOITERING*, *LANDING*, *LANDED* and *CRASHED*. Initially the UAV is in the ready state. Upon command the UAV moves to take off for a specific duration and then to flying or loitering. The flying state is when the UAV is following a flight plan. The loitering state is when the UAV is circling a specific location. The UAV will automatically enter the loitering state after completing its flight plans. While airborne the UAV, upon command, moves to the landing state for a

specific duration before moving into the landed state. Once landed the UAV must be moved into the ready state before it can take off again.

The Actors in this model, thus far, represent a fairly high level of abstraction. Fortunately, the DiRG conceptual framework allows incremental extension of the models by adding lower levels of abstraction. This is accomplished by introducing *sub-Actors* into the model. We illustrate how the Actor/sub-Actor hierarchy can be used by describing two UAV sub-Actors. In these examples the sub-Actors receive all the same input as the parent Actor and all sub-Actor output is sent from the parent Actor.

The first UAV sub-Actor is the UAVBattery. It contains the following states: *INACTIVE*, *ACTIVE*, *LOW*, and *DEAD*. Initially the battery is inactive. The battery is assigned a duration and a low battery threshold. When the UAV receives the take off command the battery enters the active state. The batteries next state is set to low at time $current\_time + battery\_duration - low\_battery\_threshold$. When the battery enters the low state its next state is set to dead at time $current\_time + low\_battery\_threshold$.

A second sub-Actor is the UAVFlightPlan. This represents the flight plan flown by the UAV. The flight plan requires a specific amount of time to complete. The UAVFlightPlan has the following states: *NONE*, *ACTIVE*, *PAUSED*, and *COMPLETE*. Initially the flight plan is set to none. After the operator creates a flight plan using the VeGUI then the flight plan moves to active. During a flight the UAV may loiter, land, or flyby; this causes the flight plan to move to paused. When the UAV begins following the flight plan again it returns to active. After the UAV has flown the flight plan for the specified duration the flight plan enters the complete state.

The results discussed below include four other UAV sub-Actors: UAVHeightAboveGround, UAVSignal, UAVVidFeed, and FlybyAnomaly. Details are omitted in the interest of space.

2.6.2 Events

We used the Event Abstraction for several different elements of the UAS-enabled WiSAR problem. Adding this abstraction allows humans analyzing the system to give a set of inputs to the model and observe the consequences of the inputs.

The following Events were encountered in various UAS-enabled WiSAR field trials and represent a sample of interesting possible operating conditions for the Actors. In the interest of space we list these events while omitting their descriptions. NewSearchAOIEvent, TargetDescriptionEvent, TerminateSearchEvent, LowHAGEvent, LostSignalEvent, TruePositiveAnomalyEvent, FalsePositiveAnomalyEvent, and BadVideoFeedEvent.[1]

2.6.3 Asserts

As a general rule in model-checking, the more complex the model the more that can go wrong. Detecting flaws in the model is extremely valuable because such flaws trigger further evaluation. We present a case study in the next section that illustrates how the evaluation can identify things that need to change in the WiSAR process to avoid serious errors and perhaps failure to find the missing person.

Unfortunately, it can be challenging to differentiate between important flaws and coding bugs. To catch all errors, both flaws and bugs, we use Java Asserts. JPF automatically halts processing when it encounters a false assertion, allowing us to determine if the error is a bug or a flaw.

The model uses asserts in two ways. The first is detection of an undesired state. If an actor enters an undesirable state then an assertion halts the simulation. An example of this is the *UAV_CRASHED* state. The second deals with inputs. Many operations are sequential. They require a specific state and input before the next task can be performed. By looking at an Actors received inputs we are able to tell if an Actor is out of sync with the other Actors. An example of this is the *VeOp_TAKE_OFF* input for the UAV. If the UAV is

[1] HAG = Height Above Ground, AOI = Area of Interest

already airborne and it receives this input we know that the operator is out of sync with the UAV.

Asserts are critical to debugging and verifying of the model. We found that having too many asserts is preferable to having too few.

2.6.4 Case Study: Anomaly Detection

The scenario illustrated in figure 2.3 represents a portion of what should occur when the video operator believes a target has appeared on the video GUI. Each vertical swimlane represents an Actor/ DiRG. Periodically during a flight the UAV will fly over an anomaly. An anomaly can be either a false positive or a true positive, meaning that it is either the desired target or it is not. If the video operator believes that it is the target then a flyby request is made through the video GUI. This request is then made visible to the operator through the operator GUI. When the operator decides to perform the flyby request he signals this through the operator GUI and begins to manually direct the UAV to the location of the anomaly. While the operator is directing the UAV the video operator closely examines the video stream until the anomaly is visible again. The video operator then decides if it is a true target sighting or a false positive. The video operator communicates this to the operator through the video GUI. If it was a target sighting then the video operator passes the information to the mission manager who then passes it to the parent search. This high level view communicates the basic structure of the communication between the different actors.

2.7 Results

In this section, we first discuss model-checking results and then present insights from the modeling process.

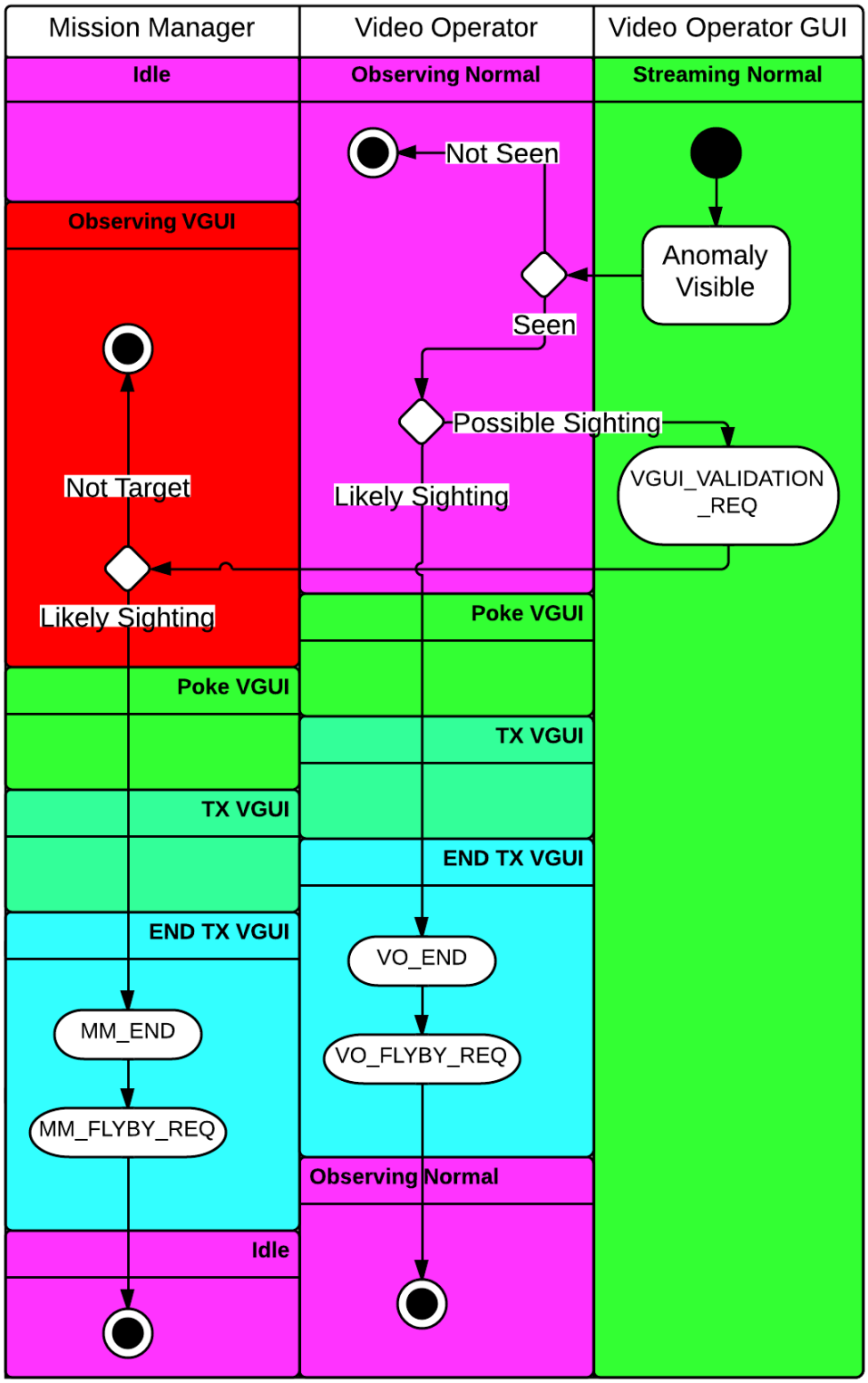

Figure 2.3: Anomaly Detection Model: Swim lanes represent actors. Arrows represent input/output. Colored sections represent actor states.

2.7.1 JPF Results

Model checking constructs an exhaustive proof, by enumerating every reachable state of a system, to establish a property. The JPF model checker uses native Java as the modeling language to describe a system, and implements a custom virtual machine to systematically explore the reachable state space of the compiled Java program. The reachable state space of a sequential model is trivial to enumerate but the reachable state space grows exponentially with the level of non-determinism (i.e., random choice).

The source of non-determinism in the WiSAR model is in timing. The model defines time bounds for different events and tasks in the system. When the model is run, it non-deterministically chooses delays from within those bounds. For example, the time to land the UAV is a uniform random variable bounded between 60 and 1,800 time units. In the WiSAR model, there are 69 different tasks or events with non-deterministic durations. As it is not feasible to enumerate the entire reachable state space of such a large model, this work implements several standard heuristics to limit the non-determinism: minimum(maximum) time only; minimum and maximum times only; and minimum, maximum, and mean times only. As expected, the first heuristic using only the minimum(maximum) time bound results in a sequential model with no non-determinism.

The total number of meaningful lines of code, code within methods, for the WiSAR model is 3,804. JPF is able to analyze the model using the minimum heuristic in negligible time only enumerating 124 states on a MacBook Air with 4GB of memory and an Intel Core I7 1.6 Ghz processor. The maximum heuristic also has negligible time but fewer behaviors with only 16 states. The difference in states is due to the UAV not needing to land several times to recharge its battery.

The minimum and maximum bounds heuristic in JPF generates an important and interesting result. JPF finds a combination of task durations that result in an infinite loop using the heuristic. The same infinite loop can be recreated outside of JPF by repeatedly running the model over and over again as random trials until one of the trails goes into an

infinite loop. The power of using JPF is that it finds the infinite loop every time without the need for the random trials. The root cause of the infinite loop was a flawed communication protocol that implicitly relied on specific delays in the interaction. The protocol has since been corrected and JPF verifies the model to now terminate under all combinations of minimum or maximum delays: 3,911 states in 6 seconds of running time.

The greatest number of states enumerated by JPF comes from using the minimum, maximum, and mean heuristic. For all 69 points of non-determinism in the WiSAR model, JPF exhaustively considers 3 distinct values for each point, and checks every possible combination. JPF enumerates 51,344 states in around 52 seconds using the heuristic. None of the states violate the current set of assertions in the model and the model terminates under all the duration combinations. The jump in the number of states between the minimum and maximum bounds heuristic and this heuristic illustrates the exponential state explosion inherent in model checking.

The JPF model checking does not prove the WiSAR model is the desired model or even a correct working model. There is considerable research yet to be completed in writing the system level requirements of WiSAR and then having JPF verify each of those requirements. If JPF finds a violation on any given requirement, there is still considerable work to determine if the requirement is correct (i.e., really what is desired from the system), if the model has a bug, or if the protocols in the model are fundamentally flawed and not able to implement the requirement. These topics are future work for the WiSAR model.

2.7.2 Lessons from Modeling

One of the goals of using model-checking with UAS-enabled WiSAR is to discover problems and opportunities with the structure of the organization. This section presents several important lessons for UAS-enabled WiSAR that were obtained through the modeling exercise.

First, while modeling the VidOp it became apparent that there was a problem with the organization. This problem occurs because, while the VidOp is marking an anomaly,

the video feed continues to run. This means that it is possible that the VidOp may miss detecting the target. If the VidOp pauses the video, the feed falls behind the live video feed which makes flyby requests more expensive because the UAV will have to backtrack to the anomaly sighting. We analyzed why this problem was not discovered in the WiSAR field trials. The answer is that the field trials included multiple video feeds with multiple observers, a condition that is not likely to occur in a resource-limited search. A lesson from this observation for WiSAR is that technology needs to be developed that allows the WiSAR team to manage this problem. A more general lesson is that the modeling and model-checking process uncovered a potential problem before it appeared in practice.

A second lesson was learned when performing model-checking of a flyby. Our model showed two problems, resuming a flight plan after a flyby and needing to keep a list of flyby requests. We solved this in the model by adding visible queues to the VeGUI and VidGUI and allowing the VeGUI to store multiple flight plans. As before, we analyzed why these problems were not discovered in the WiSAR field trials. The answer is that these problems did occur but were not documented. The lesson for WiSAR is that the VeGUI and VidGUI need new features to support real searches. A broader lesson is that the modeling exercise can be used to not only detect problems but specify the requirements for fixing them.

2.8 Conclusions and Future Work

We have presented DiRGs expressed as Mealy state machines for the purpose of modeling WiSAR in a way that will give insight into improving the WiSAR processes. In addition we have coded these Mealy machines in Java and performed model-checking using the JPF tool for the purpose of gaining even more insight into our model by running it. This contrasts with previous modeling attempts. Results show that additional insight was gained, and that it was possible to introduce new processes into the model and see the effect of those changes.

The result of the modeling and model checking processes was the detection of problems within WiSAR that were not seen during other analysis, or were seen but not documented.

The processes also gave insight, and in some cases specifications, for fixing the encountered problems.

Future work will use explicit declarations to formalize Actor states and transition matrixes. By formalizing these properties it will be possible to find transition errors immediately; it will also make it possible to compare the model with the model documentation for accuracy. This may also make it possible to export the state machine into other model checkers.

We also plan to add sequential constraints to the model using Actors. These Actor will embody the desired sequence of tasks and transitions and will throw assertions if a sequence is not executed in the desired order. This will help us verify that the model is following the desired behaviors.

Acknowledgment

The authors would like to thank Neha Rungta of NASA Ames Intelligent Systems Division for her help with JPF and Brahms. The authors would also like to thank the NSF IUCRC Center for Unmanned Aerial Systems, and the participating industries and labs, for funding the work. Further thanks go to Jared Moore and Robert Ivie for their help coding the model and editing this paper.

Chapter 3

Workload Metrics

The paper presented in chapter 2 was written before the workload concepts and modeling language semantics had settled into their current form. Jared et al. [21], in concert with this work, published a paper that expands on the concepts in chapter 2 but is also slightly outdated. Thus we feel it expedient to summarize those aspects of the language which are critical to our metrics before we present the metrics themselves. To assist in this effort we have prepared a simple scenario which includes a partial model and illustrations of the DiRG, DiTG, and labeled state transition system.

3.1 Example Scenario

In this scenario there are two people, Alice and Bob. Alice is standing next to Bob listening to a friend on her cell phone. Bob suddenly remembers that he wants to ask Alice out. Not noticing that Alice is listening to her phone Bob starts to ask Alice on a date. When this happens Alice looks at Bob and points to her phone, signaling to him that she is on the phone. Bob stops talking and decides between waiting for her to finish or walking away. Eventually Bob walks away.

3.1.1 Actors

From the scenario above we chose to create three Actors: A) Alice, B) Bob, and C) Cell Phone. Actors represent any aspect of the system that has state. An Actor can be anything, in our example we have two humans and a cell phone. We could also create a sub-Actor

which is part of a larger Actor, such as Bob's hair, and give it states like messy or combed. Actors can also be very abstract or very detailed. The more states an Actor contains, the more expressive it becomes.

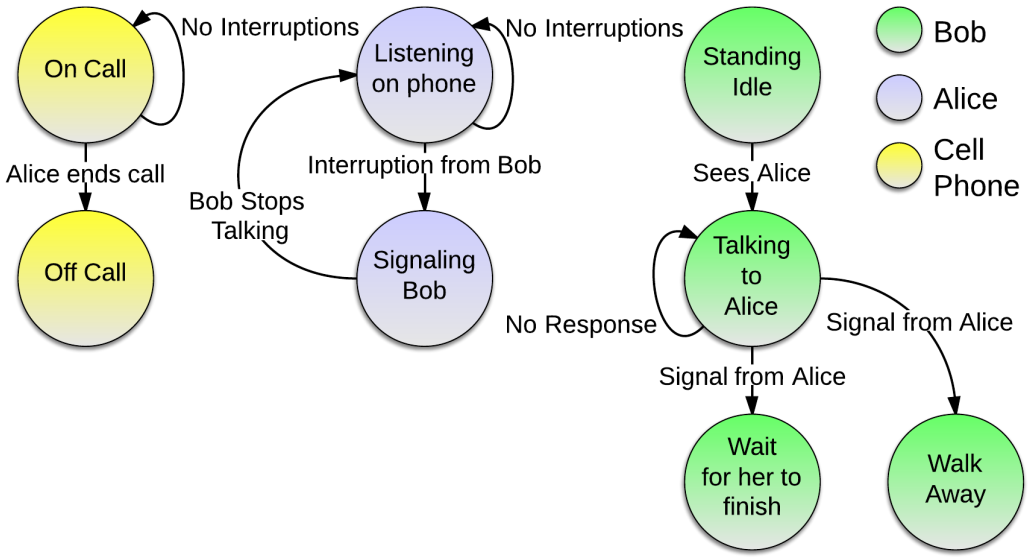

Figure 3.1: Directed Role Graph for Alice and Bob scenario

3.1.2 DiRG

We express an Actor as a Directed Role Graph (DiRG). A DiRG represents how an Actor is allowed to flow between states. Figure 3.1 shows the DiRGs for both Alice and Bob as a state transition system. We see that Alice is initially in a *Listening on phone* state while Bob is in the *Standing idle* state. Alice can either stay in the *Listening on phone* state, shown by the looping transition, or move into the *Signaling Bob* state. Once in the *Signaling Bob* state her only choice is to stay there forever or to return to the *Listening on phone* state. Individually these DiRGs are of little value, together they begin to express the larger system. We see from the labels on the DiRG that Alice and Bob are interacting with one another and

influencing the transitions of the other. Before we discuss Actor transitions we must first define the inter-Actor relationship that allows Actors to influence one another.

3.1.3 Channels

We define these inter-Actor connections as Channels. A channel is a uni-directional communication medium which allows an Actor to send information to another Actor. Each Channel is composed of a source Actor, a target Actor, and a type. The source Actor sends information as *output*, the target Actor receives the information as *input*, and the type specifies which communication medium is being used. When describing the metrics we sometimes refer to the Channel source as the input source and the Channel target as the output target. In the case of Alice and Bob we use audio, visual, and manual Channels[32], the case study in chapter 4 also uses a Data channel that represents network communication.

We also designate that each Channel can represent multiple layers of communication. To show this we will use the visual channel from Alice to Bob as an example. We can express Alice's *output*, Bob's *input*, as two different layers on the visual channel, one for Alice's body language, and another for her facial expressions. This allows us to explicitly set how much data is being sent over the channel, it also allows us to express multiple visual inputs for Bob without creating a *channel conflict*. A *channel conflict* occurs when an Actor is receiving input from two or more channels of the same type. In our example scenario Alice is listening to her cell phone which uses an audio channel. At the same time Bob is talking to Alice on a different audio channel. Because Alice is receiving *input* on multiple audio channels she has an audio channel conflict.

3.1.4 DiTG

To express a systems Channels we use a Directed Team Graph (DiTG)[21]. The DiTG defines all the channels that exist between the Actors within the system. Figure 3.2 shows the DiTG for our Alice and Bob scenario. From the figure we can see that Alice has two channels to

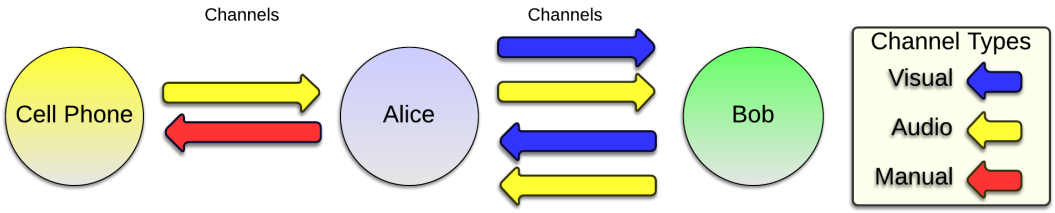

Figure 3.2: Directed Team Graph for Alice and Bob scenario

Bob, an audio and a visual and an audio channel to her cell phone. We also see that Bob has two channels to Alice, an audio and a visual. Lastly, the cell phone has an audio channel to Alice. While the simplicity of this scenario makes it difficult to see the value of the DiTG, by combining the DiTG with the DiRG we have effectively constrained the Actor behavior and communication for the entire system. With the constraints in place we are ready to define the behavior of the system, which we do with Transitions.

3.1.5 Transitions

Transitions represent an Actors behavior. Transitions tells us about an Actors state, what caused the Actor to change state, and how that change effects the system. Transitions are composed of a start state, an end state, a set of input equations, a set of outputs, a duration, and a priority. The Transition start and end states are the states of the Actor and must not violate the DiRG.

The Transition input equations are used to determine if the transition is enabled. Each equation is composed of a source value, a predicate, and an expected value. The source value is obtained from one of two sources, Channels or Memory. Actor memory is an internal variable that allows an Actor to store and retrieve data. For predicates we use equal to, less

than, greater than, etc. The structure of the input equations allows each equation to evaluate to a simple true or false. If all input equations evaluate to true then the Transition is enabled.

The Transition outputs contain all output generated by the transition as a set of target value pairs. The target is the Channel or Memory variable that will receive the designated value. The Transition duration is a range which represents the minimum and maximum number of time steps that a transition will remain *active* before it *fires*. When an Actor decides to transition it selects an enabled transition to become *active*. While a Transition is *active* the transition outputs are sent out but the Actor does not change state until the Transition *fires*. We sometimes refer to inputs and outputs as being *active*, this implies that they are coming from an *active* Transition and that the value is not null. The final Transition element is priority. Priority reflects how important a Transition is to the Actor relative to the other Transitions. If multiple Transitions are enabled the Actor will choose the Transition with the highest priority.

3.1.6 Labeled State Transition System

The Model Abstraction Framework modeling language allows each of these concepts to be expressed in a model A. The model is then converted to a labeled state transition system that is sent to the simulator for metric collection. We chose the labeled state transition system because the state transition system lends itself well to model checking while the label allows us to add specific data which relates to workload. In our case the label translates directly to the transition input equations, outputs, duration, and priority.

In figure 3.3 we can see that every transition has a label. These labels are used by the simulator to determine if the transition is enabled, what channels will become *active*, and how long those channels should remain *active* before the state changes. To make this clear we will describe $Label_A12$ of Alice's transition from the *Listening on phone* state to the *Signaling Bob* state. From figure 3.3 we see that when Bob is speaking to Alice there is an audio channel that is *active* (opaque yellow arrow). The description on the transition implies

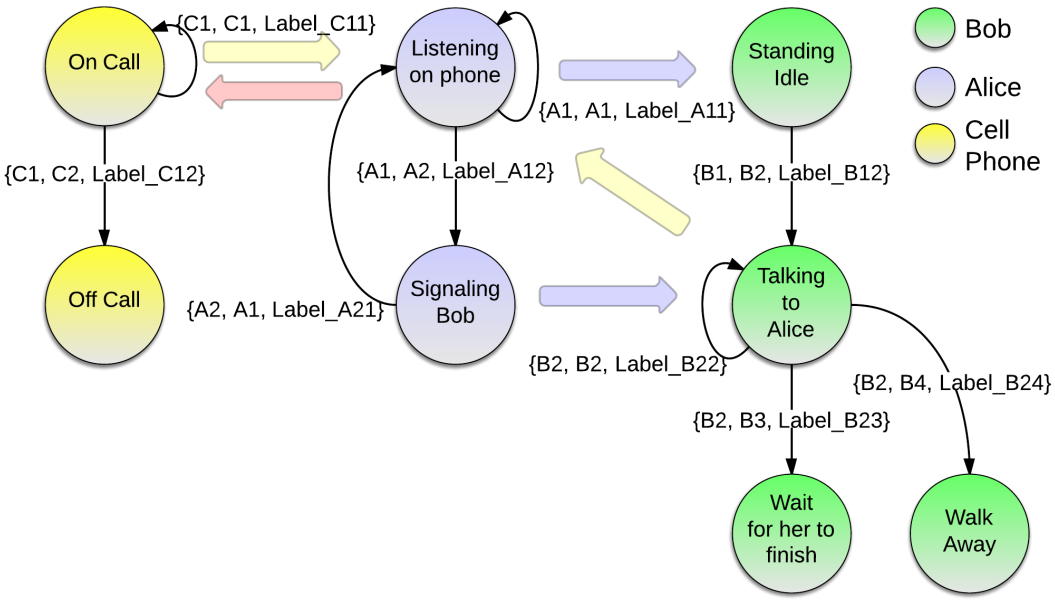

Figure 3.3: Labeled State Transition System for Alice and Bob scenario

that Bob's talking has somehow triggered the transition, to represent this in the label we create two input equations. Audio channel from Bob does not equal null. And audio channel from Cell Phone does not equal null. This means that when Alice is listening to her cell phone if she hears Bob at the same time then this transition becomes enabled. If she chooses to follow the transition then the transition becomes *active* and the transition outputs will become active. In this case the output is a signal on the visual channel from Alice to Bob. We will make the duration 5 seconds with a priority of 1.

Now that we have a basic understanding of the data that the simulator has access to during the simulation we can now discuss how we translate this data into meaningful metrics. We will first discuss our baseline workload metric. Afterwards we will describe the two workload metrics we created as part of this work.

3.2 Adapted Wickens' Metric

Since we are interested in metrics that reveal human workload we have chosen to replicate Wickens' computational model [32], shown in equations 3.1-3.2, using data gathered from the Model Abstraction Framework. Wickens' model is a measure of resource demand and overlap that has been shown to predict performance degradation, making it a good baseline metric for evaluating consistency with known high workload events and for comparison with the new metrics presented later in the chapter.

Wickens' computational model is based on the concept of tasks, where a task is some arbitrary unit of activity. Wickens' model calculates the resource interference between two tasks, a value that is closely related to mental workload [32]. The problem with using this computational model as a metric within the Model Abstraction Framework is the difference in the modeling paradigms. In this work we abstract the notion of a task, relying instead on states and transitions to infer activity. The challenge is to apply the task-based abstraction of Wickens' model to the state/transition based abstraction of the Model Abstraction Framework.

$$W_{Wickens}(T_1, T_2) = R_{Demand}(T^1, T^2) + R_{Conflict}(T^1, T^2) \tag{3.1}$$

$$R_{Demand}(T^1, T^2) = T^1_{demand} + T^2_{demand} \tag{3.2}$$

$$R_{Conflict}(T^1, T^2) = \sum_{x=1}^{dimensions} \begin{cases} 1 & T^1_x = T^2_x \\ 0 & otherwise \end{cases} \tag{3.3}$$

Wickens' model $W_{Wickens}$, equation 3.1, calculates the resource interference between two tasks, represented as T^1 and T^2, through the use of two components: resource demand R_{Demand} and resource conflict $R_{Conflict}$. Resource demand is a subjective measure of the cognitive resources required by a task. In equation 3.2 we see that the resource demand of tasks T^1 and T^2 is calculated by summing the resource demand of both tasks. To keep

the model simple and intuitive Wickens limits T_{demand} to a range of 0 to 2, where 0 is an automated task and 2 is a difficult task. Thus R_{Demand} will always be between 0 and 4.

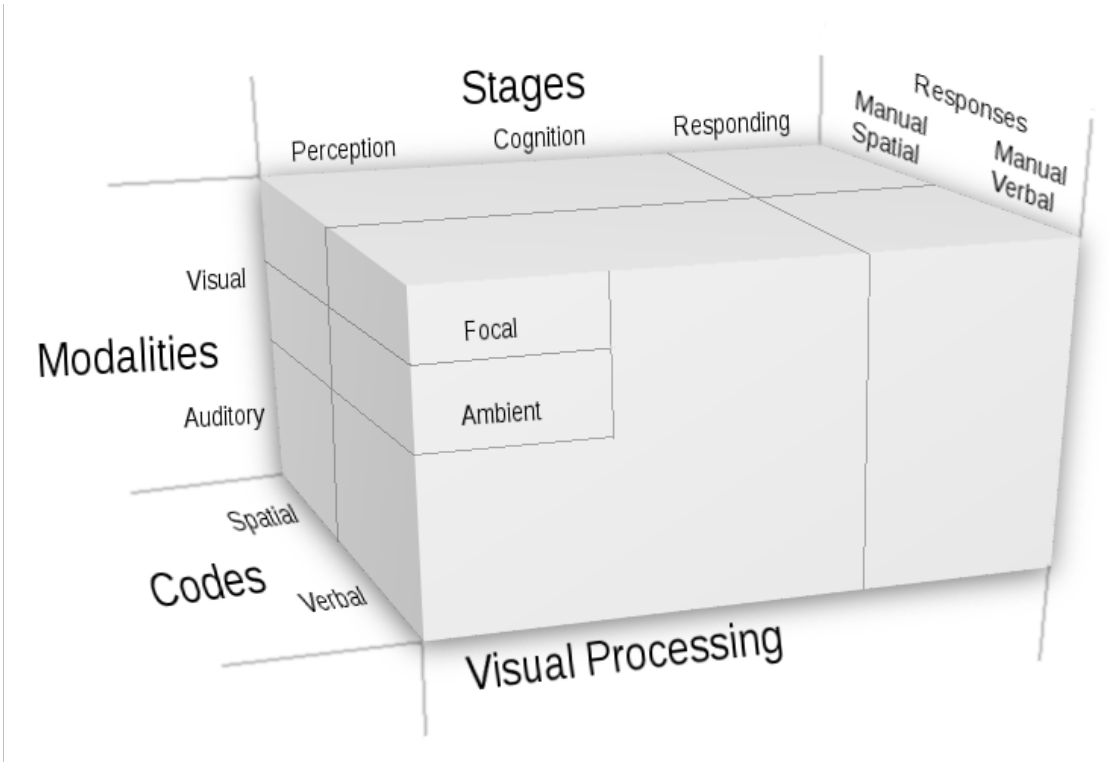

Figure 3.4: Multiple Resource Theory Dimensions [32]

For resource conflict two tasks are considered to have conflicting resources when they share resources within one of the dimensions illustrated in figure 3.4. In equation 3.3 when T_x represents the task resources for dimension x, then $R_{Conflict}$ is calculated by taking the sum of the total number of resource conflicts between two tasks. Since there are only four dimensions $R_{Conflict}$ will always be between 0 and 4. For example if $Task_A$ required a person to listen for distinct sounds while $Task_B$ required listening to a conversation, the resource conflict would be 2. One in the *Stages* dimension for perception, and another in the *Modalities* dimension for auditory. Since one deals with spacial sound and the other verbal sound no resources are shared in the *Codes* dimension. As neither task dealt with visual perception there are no conflicts in the *Visual Processing* dimension.

3.2.1 Actor Load

To mimic resource demand within the Model Abstraction Framework we introduce the notion of Actor Load. Actor Load represents an abstraction of the load an Actor is under while in a specific state. Similar to resource demand in Wickens' model, Actor Load is a subjective value assigned by the modeler to each state in the system. To make this value more intuitive and simple we have abstracted it into an integer value ranging from 0-4. An Actor Load of 0 represents little to no load on the actor. These are automated or transitional states where the Actor is idle or has minimal contact with the system. An Actor Load of 4 represents simultaneously performing multiple high difficulty tasks. These are states where an Actor is pushed to the limit of their cognitive capabilities. Any value between 0 and 4 is some combination of task difficulty and the number of tasks being performed.

3.2.2 Determining Dimensional Conflicts

To calculate the dimensionality of resource conflicts we need a way to determine when multiple tasks are being performed. Since the Model Abstraction Framework does not define specific tasks we must find another method to approximate when multiple tasks are being performed. We accomplish this by making the assumption that if an Actor has input from multiple sources then multiple tasks are being performed.

$$D_{Stages}(\Sigma, \Lambda) = \begin{cases} 1, \Sigma_{sources} > 1 & \text{Multiple Input Sources} \\ 1, \Lambda_{targets} > 1 & \text{Multiple Output Targets} \\ 0, otherwise & \end{cases} \tag{3.4}$$

With this assumption we are ready to calculate dimensional conflicts as illustrated in Figure 3.4. If Σ represents the set of all inputs and Λ the set of all outputs then we calculate the dimensional conflicts as follows. For the Stages dimension (Perception, Cognition, Response), equation 3.4, we check to see if there are multiple sources of *active* input or

multiple output targets, not including memory inputs and outputs. If multiple sources or targets exist then the stages dimensionality (D_{Stages}) has a conflict.

$$D_{Modalities}(\Sigma, \Lambda) = \begin{cases} 1, \Sigma_{active}^{audio/visual} > 1 & \text{Multiple Active Inputs} \\ 1, \Lambda_{active}^{audio/visual} > 1 & \text{Multiple Active Outputs} \\ 0, otherwise \end{cases} \tag{3.5}$$

For the Modalities dimension (Audio, Visual), equation 3.5, we check if there is more than a single *active* channel, input or output, for a specific channel type. If there is then the modality dimensionality ($D_{Modality}$) has a conflict.

$$D_{Codes}(\Sigma, \Lambda) = \begin{cases} 1, \Sigma^{audio} + \Lambda^{audio} > 1 \\ 1, \Sigma^{visual/manual} + \Lambda^{visual/manual} > 1 \\ 0, otherwise \end{cases} \tag{3.6}$$

For the Codes dimension (Spatial, Verbal), equation 3.6, we check that the total number of audio inputs and outputs is greater than 1 or that the total number of visual inputs, visual outputs, and manual outputs is greater than 1. Since we do not specify different types of audio output we assume that all audio is verbal, likewise we assume that all visual/manual input and output is spacial. If either check succeeds then the codes dimensionality (D_{Codes}) has a conflict.

$$D_{VP}(\Lambda) = \begin{cases} 1, \Lambda_{target}^{manual} > 1 & \text{Multiple Manual Output Targets} \\ 0, otherwise \end{cases} \tag{3.7}$$

For the Visual Processing dimension, equation 3.7, we check if there is more than one target for manual outputs. We do not check inputs as we have no way of distinguishing if a visual channel is focal or ambient. Instead we assume that manual outputs require visual focus, allowing us to determine if visual processing is split between multiple tasks. If we have

manual output to multiple targets then the visual processing dimensionality (D_{VP}) has a conflict

3.2.3 Adapted Wickens' Model

Using the Actor Load and the adapted dimensionality conflicts we are now in a position to replicate $W_{Wickens}$ within the Model Abstraction Framework. As the model values are calculated differently than Wickens' model we will refer to this model as an adapted Wickens' model $W^{Adapted}_{Wickens}$. The equation 3.8 is as follows: given an actors State S we can obtain Actor Load $S_{A_{Load}}$ as well as the inputs S_Σ and outputs S_Λ needed to calculate the four dimensionality values. We obtain $W^{Adapted}_{Wickens}$ by adding $S_{A_{Load}}$ to the sum of the four dimensionality values. This ensures that $W^{Adapted}_{Wickens}$ is between 0 and 8, just like $W_{Wickens}$.

$$W^{Adapted}_{Wickens}(S) = S_{A_{Load}} + D_{Stages}(S_\Sigma, S_\Lambda) + D_{Modalities}(S_\Sigma, S_\Lambda) + D_{Codes}(S_\Sigma, S_\Lambda) + D_{VP}(S_\Sigma, S_\Lambda) \tag{3.8}$$

Now that we have a baseline metric adapted from the related work to use as a baseline for consistency it is now time to describe two new workload metrics.

3.3 Resource Workload Metric

The resource workload metric $W_{Resource}$ attempts to measure the resource load an Actor is experiencing via inter-actor communications and memory access for each time step of the simulation. The concept is based off of the cognitive workload category presented by Jared et al. [21]. For example, when Bob is talking to Alice she is processing input on multiple audio

channels while also accessing any relevant memory. Once Alice is alone again her resource workload will have decreased as she is only processing a single audio channel.

$$W_{Resource}(S^{Current}) = S_{ActorLoad} + ChannelConflicts(S_\Sigma) + ResourceLoad(S_\Sigma) + ResourceLoad(S_\Lambda) \tag{3.9}$$

$$ChannelConflicts(\Sigma) = \sum_{ChannelTypes} \begin{cases} 1 & \sum \Sigma^{ChannelType} \in \Sigma > 1 \\ 0 & otherwise \end{cases} \tag{3.10}$$

$$ResourceLoad(IO) = \sum IO_{Active} + \sum IO_{LayersUsed} + IO_{MemoryUsed} + \sum IO_{Active}^{ChannelTypes} \tag{3.11}$$

We see from equation 3.9 that resource workload $W_{Resource}$ is composed of three sub-metrics: channel conflicts, resource load, and Actor Load. Given the current state $S^{Current}$ we can extract the Actor Load $S_{ActorLoad}$, the set of inputs S_Σ from the available transitions, and the set of outputs S_Λ from the active transition. We can then pass these values to equations 3.10 and 3.11 to obtain the required sub-metrics.

Channel conflicts occur whenever more than one active channel shares a type, such as an Actor receiving input on multiple audio channels. Given a set of input channels (Σ) equation 3.10 calculates the total channel conflicts by counting the number of $ChannelTypes$ that occur more than once in the set of all inputs. Since we currently only allow visual and audio input for human Actors this value ranges from 0 to 2.

Resource load attempts to quantify the load being placed on the Actor's resources. We break the resource load into two parts, input and output, the final result being the sum of both parts. Each part is calculated using equation 3.11. The equation states that given a set of inputs/outputs IO the respective resource load is the sum of the number of active channels, number of layers used, number of memory objects used, and the number of active

channel types. Actor Load is the same metric that is used in the adapted Wickens' metric and is included here because it is a direct reflection of the modelers belief of what the resource load is during this state.

3.4 Decision Workload Metric

The decision workload metric attempts to measure the complexity of the decision making process by measuring the size of the decision space for each time step of the simulation. The concept is based off of the algorithmic workload category presented by Jared et al. [21]. For example, when Bob begins talking to Alice the added input enables an additional transition. This means that Alice must now choose to keep trying to listen to the phone while Bob talks or signal Bob to stop talking, thus increasing her decision workload. When Bob stops talking Alice's decision workload decrease as there is less input to process and fewer enabled transitions to choose from.

$$W_{Decision}(S^{Current}) = S_{EnabledTransitions} + S_{\Sigma^{Read}} + S_{\Lambda^{Set}} + DurationComplexity(S^{Current}) \tag{3.12}$$

To calculate the decision workload metric we use equation 3.12 which is composed of four sub-metrics: decision complexity, input complexity, output complexity, and duration complexity. The decision complexity represents how many options were available to choose from and is calculated as the number of enabled transitions $S_{EnabledTransitions}$ for the current state. In the case of Alice and Bob when Alice signals to Bob that she is on the phone Bob can choose to keep talking, stop talking and wait for Alice to finish, or walk away; thus Bob has a decision complexity of 3 while Alice is signaling.

The input complexity represents the number of inputs that the Actor must process to make a decision. It is calculated by summing the total number of inputs read by transitions $S_{\Sigma_{Read}}$, this includes active channels and memory variables. Using the previous example,

when Bobs decision complexity was three his input complexity could have been two, one from Alice's signal and another from a memory value representing how cute Alice is to Bob. The output complexity is the same as the input complexity only the value is calculated from the total number of outputs $S_{\Lambda^{Set}}$ that can be set.

$$DurationComplexity(S) = \log \frac{max(S_{Duration})}{60} \tag{3.13}$$

The duration complexity represents complexity that is related to the size of the task(s) being performed. The underlying assumption being that difficult tasks take more time to complete than simple tasks. While this assumption is not always true it is possible that some future variation of this metric may prove useful. Due to the high variance and low trust associated with this metric we decided to normalize the duration complexity using equation 3.13. By assuming that durations are in seconds and that any duration that is less than a minute has zero duration complexity we can convert durations into minutes and then use a logarithmic scale for the final value. While the diminishing returns of this model allow us to show the duration complexity within the context of our other metrics it does not account for human fatigue and other factors associated with long running tasks. We leave it up to future work to determine the usefulness of duration complexity and how it should be calculated.

Chapter 4

Case Study: UAS operating in the NAS

To analyze the workload metrics presented in the previous chapter we chose to perform a case study by modeling the integration of a UAS into the National Airspace System. First we will describe the scenario that we modeled and the assumptions required to make the model work. Then we will present an analysis of the workload metrics gathered using the Model Abstraction Framework.

4.1 Model Scenario and Assumptions

Since we would eventually like to correlate these workload metrics with real human workload our goal was to model a UAS that presented high and low workload periods similar to those observed during the WiSAR flight tests. Such as creating fight plans, monitoring normal flight, handling active alarms, introducing increased autonomy, and handling operator interruptions. This will allow us to check that the results are generally consistent with a known workload profile. The model we chose was the integration of a UAS into the National Airspace System.

To help illustrate this model we have created a DiRG that shows the states and data flow for each Actor. Figure 4.2. A matching DiTG for the model showing the Actor communication channels can be seen in figure 4.1.

An Unmanned Aerial System (UAS) plans to operate within the National Airspace System. The UAS is composed of a server (UAS Server), an Unmanned Aerial Vehicle Operator (UAV Operator), and an Unmanned Aerial Vehicle (UAV). The UAV Operator controls the UAS Server through a graphical user interface (UAS GUI) that in turn controls

the UAV, as illustrated in the DiTG shown in figure 4.1. The UAV will take off and land at an airport serving both manned and unmanned aerial vehicles. The UAV Operator is located at this airport and visually monitors the takeoff and landing of the UAV. UAV state is continuously shown on the UAS GUI.

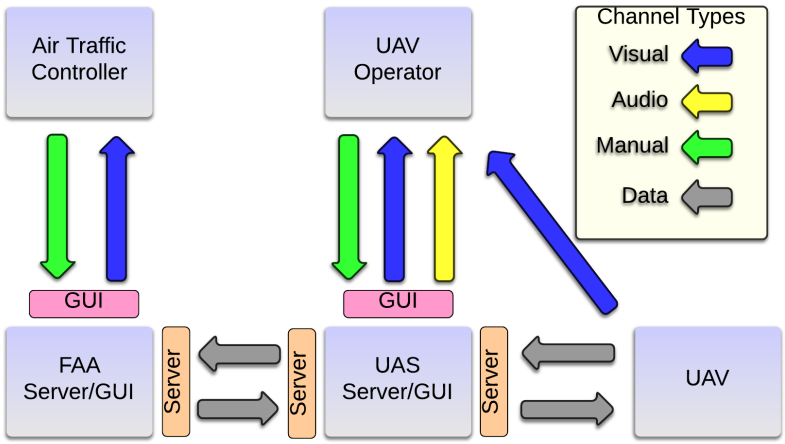

Figure 4.1: DiTG: UAS integration into the NAS Model

There is an FAA Server that provides real-time notice to airmen (NOTAM) information. The UAS Server connects to this server, receives the NOTAMs and displays them on the UAS GUI. Figure 4.1. The FAA system also allows the UAS Server to file flight plans. Filed flight plans are automatically checked for simple conflicts such as crossing NOTAMS or duplicate takeoff/landing times. If there is a conflict the FAA Server flags the flight plan for an Air Traffic Controller (ATC) and displays these requests on an FAA GUI that is monitored by the ATC. The ATC then approves or denies flight plans, using the FAA GUI, at their leisure. This approval/denial then becomes available to the UAS Server, which displays it on the UAS GUI as illustrated in the DiRG shown in Figure 4.2.

The FAA Server also provides radar information to the ATC through the FAA GUI. The ATC uses this information to spot potential collisions with UAVs. If a potential UAV

collision is detected the ATC creates an emergency NOTAM in the region of conflict. This emergency NOTAM is then sent to the UAS Server, which displays the emergency NOTAM on the UAS GUI. If the UAV is in or near the emergency NOTAM it must change course to immediately evacuate/avoid the NOTAM. This can be done automatically by the UAV or manually by the UAV Operator. Once the UAV has finished avoiding the emergency NOTAM it enters a loiter state. The UAV Operator is then required to change the flight plan before the UAV will leave the loiter state. Figure 4.2.

The UAV also has a radar that can detect nearby objects. This information is displayed on the UAS GUI and if the UAV Operator detects a potential collision they will begin the deconfliction procedure that requires changing the current flight plan. Once the UAV has landed the scenario is considered complete.

4.1.1 Assumptions

Given our lack of domain knowledge regarding the National Airspace System we have had to make a number of assumptions in order to achieve a high level of abstraction. The assumptions required for the model to perform as designed are:

- The UAV has an unlimited flight time, never loses contact with the UAS Server, can takeoff and land without incident, has accurate GPS data, and is non line-of-sight.
- The UAS Server/GUI never loses connection to the FAA Server, always has instant communication with the UAV and FAA Server, has no bugs, can create flight plans, can detect NOTAMs on the flight plan, can automatically direct the UAV out of an emergency NOTAM, displays radar information from the UAV, and never goes down.
- The UAV Operator detects all warnings displayed on the UAS GUI, generates flight plans that do not touch NOTAMS, can always deconflict the UAV, and never gets fatigued.

- The FAA Server/GUI distributes NOTAMS and automatically detects if a flight plan needs to be approved by the ATC.
- The ATC detects all information displayed on the FAA GUI, can add NOTAMS with the FAA GUI, always adds NOTAMS correctly, and approves all flight plans.

4.2 Model Results

We created three scenarios by adding small variations to the model. The first scenario involved an uneventful flight. The second scenario involves the UAV Operator manually avoiding an emergency NOTAM. The third scenario involves the UAV automatically avoiding an emergency NOTAM. The second and third scenarios also involve a possible UAV collision and the ATC adding a NOTAM onto the UAV flight path. Additionally we added a high workload scenario that involves the UAV Operator being interrupted by another person while deconflicting the UAV. The respective results are shown in figures 4.3-4.9.

For each scenario we have four charts depicting the UAV Operator workload and four charts depicting the ATC workload; we currently ignore the workload for non-human Actors. The first three charts represent the resource input, resource output, and decision workload. The last chart shows the combined total workload next to the adapted Wickens' model. For each chart the y axis represents the workload value while the x axis represents Actor state for the given delta time. What this means is that the x axis cannot be seen as a normal timeline since each time step represents an arbitrary amount of time. Instead this should be viewed as a progressive change in Actor workload for each time step where a transition was fired.

The results we obtained from the Model Abstraction Framework are very encouraging due to the general consistency with the expected workload. Despite a few unexplained anomalies, example in figure 4.6, and some unusual workload spikes, shown in figure 4.7, we can see that generally the Actors have more workload when performing active tasks and less workload for simple tasks. A clear example of this is the UAV Operator workload in figure 4.3. The workload spikes while creating the flight plan then remains low while monitoring the

UAV GUI for information. When that information is received we see a small spike followed by a prolonged workload spike while the UAV Operator is deconflicting the flight plan.

4.2.1 Inter Actor Comparison

When comparing the UAV Operator and ATC baseline workload, shown in Figures 4.3 and 4.4, the relationship between the UAV Operator and the ATC becomes very clear. For this scenario the UAV Operator creates a flight plan for the mission. When the UAV Operator finishes the flight plan it is sent to the FAA Server where it is flagged to be approved by the ATC. The ATC then performs the approval and returns back to normal operations. The UAV Operator receives the approval and begins normal mission operations that increase the workload. This is reflected beautifully in the Actor workload. The UAV Operator workload spikes during flight plan creation then flat lines while waiting for a response. Once the UAV Operator receives the response their workload spikes again as they perform the mission. The ATC workload is just the opposite, spiking while approving the flight plan and returning to normal afterwards. The comparison of the other scenarios show the same correlations.

We also see the effects of the UAV Operator and ATC relationship in the *manual avoid* scenario, shown in Figures 4.5 and 4.6, when the delayed action of the UAV Operator causes the ATC to remain in the *avoid conflict* state for an abnormally long period of time. By looking at the Actor state we see that the UAV Operator was already in the process of deconflicting the UAV and was not able to immediately respond to the ATC emergency NOTAM. The *auto avoid* scenario, shown in Figures 4.7 and 4.8, shows an improvement in this regard as the UAV automatically avoids the emergency NOTAM while the UAV Operator is busy. This example demonstrates how the workload modeling allows us to detect problems using the model and then analyze fixes to those problems.

4.2.2 Intra Actor Comparison

At first glance the UAV Operator workload between the *manual avoid* scenario, Figure 4.5, and the *auto avoid* scenario, Figure 4.7, appear almost identical. This is good as these scenarios are nearly identical. Upon closer examination we see that when the UAV Operator has to manually avoid the emergency NOTAM there is a prolonged increase in resource input workload, an extra spike of resource output workload, and a noticeable spike in decision workload once the UAV Operator begins the process of avoiding the NOTAM. This results in a significant workload spike compared to the more stable workload seen in the *auto avoid* scenario. Another less noticeable difference is that the manual emergency NOTAM avoidance takes longer to complete than the auto avoid feature, something that is critical in these types of situations.

To make this simple model a little more interesting we decided to add an interruption to the *auto avoid* scenario. See Figure 4.9. When the UAV Operator begins the first deconfliction procedure he or she is interrupted by another person, a dummy Actor added to the model just for this interruption. Instead of ignoring this interruption the UAV Operator attempts to communicate with this person while still performing the deconfliction procedure. This results in a visual channel conflict as the UAV Operator is watching both the UAS GUI and the person that caused the interruption. In addition to this the UAV Operator is now in a high load/multi tasking state. The result is a workload value of 20, almost double the next highest workload value seen in the simulation.

While we are mostly pleased with the workload results there are a few anomalies in the results that are misleading. The most commonly occurring anomaly is a drastic decrease in workload followed immediately be an equally drastic increase in workload, see figure 4.6. This tends to happen when an Actor is transitioning into the same state it just left. The spike happens because we are collecting metrics twice for each step in the delta clock; once right before the active transitions are fired and once after they are fired. In future work we may collect specific metrics once per delta clock step to prevent these anomalies.

4.2.3 Workload Analysis

To view the metric data more clearly we split the data into four different charts that represent the *resource input workload*, *resource output workload*, *decision workload*, and *overall workload*. In the *resource input workload* chart we get an idea of how many channels, layers, and memory objects the Actor is observing. By comparing this with the *resource output workload* we can see when the inputs occasioned a response from the Actor. For the most part these charts are well behaved and follow the workload patterns that we know about.

On the other hand the *decision workload* typically jumps all over the place. While we believe the increase in *decision workload* does in fact increase *overall workload*, we are unsure how to weight this value in comparison with the *resource workload*. This is something that needs to be verified through sensitivity and user studies in future work.

The last analysis we would like to make is the comparison of our workload metric to that of the adapted Wickens' model from chapter 3. The general flow of the two metrics is very similar, as is expected since everything in the Wickens' model also exists in our workload metric. What is more interesting is where the two metrics differ. Our workload metric is comprised of 9 different values each of which can range from zero to two or more. This adds a fair amount of detail into the workload measurement. An example of this can be seen in the *auto avoid* scenario shown in Figure 4.7. After avoiding the emergency NOTAM the UAV Operator continues to monitor the UAS GUI eventually avoiding another potential conflict. The adapted Wickens' model shows a flat line during this portion of the scenario while our metric shows a heart beat that correlates with the UAV Operator actions.

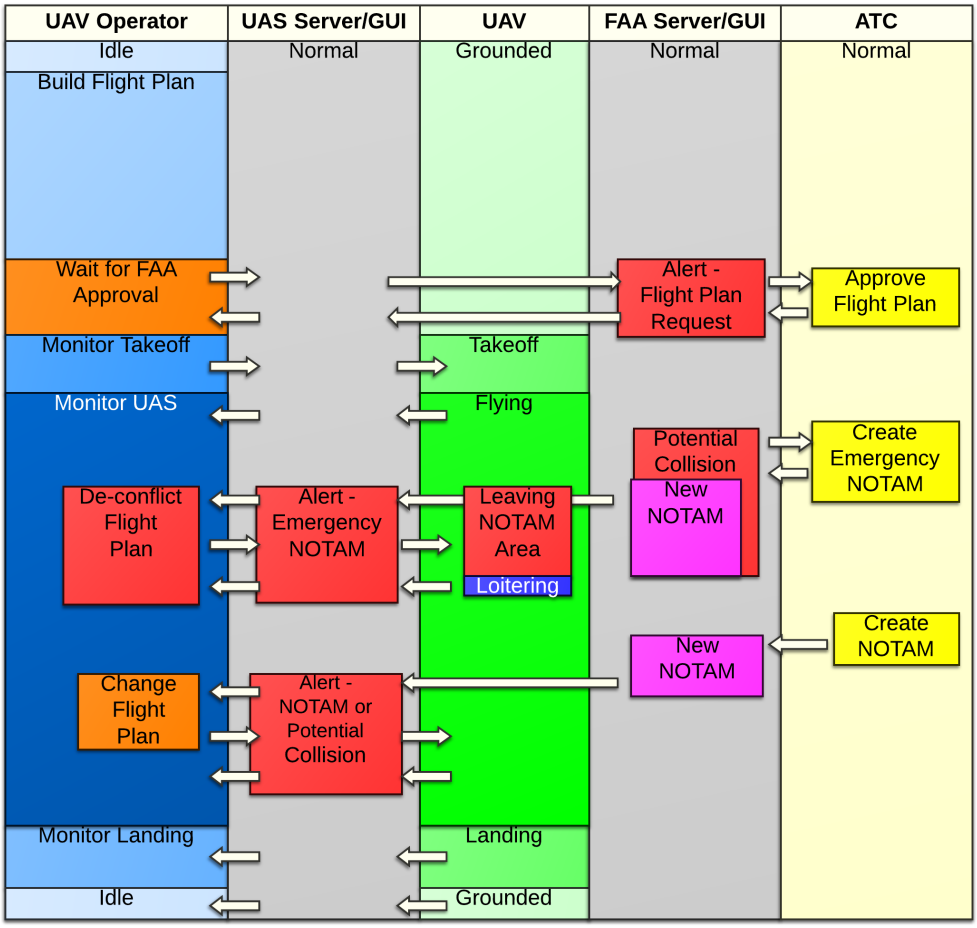

Figure 4.2: DiRG: UAS integration into the NAS Model. Columns represent Actors, column sections represent states, and arrows represent inter Actor data flow.

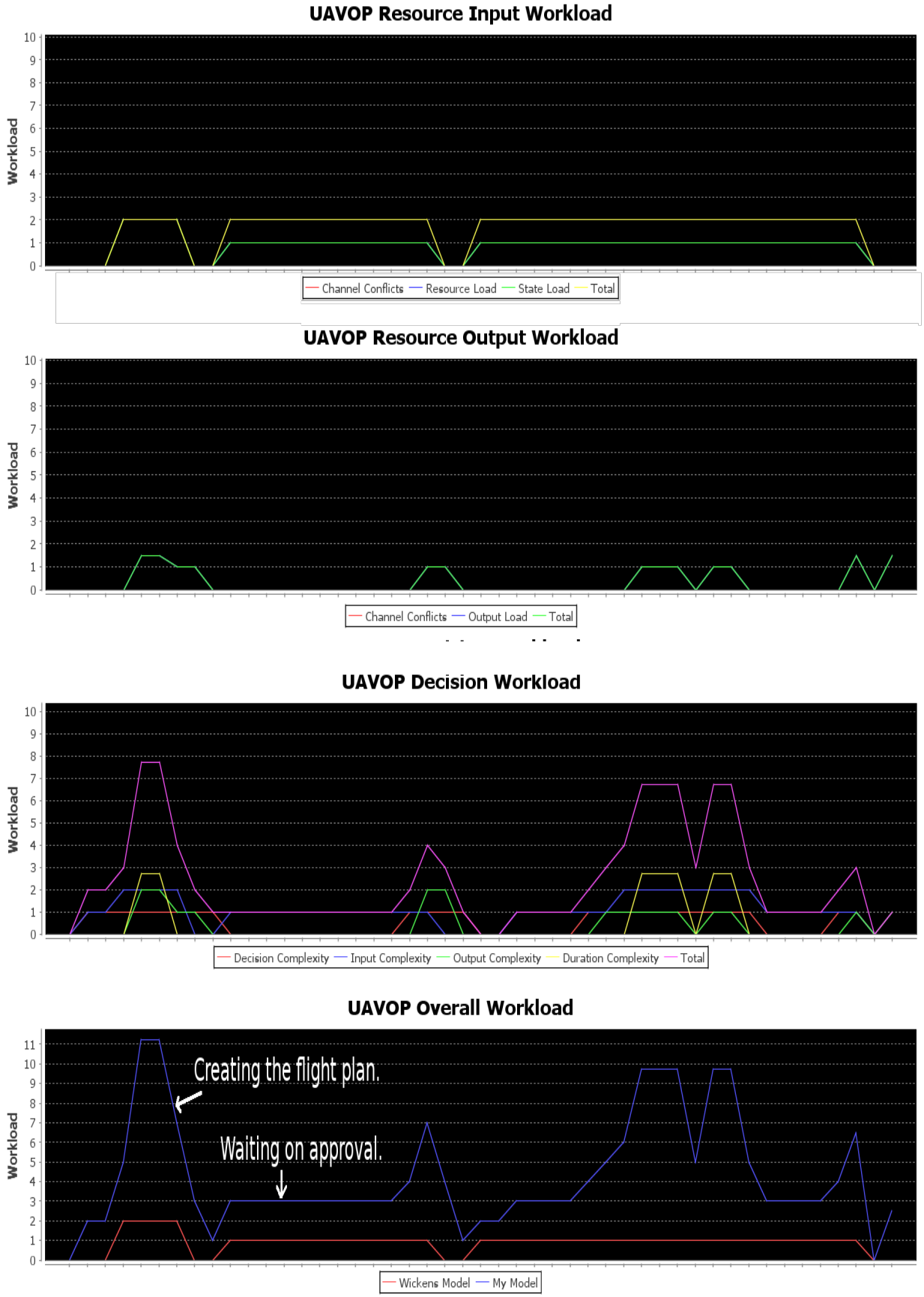

Figure 4.3: UAV Operator: Baseline

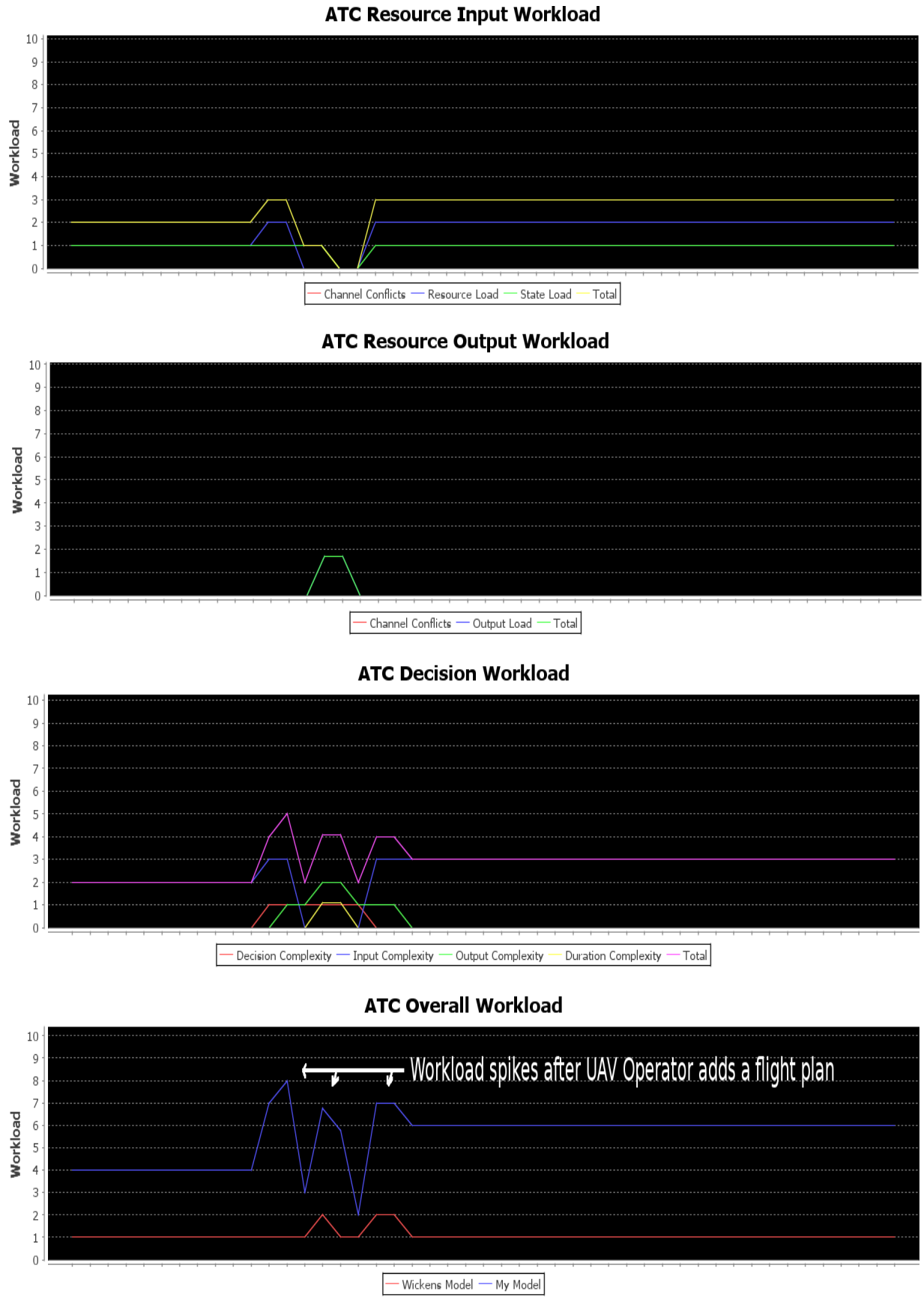

Figure 4.4: ATC: Baseline

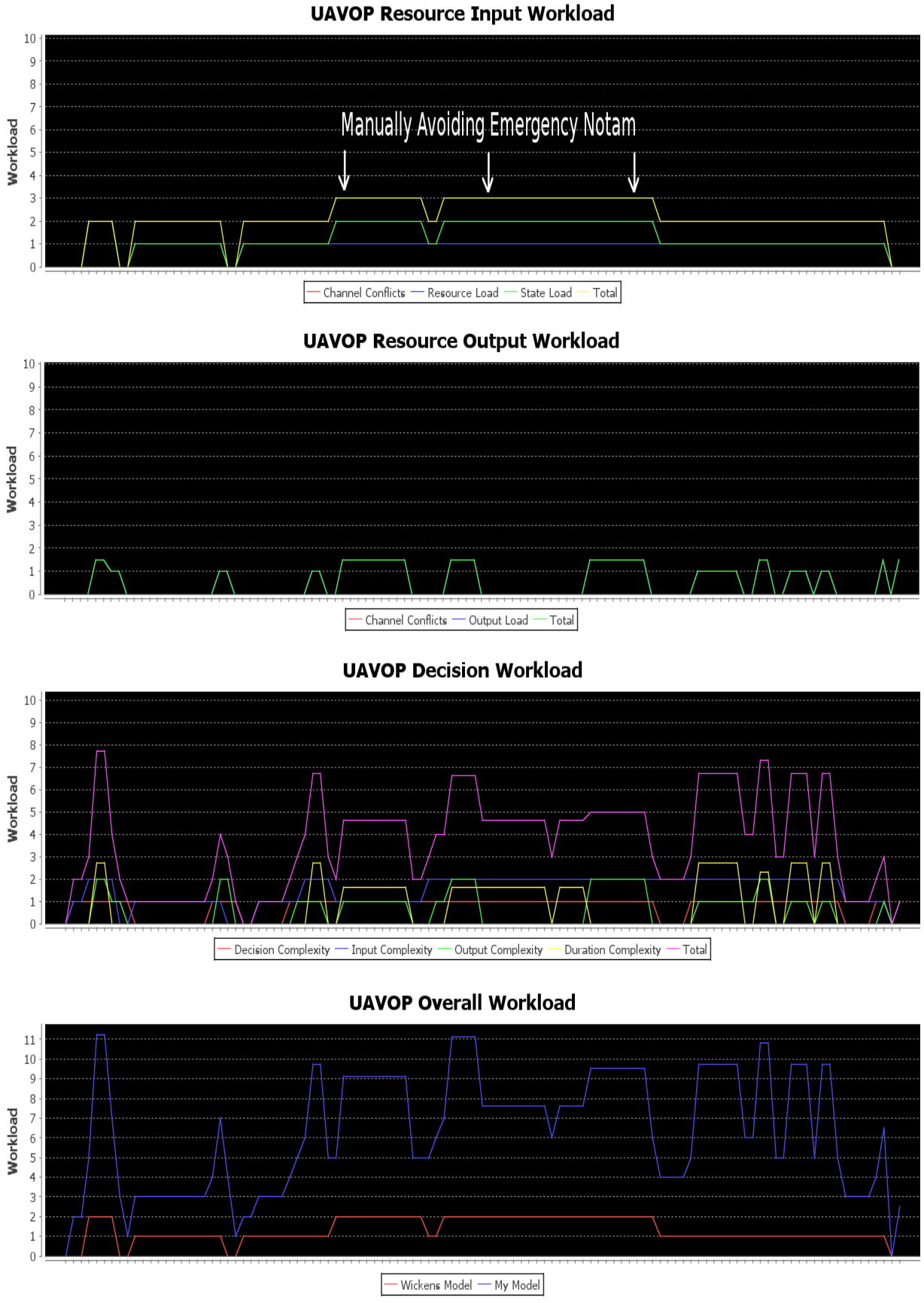

Figure 4.5: UAV Operator: Manually Avoided Emergency NOTAM

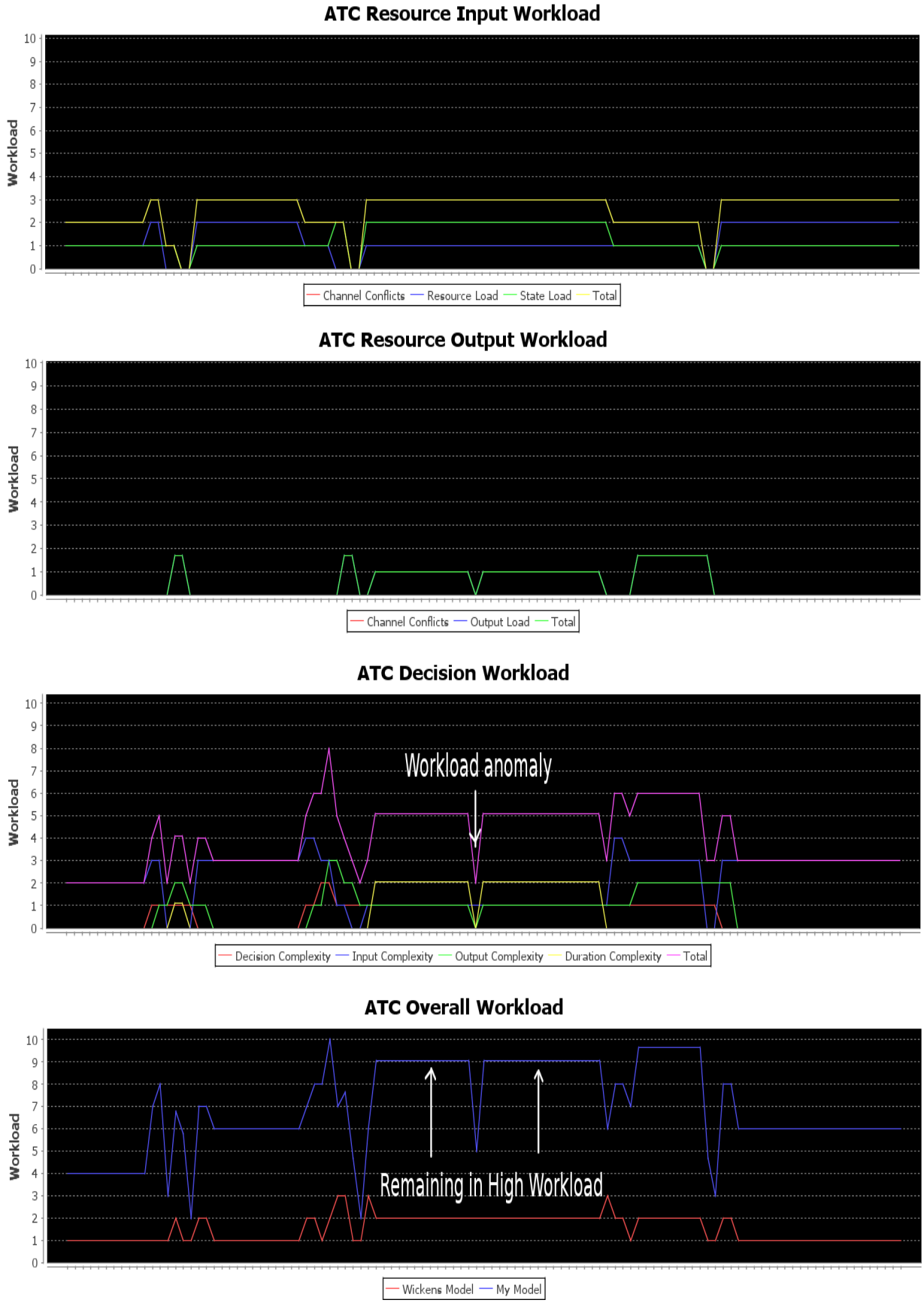

Figure 4.6: ATC: Manually Avoided Emergency NOTAM

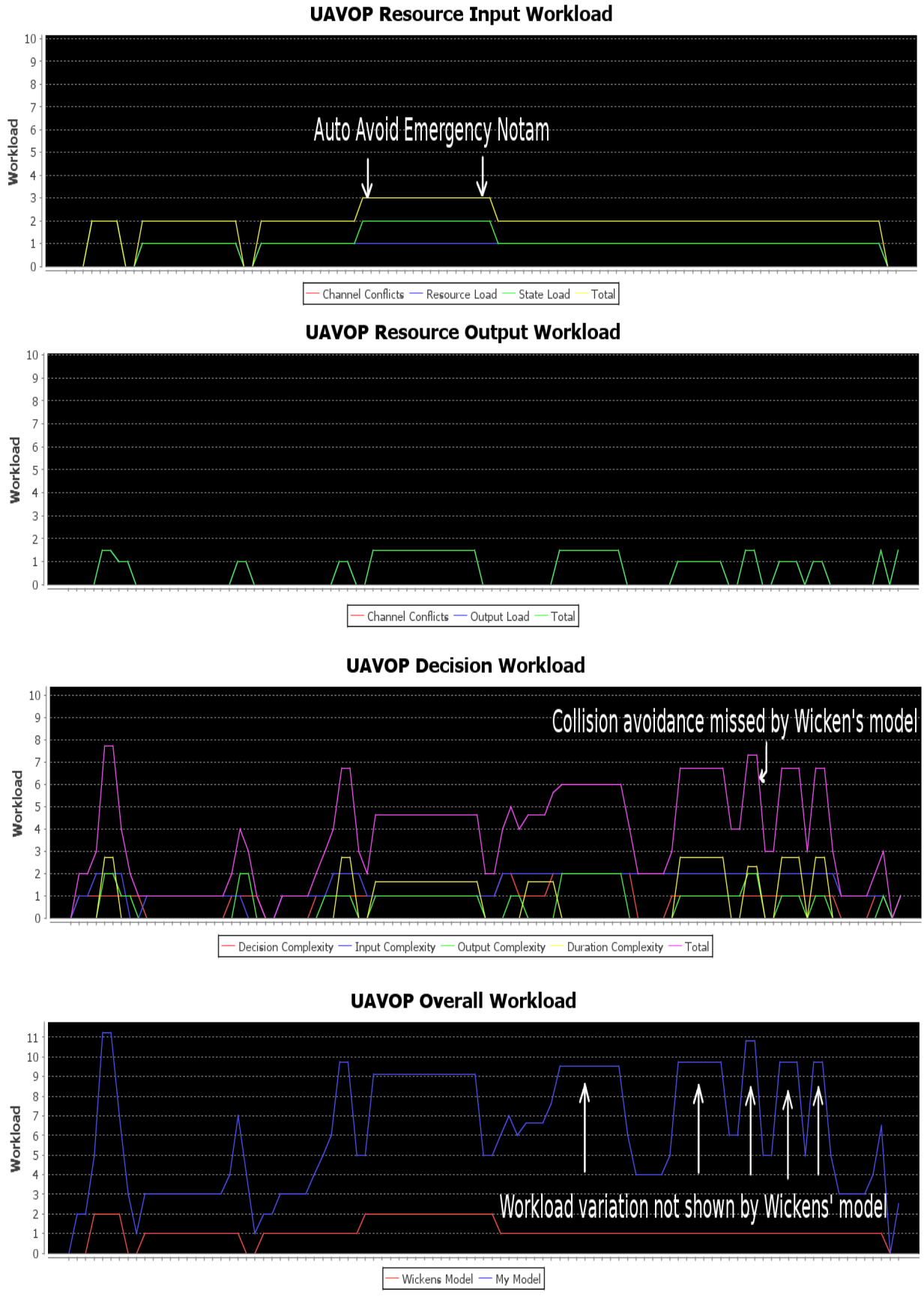

Figure 4.7: UAV Operator: Auto Avoid Emergency NOTAM

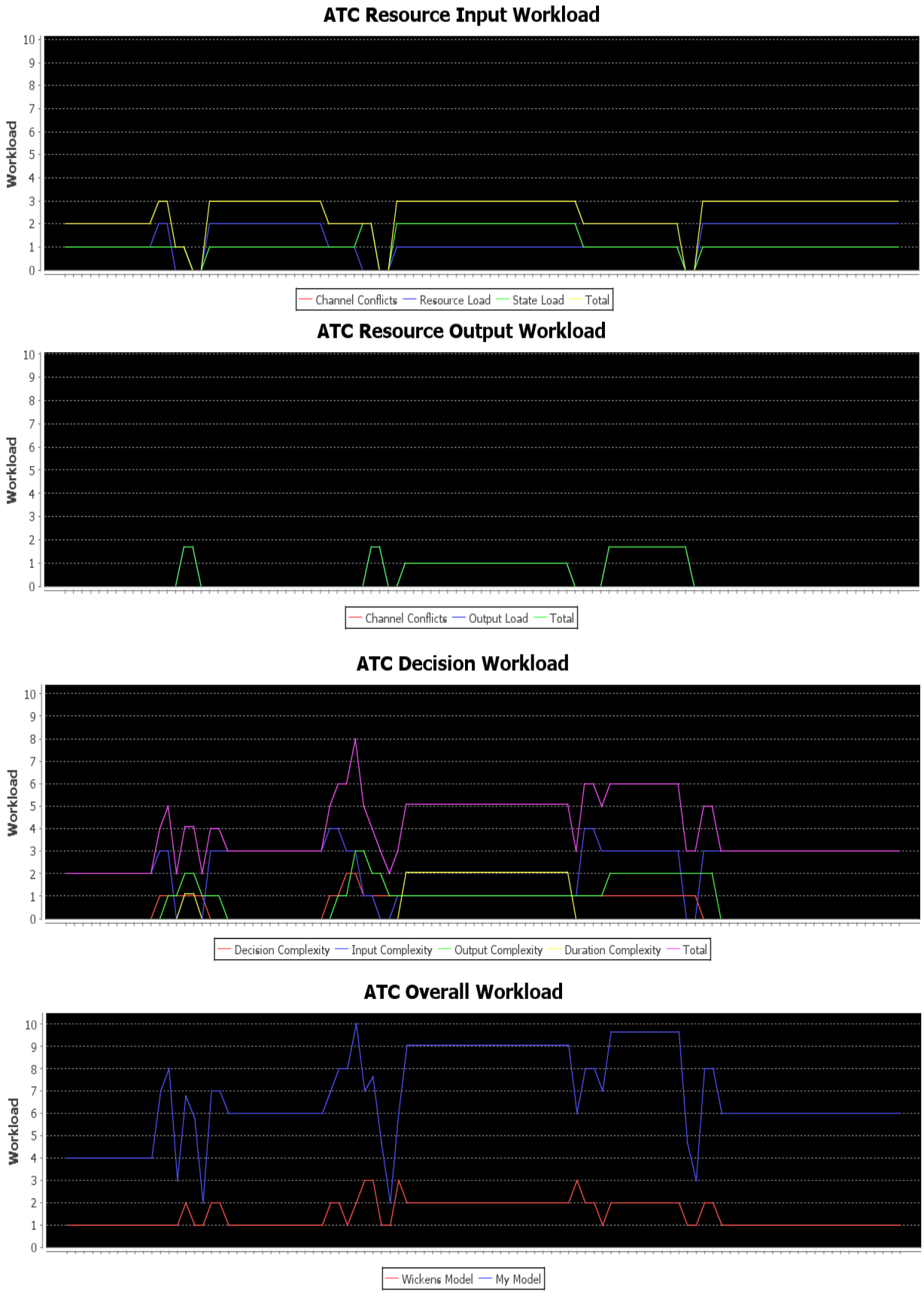

Figure 4.8: ATC: Auto Avoid Emergency NOTAM

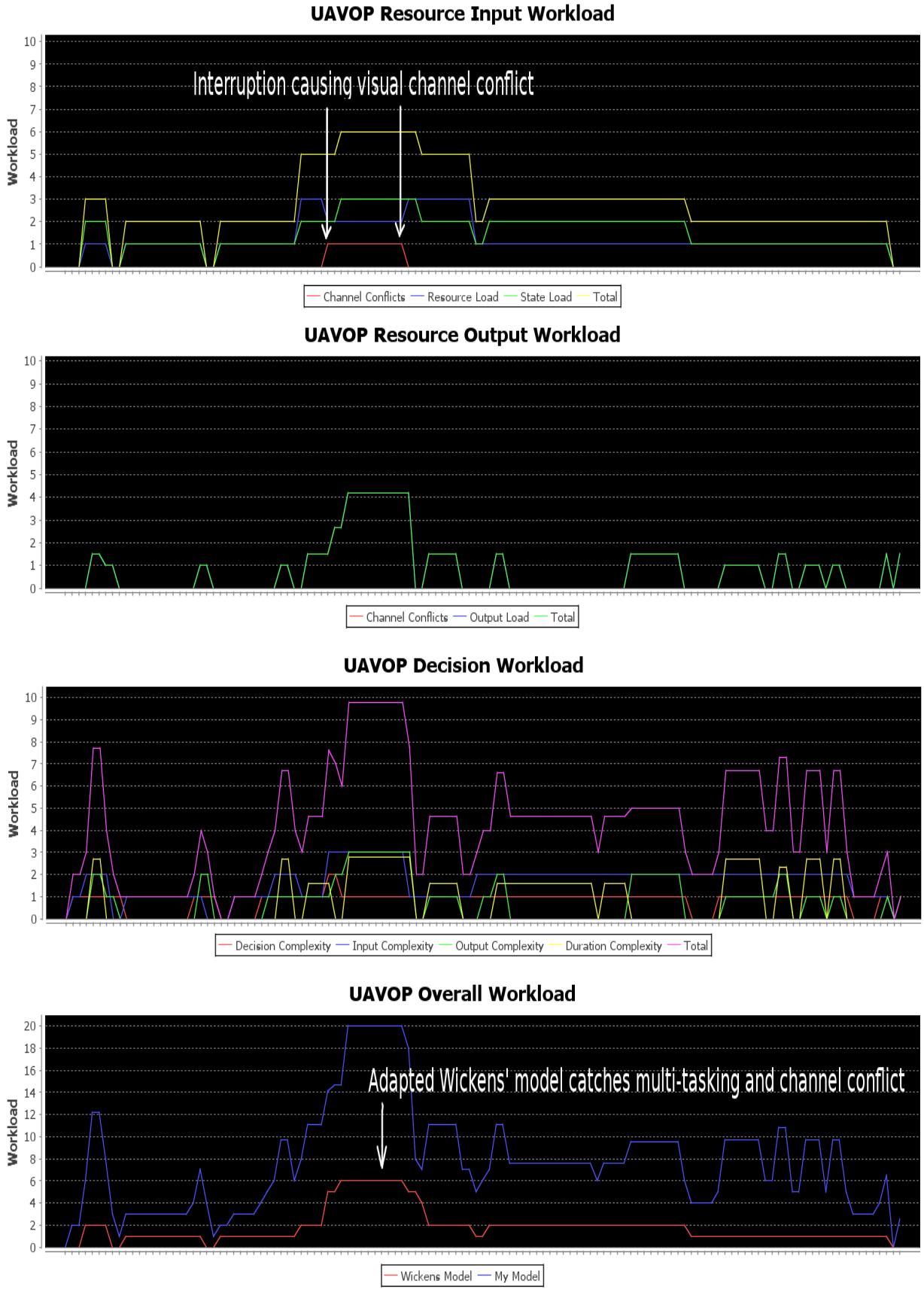

Figure 4.9: UAVOP: Auto Avoid Emergency NOTAM w/ Interruption

Chapter 5

Related Work

Threaded cognition theory states that humans can perform multiple concurrent tasks that do not require executive processes. By making a broad list of resource assumptions about humans, threaded cognition is able to detect the resource conflicts of multiple concurrent tasks. Additionally threaded cognition is able to model task learning. Our model differs from Threaded cognition theory in a few key areas, our model does not allow learning which decreases workload through skill mastery. Also, it does not distinguish between perceptual and motor resources, instead perception and action both use the same resource. In almost all other aspects our model behaves in a similar fashion. [29]

Other similar work has attempted to predict the number of UAVs an operator can control, otherwise known as *fan-out*. [12] This work used queuing theory to model how a human responds in a time sensitive multi-task environment. Queuing theory is helpful in determining the temporal effects of task performance by measuring the difference between when a task was received and when it was executed. At the highest level our model uses Queuing theory, our Actors can only perform a single transition at a time. We differ, however, in that a transition may represent multiple concurrent tasks.

Act-R is a cognitive architecture which attempts to model human cognition and has been successful in human-computer interaction applications. [3] The framework for this architecture consists of modules, buffers, and a pattern matcher which in many ways are very similar to our own framework. The major difference being the cognitive detail available

with ACT-R such as memory access time, task learning, and motor vs perceptual resource differences. [22, 26]

Brahms is a modeling language, developed by the Nasa Ames Research Center, used for modeling multi-agent systems. [8] The language allows the modeler to create highly detailed models consisting of agents, objects, locations, activities, and more. Brahms also allows modeling of time/resource requirements and has been used to find communication bottlenecks and to estimate the value of adding automation. In many ways Brahms is very similar to the Model Abstraction Framework and at some point it may be possible to introduce the workload metrics we are developing into the Brahms language. We chose not to use Brahms for this research due to the complexity of the modeling language, preferring instead a simple modeling language that would make experimenting with the language implementation more feasible.

Chapter 6

Summary and Future Work

As part of this thesis work we have developed the Model Abstraction Framework. This simple modeling framework allowed us to experiment with generating different types of raw data from a human-machine model. The modeling framework uses the concepts of Directed Role Graphs and Directed Team Graphs to build models constrained by specific behavior and communication. The framework then takes this model and converts it into a labeled state transition system that is run on the simulator. During the simulation we collect raw data from the states, transitions, and labels.

Using this raw data we have demonstrated the generation of three workload metrics: Adapted Wickens' Model, Resource Workload Metric, and Decision Workload Metric. The Adapted Wicken's Model, our baseline metric, is an adaptation of a simple model meant to measure cognitive resource load [32]. The Resource Workload Metric, based off of the cognitive workload category presented by Jared et al. [21], represents the cognitive resource load from inter-actor communication and memory access. The Decision Workload Metric, based off of the algorithmic workload category presented by Jared et al. [21], represents the complexity of the decision making process.

Using these metrics we demonstrated two important results. The first result is a general consistency between our baseline model, Adapted Wickens' Model, and the combination of the Resource and Decision workload metrics. While the Resource and Decision workload metrics are more expressive they trend very closely with the Adapted Wickens' Model.

The second result was a general consistency with the known high and low workload periods observed during the UAV Enabled Wilderness Search and Rescue (WiSAR) flight tests. Indeed the results from the Resource and Decision workload metrics was more closely aligned to our observations than that of the Adapted Wickens' Model.

Additionally we are encouraged by the predicted workload reduction observed when we introduced automation into scenario three of the case study. The ability to predict workload reduction with models is one of the primary objectives towards reducing the number of human required to operate a UAS.

6.1 Threats to Validity

While these results are encouraging we realize that they do not yet represent human workload. The Model Abstraction Framework was developed to enable experimentation with the generation of raw data. No studies have been performed showing that the Model Abstraction Framework can accurately model human-machine systems. No work has been done to show that the raw data generated by the framework correlates to memory usage, decision making, cognitive resource usage, and other recognized aspects of human workload.

The baseline workload results were generated by the Model Abstraction Framework and not by a separate modeling framework that has been shown to correctly model human-machine systems and cognitive resource load. Since there is no guarantee that converting the model used by the Model Abstraction Framework to another modeling language produces the same model we chose to instead attempt to convert the workload metric from a task based paradigm to a state/transition based paradigm.

The WiSAR high and low workload profile used for validating workload consistency is not the result of actively measuring different aspects of human workload during a flight test. Instead the workload profile was obtained from in-flight observations and post-flight surveys for a limited number of flight tests.

6.2 Future Work

The ultimate goal of this work is to create a UAS for WiSAR that allows a single human to perform all of the necessary tasks. In order to achieve this goal there is still a lot of work to be done. This thesis represents the beginning phase of this research, which is to begin comprising workload metrics. The next phase will be to modify and validate that the metrics reflect human workload, and the last phase will be to show that the predicted workload reductions are indeed reductions in human workload.

This work focuses on two facets of human workload, cognitive resource load and decision complexity. To predict human workload more facets must be explored. Jared et al. [21] are expanding on these metrics by experimenting with a temporal workload category[21]. They are also developing second and third order metrics within all three workload categories. Additionally it may be necessary to expand the scope of the human workload categories before a moderately accurate measure of human workload can be obtained.

The next step will be to find the correlations between the different workload aspects in order to use the correct ratios to predict workload. Once a metric profile for measuring human workload has been achieved, there may be many, the next phase of modifying and validating the metrics begins.

Appendix A

XML Model Parser

This chapter describes the XML structure used to create the models used to predict workload within this thesis. We first describe the XML structure components, we then briefly describe how the model is sent to the simulator. A link to the full XML models used in our results can be found in appendix B.

A.1 XML Structure

We defined the following XML structure to represent the labeled state transition system sent to the simulator.

The root node is the <*team*> element which consists of the *name* attribute and three child elements: <*channels*>, <*actors*>, and <*events*>.

```
<team name="Basic UAS in NAS">
  <channels></channels>
  <actors></actors>
  <events></events>
</team>
```

A.1.1 Channels

The <*channels*> element represents the DiTG and is comprised of any number of <*channel*> elements. Each <*channel*> element represents a single uni-directional communication channel between a source Actor and a target Actor with attributes for the name, type, source, target,

and optional dataType. This is the only location in the XML that defines channels. Every other <*channel*> element references one of these channels by using the channel name as its value. In future work, we hope to identify other properties of channels such as bandwidth.

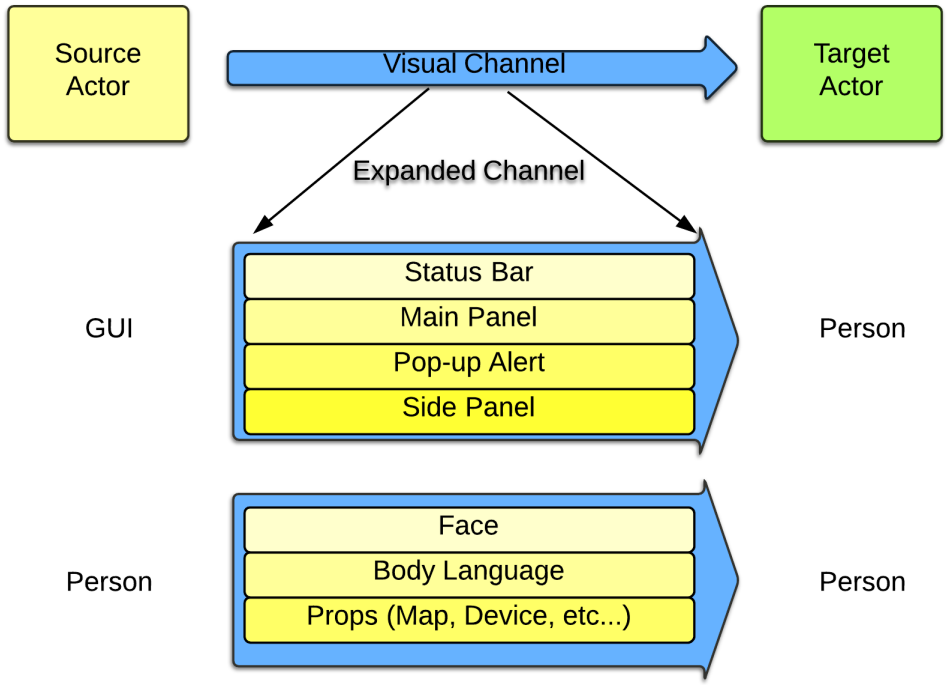

Figure A.1: Communication Channel Layers

Each channel can have any number of layers, see figure A.1. For example, a visual channel from one person to another has two channels, one that analyzes the face and another that analyzes the body. There may be more accurate channel layering for this scenario but these layers satisfy this example. When communication occurs on this visual channel the recipient can examine as many layers as are needed for the given state. If the person is in a distracted state then they may not be looking at the face layer, which may reduce the probability of comprehending the corresponding audio channel. Another example where this

is useful is in GUIs. Each item that a GUI represents to the user can be represented as a different layer. The more layers a GUI presents to a user the more complex it becomes, potentially making it more difficult for a person to analyze due to conflicting resources [29].

```
<channels>
  <channel name="V_UAS_UAVOP" type="VISUAL" source="UAS" target="UAVOP"/>
  <channel name="M_UAVOP_UAS" type="MANUAL" source="UAVOP" target="UAS"/>
</channels>


<channel>M_UAS_UAVOP</channel>
<channel name="MANUAL_UAVOP_UAS">
  <layer name="MOUSE"><null/></layer>
  <layer name="KEYBOARD"><null/></layer>
</channel>
```

A.1.2 Actors

The <*actors*> element represents the DiRGs contained in the model and is comprised of one or more <*actor*> elements. Each <*actor*> has a *name* attribute, which must be unique, and a *showMetrics* attribute and is made up of several child elements. The <*inputchannels*> and <*outputchannels*> elements are placed here to validate the DiTG. The *actor* may have zero or more *memory* elements that represent declarative memory, memory that is an active part of the cognitive decision process, and is used internally as part of the labeled state transitions. The <*actor*> contains a list of one or more <*state*> elements inside of the <*states*> element. Additionally the <*actor*> element must contain a single <*startState*> element with a value that matches the name attribute of one of the <*states*> child elements.

```
<actors>
  <actor name="UAVOP" showMetrics="true">
    <inputchannels>
```

```
      <channel>V_UAS_UAVOP</channel>
      <channel>A_UAS_UAVOP</channel>
      <channel>V_UAV_UAVOP</channel>
    </inputchannels>
    <outputchannels>
      <channel>M_UAVOP_UAS</channel>
    </outputchannels>
    <memory name="FLIGHT_PLAN_READY" dataType="Boolean">false</memory>
    <startState>IDLE</startState>
    <states>
      <state name="IDLE" load="0"></state>
    </states>
  </actor>
</actors>
```

A.1.3 States and Transitions

The <*state*> element represents the 'State' in the labeled state transition system. Each <*state*> element has a *name* attribute, which must be unique to the parent <*actor*> element, and a *load* attribute with a value of 0-4 (we will discuss state load in the next chapter). The <*state*> element also contains zero or more <*transition*> child elements. If no <*transition*> child elements are specified then it is an end state.

The <*transition*> element represents the 'Transition' in the labeled state transition system and is composed of *durationMin* and *durationMax* attributes that are used to create a transition duration range to enable timing analysis in JPF. It also has a *priority* attribute used to rank a transitions priority with respect to the other state transitions. Each <*transition*> element contains four child elements: <*description*>, <*inputs*>, <*outputs*>, and <*endState*>.

The <*description*> element value contains a description of the transition that is meant to help give the modeler insight into what the transition is actually doing. It is also used in the debug logs to help track down modeling errors.

The <*inputs*> and <*outputs*> elements represent the 'Label' in the labeled state transition system. These elements contain zero or more <*memory*> and <*channel*> elements. All <*inputs*> element children have a *name* attribute, a *predicate* attribute, an optional *dataType* attribute and a value. If the child element is a <*channel*> then it may optionally declare only the *name* attribute and one or more <*layer*> child elements to specifically describe the different types of data being sent over the channel. The <*layer*> element consists of *name*, *predicate*, and optional *dataType* attributes with an accompanying data value. The <*channel*> name must correlate with an <*actor*> <*inputchannels*> <*channel*> value. The *predicate* attribute contains one of six predicate values: equal to, not equal, greater than, less than, greater than or equal, or less than or equal. The *dataType* is one of three options: String, Integer, or Boolean with a default of String. The value of these child elements is the value that will be compared against the channel or memory value.

The <*outputs*> element is similar to the <*inputs*> element except that its children do not specify *predicate* or *dataType* attributes.

This structure allows the modeler to use a 2 dimensional labeling system. The first dimension is the direction of the data flow, input or output. The second dimension is the source of the data; channel, layer, or memory. This allows the labeled state transition system to be very flexible while using a relatively simple syntax. We should also point out that the transition <*inputs*> element uses the following propositional logic: $(Child_1 \bigwedge Child_2 \bigwedge \ldots \bigwedge Child_n)$. This means that all transition inputs are evaluated together, preventing the modeler from defining multiple sets of inputs for a single transition. Thus each labeled state transition, no matter how similar to another, must be expressed as a separate <*transition*> element.

The value of the <*endState*> element contains the name of the Actor's ending state if this transition is applied. This value cannot be empty and must match an existing <*state*> name.

```
<state name="END_BUILD_FP" load="0">
  <transition durationMin="1" durationMax="1" priority="1">
    <description>Clear mouse and keyboard output layers</description>
    <inputs>
    </inputs>
    <outputs>
      <channel name="M_UAVOP_UAS">
        <layer name="MOUSE"><null/></layer>
        <layer name="KEYBOARD"><null/></layer>
      </channel>
    </outputs>
    <endState>WAITING_ON_FAA</endState>
  </transition>
</state>
```

A.1.4 Events

The <*events*> element contains a child <*event*> element for each unique type of event the system can experience. The <*event*> has a *name* attribute, must be unique, and a *count* attribute that determines how many times the event will be fired during the simulation. Each <*event*> element defines a single <*transition*> element that differs from the previously mentioned transition element in that it does not define duration, priority, or an end state. Since Events have a single state and a single transition there is no need for these additional transition attributes. At the implementation level this forces Events to act as sipmle triggers that perturb the model in interesting ways.

```
<events>
  <event name="NEW_MISSION" count="1">
    <transition>
      <description> Start a new UAV mission</description>
      <inputs>
      </inputs>
      <outputs>
        <channel name="NEW_MISSION">NEW_MISSION</channel>
      </outputs>
    </transition>
  </event>
</events>
```

A.2 XML Model Parser

Figure A.2 illustrates the process of parsing the XML model. To accomplish this we created a Java class named XMLModelParser. This class performs two main functions. The first is to create the necessary Java objects and the second is to perform basic error checking on the model.

The XMLModelParser works by first loading the XML model into an XML object. This allows us to search the XML structure for specific elements and attributes. The structure of the XML model facilitates the parsing effort since each child element is also a child object. The first object that is created is the Team. Next the DiTG is added to the team by creating ComChannel objects for each channel. The Actor parsing is the most complex. Each actor element defines input and output channels, memory, states, transitions, and labels.

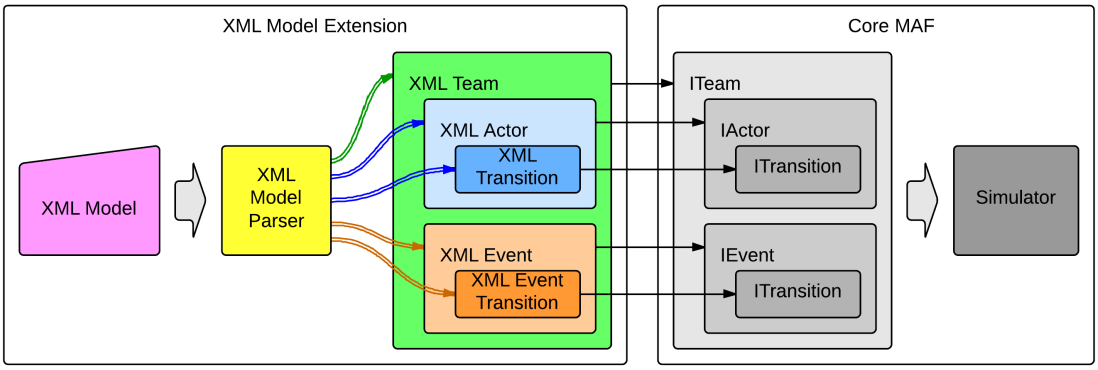

Figure A.2: XML Model Parser Breakdown

A.2.1 Attaching to the Simulator

To pass a valid labeled state transition system to the simulator, the XMLModelParser must construct Java classes that implement the Modeling Interface. To accomplish this we created a set of generic XML classes. Each class implements a specific portion of the Modeling Interface such as the ITransition or IActor interfaces. These interfaces allow the simulator to query and execute transitions on the labeled state transition system. The list of XML classes includes the XMLTeam, XMLActor, XMLTransition, XMLEvent, XMLEventTransition and other helper classes. A link to the source code can be found in appendix B. Each of these generic XML classes lies separate from the core simulation code.

This means that the addition of the XML parser does not prevent the use of custom Actors or Transition classes, use of a different model parser, or extension of our existing XML parser. In fact our XML model parser is meant to be expanded to handle a more robust set of models. As we developed a model using this XMLModelParser we had several occasions to extend its capabilities.

Appendix B

Code and Models

The source code for the modeling framework, xml parsing, WiSAR model, and UAS integration into the NAS models can be obtained on github from the following link:

`https://github.com/tjflexmaster/UAV_ROLE_MODEL`

Appendix C

Debug Log Generated by the Model

This is the output log generated when running the UAS integrated into the NAS model with automatic emergency NOTAM avoidance. The main simulation cycle consists of three main parts: Load transitions, advance time, and fire transitions.

```
Load Transition(1):  Start a new UAV mission
Time: 1
Fired Transition:  Start a new UAV mission
-------------------------------------------------
Load Transition(1): UAVOP: StartState: IDLE EndState: BUILD_FP Description: Received new mission, begin building flight plan.
Time: 2
Fired Transition: UAVOP: StartState: IDLE EndState: BUILD_FP Description: Received new mission, begin building flight plan.
-------------------------------------------------
Load Transition(900): UAVOP: StartState: BUILD_FP EndState: BUILD_FP Description: Build a flight plan
Time: 902
Fired Transition: UAVOP: StartState: BUILD_FP EndState: BUILD_FP Description: Build a flight plan
-------------------------------------------------
Load Transition(1): UAVOP: StartState: BUILD_FP EndState: END_BUILD_FP Description: Click the Send flight plan button
Time: 903
Fired Transition: UAVOP: StartState: BUILD_FP EndState: END_BUILD_FP Description: Click the Send flight plan button
-------------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: Send flight plan to FAA
Load Transition(1): UAVOP: StartState: END_BUILD_FP EndState: WAITING_ON_FAA Description: Clear mouse and keyboard output layers
Time: 904
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: Send flight plan to FAA
Fired Transition: UAVOP: StartState: END_BUILD_FP EndState: WAITING_ON_FAA Description: Clear mouse and keyboard output layers
-------------------------------------------------
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: Received Flight plan from UAS
Time: 905
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: Received Flight plan from UAS
-------------------------------------------------
Load Transition(1): ATC: StartState: NORMAL EndState: APPROVING_FLIGHT_PLAN Description: ATC is ready to approve flight plans.
Time: 906
Fired Transition: ATC: StartState: NORMAL EndState: APPROVING_FLIGHT_PLAN Description: ATC is ready to approve flight plans.
-------------------------------------------------
Load Transition(200): ATC: StartState: APPROVING_FLIGHT_PLAN EndState: END_APPROVING_FLIGHT_PLAN Description: ATC is approving flight plans.
Time: 1106
Fired Transition: ATC: StartState: APPROVING_FLIGHT_PLAN EndState: END_APPROVING_FLIGHT_PLAN Description: ATC is approving flight plans.
-------------------------------------------------
Load Transition(1): ATC: StartState: END_APPROVING_FLIGHT_PLAN EndState: NORMAL Description: ATC finished approving the flight plan.
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: FAA received approved flight plan from ATC, send to UAS
Time: 1107
Fired Transition: ATC: StartState: END_APPROVING_FLIGHT_PLAN EndState: NORMAL Description: ATC finished approving the flight plan.
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: FAA received approved flight plan from ATC, send to UAS
-------------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: Received flight plan approved from FAA
Time: 1108
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: Received flight plan approved from FAA
-------------------------------------------------
Load Transition(30): UAVOP: StartState: WAITING_ON_FAA EndState: END_WAITING_ON_FAA Description: FAA approved the flight
Time: 1138
Fired Transition: UAVOP: StartState: WAITING_ON_FAA EndState: END_WAITING_ON_FAA Description: FAA approved the flight
-------------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAVOP sent take off command
Load Transition(1): UAVOP: StartState: END_WAITING_ON_FAA EndState: MONITOR_TAKEOFF Description: Clear user output
Time: 1139
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAVOP sent take off command
Fired Transition: UAVOP: StartState: END_WAITING_ON_FAA EndState: MONITOR_TAKEOFF Description: Clear user output
-------------------------------------------------
Load Transition(1): UAV: StartState: GROUNDED EndState: TAKEOFF Description: Move to takeoff from UAVGUI cmd
Time: 1140
Fired Transition: UAV: StartState: GROUNDED EndState: TAKEOFF Description: Move to takeoff from UAVGUI cmd
-------------------------------------------------
Load Transition(300): UAV: StartState: TAKEOFF EndState: FLYING Description: Automatic Transition to Flying
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to takeoff, show this to the UAVOp
```

```
Time: 1141
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to takeoff, show this to the UAVOp
-----------------------------------------------
Time: 1440
Fired Transition: UAV: StartState: TAKEOFF EndState: FLYING Description: Automatic Transition to Flying
-----------------------------------------------
Load Transition(1): Potential collision on UAS radar
Load Transition(1): FAA radar shows a potential collision with a UAV
Load Transition(1): Request that ATC create a new NOTAM on UAV FP
Load Transition(10000): UAV: StartState: FLYING EndState: LANDING Description: UAV normal flight
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to fly, show this to the UAVOp
Load Transition(1): UAVOP: StartState: MONITOR_TAKEOFF EndState: MONITOR_UAVGUI Description: UAV is airborne, move to monitor GUI
Time: 1441
Fired Transition: Request that ATC create a new NOTAM on UAV FP
Fired Transition: Potential collision on UAS radar
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to fly, show this to the UAVOp
Fired Transition: FAA radar shows a potential collision with a UAV
Fired Transition: UAVOP: StartState: MONITOR_TAKEOFF EndState: MONITOR_UAVGUI Description: UAV is airborne, move to monitor GUI
-----------------------------------------------
Load Transition(1): ATC: StartState: NORMAL EndState: CREATE_NOTAM Description: ATC needs to create a new NOTAM
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: Potential collision on FAA Radar, show the user
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAS received radar collision event
Load Transition(900): UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_UAVGUI Description: Monitoring the UAVGUI
Time: 1442
Fired Transition: ATC: StartState: NORMAL EndState: CREATE_NOTAM Description: ATC needs to create a new NOTAM
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: Potential collision on FAA Radar, show the user
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAS received radar collision event
-----------------------------------------------
Load Transition(1): ATC: StartState: CREATE_NOTAM EndState: CREATE_EMERGENCY_NOTAM Description: ATC sees the alert about a UAV collision and tries to avoid it
Load Transition(1): UAVOP: StartState: MONITOR_UAVGUI EndState: AVOID_COLLISION Description: Operator sees a potential conflict and trys to avoid it
Time: 1443
Fired Transition: ATC: StartState: CREATE_NOTAM EndState: CREATE_EMERGENCY_NOTAM Description: ATC sees the alert about a UAV collision and tries to avoid it
Fired Transition: UAVOP: StartState: MONITOR_UAVGUI EndState: AVOID_COLLISION Description: Operator sees a potential conflict and trys to avoid it
-----------------------------------------------
Load Transition(60): ATC: StartState: CREATE_EMERGENCY_NOTAM EndState: END_CREATE_EMERGENCY_NOTAM Description: ATC is creating an Emergency Notam around the collision
Load Transition(300): UAVOP: StartState: AVOID_COLLISION EndState: AVOID_COLLISION Description: Operator is in the act of deconflicing the UAV
Time: 1503
Fired Transition: ATC: StartState: CREATE_EMERGENCY_NOTAM EndState: END_CREATE_EMERGENCY_NOTAM Description: ATC is creating an Emergency Notam around the collision
-----------------------------------------------
Load Transition(1): ATC: StartState: END_CREATE_EMERGENCY_NOTAM EndState: AVOID_UAV_COLLISION Description: ATC is creating an Emergency Notam around the collision
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: ATC sent an Emergency NOTAM to the FAA system, send it to the UAS
Time: 1504
Fired Transition: ATC: StartState: END_CREATE_EMERGENCY_NOTAM EndState: AVOID_UAV_COLLISION Description: ATC is creating an Emergency Notam around the collision
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: ATC sent an Emergency NOTAM to the FAA system, send it to the UAS
-----------------------------------------------
Load Transition(500): ATC: StartState: AVOID_UAV_COLLISION EndState: AVOID_UAV_COLLISION Description: ATC is waiting for collision to stop.
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: Receive Emergency Notam on the UAV, force UAV away from NOTAM
Time: 1505
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: Receive Emergency Notam on the UAV, force UAV away from NOTAM
-----------------------------------------------
Load Transition(1): UAV: StartState: FLYING EndState: AVOID_NOTAM Description: UAV gets a command to avoid a NOTAM
Time: 1506
Fired Transition: UAV: StartState: FLYING EndState: AVOID_NOTAM Description: UAV gets a command to avoid a NOTAM
-----------------------------------------------
Load Transition(300): UAV: StartState: AVOID_NOTAM EndState: LOITER Description: Avoid the Emergency NOTAM
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV is avoiding a NOTAM, show this to the UAVOp
Time: 1507
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV is avoiding a NOTAM, show this to the UAVOp
-----------------------------------------------
Time: 1743
Fired Transition: UAVOP: StartState: AVOID_COLLISION EndState: AVOID_COLLISION Description: Operator is in the act of deconflicing the UAV
-----------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: Radar collision has been avoided
Time: 1744
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: Radar collision has been avoided
-----------------------------------------------
Load Transition(1): UAVOP: StartState: AVOID_COLLISION EndState: MONITOR_UAVGUI Description: Collision has been avoided, return to watching the UAVGUI
Time: 1745
Fired Transition: UAVOP: StartState: AVOID_COLLISION EndState: MONITOR_UAVGUI Description: Collision has been avoided, return to watching the UAVGUI
-----------------------------------------------
Load Transition(1): UAVOP: StartState: MONITOR_UAVGUI EndState: AVOID_EMERGENCY_NOTAM Description: Operator notices an emergency notam on the UAV and attempts to assist
Time: 1746
Fired Transition: UAVOP: StartState: MONITOR_UAVGUI EndState: AVOID_EMERGENCY_NOTAM Description: Operator notices an emergency notam on the UAV and attempts to assist
-----------------------------------------------
Load Transition(300): UAVOP: StartState: AVOID_EMERGENCY_NOTAM EndState: AVOID_EMERGENCY_NOTAM Description: Operator is waiting for the UAV to auto remove itself from the
Time: 1806
Fired Transition: UAV: StartState: AVOID_NOTAM EndState: LOITER Description: Avoid the Emergency NOTAM
-----------------------------------------------
Load Transition(1): End the FAA potential collision with a UAV
Load Transition(10000): UAV: StartState: LOITER EndState: LANDING Description: Default to land after loitering for a long time.
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV is now Loitering, show this to the UAVOp
Time: 1807
Fired Transition: End the FAA potential collision with a UAV
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV is now Loitering, show this to the UAVOp
-----------------------------------------------
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: Radar collision has ended.
Load Transition(5): UAVOP: StartState: AVOID_EMERGENCY_NOTAM EndState: AVOID_EMERGENCY_NOTAM Description: Operator sees that the UAV is loitering, have it resume flight n
Time: 1808
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: Radar collision has ended.
-----------------------------------------------
Load Transition(1): ATC: StartState: AVOID_UAV_COLLISION EndState: NORMAL Description: FAA radar no longer shows a potential collision
Time: 1809
```

```
Fired Transition: ATC: StartState: AVOID_UAV_COLLISION EndState: NORMAL Description: FAA radar no longer shows a potential collision
-------------------------------------------------
Load Transition(1): ATC: StartState: NORMAL EndState: CREATE_NOTAM Description: ATC needs to create a new NOTAM
Time: 1810
Fired Transition: ATC: StartState: NORMAL EndState: CREATE_NOTAM Description: ATC needs to create a new NOTAM
-------------------------------------------------
Load Transition(60): ATC: StartState: CREATE_NOTAM EndState: END_CREATE_NOTAM Description: ATC is creating a new NOTAM
Time: 1812
Fired Transition: UAVOP: StartState: AVOID_EMERGENCY_NOTAM EndState: AVOID_EMERGENCY_NOTAM Description: Operator sees that the UAV is loitering, have it resume flight now
-------------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAVOP sent resume flight
Load Transition(1): UAVOP: StartState: AVOID_EMERGENCY_NOTAM EndState: MONITOR_UAVGUI Description: NOTAM has been avoided, return to watching the UAVGUI
Time: 1813
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAVOP sent resume flight
Fired Transition: UAVOP: StartState: AVOID_EMERGENCY_NOTAM EndState: MONITOR_UAVGUI Description: NOTAM has been avoided, return to watching the UAVGUI
-------------------------------------------------
Load Transition(1): UAV: StartState: LOITER EndState: FLYING Description: Resume a normal flight
Time: 1814
Fired Transition: UAV: StartState: LOITER EndState: FLYING Description: Resume a normal flight
-------------------------------------------------
Load Transition(10000): UAV: StartState: FLYING EndState: LANDING Description: UAV normal flight
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to fly, show this to the UAVOp
Time: 1815
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to fly, show this to the UAVOp
-------------------------------------------------
Load Transition(900): UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_UAVGUI Description: Monitoring the UAVGUI
Time: 1870
Fired Transition: ATC: StartState: CREATE_NOTAM EndState: END_CREATE_NOTAM Description: ATC is creating a new NOTAM
-------------------------------------------------
Load Transition(1): ATC: StartState: END_CREATE_NOTAM EndState: NORMAL Description: ATC finished creating NOTAM return to normal
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: FAA received new NOTAM on UAV FP from ATC, send to UAS
Time: 1871
Fired Transition: ATC: StartState: END_CREATE_NOTAM EndState: NORMAL Description: ATC finished creating NOTAM return to normal
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: FAA received new NOTAM on UAV FP from ATC, send to UAS
-------------------------------------------------
Load Transition(1): FAA: StartState: NORMAL EndState: NORMAL Description: Clear New NOTAM memory
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: Receive new NOTAM from FAA which appears on the Flight Plan
Time: 1872
Fired Transition: FAA: StartState: NORMAL EndState: NORMAL Description: Clear New NOTAM memory
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: Receive new NOTAM from FAA which appears on the Flight Plan
-------------------------------------------------
Load Transition(1): UAVOP: StartState: MONITOR_UAVGUI EndState: AVOID_NOTAM Description: Operator notices new NOTAM(s) on the flight plan, begin changing the flight plan.
Time: 1873
Fired Transition: UAVOP: StartState: MONITOR_UAVGUI EndState: AVOID_NOTAM Description: Operator notices new NOTAM(s) on the flight plan, begin changing the flight plan.
-------------------------------------------------
Load Transition(600): UAVOP: StartState: AVOID_NOTAM EndState: AVOID_NOTAM Description: Operator is changing the flight plan for a normal NOTAM
Time: 2473
Fired Transition: UAVOP: StartState: AVOID_NOTAM EndState: AVOID_NOTAM Description: Operator is changing the flight plan for a normal NOTAM
-------------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAVOP sent a new flight
Load Transition(1): UAVOP: StartState: AVOID_NOTAM EndState: MONITOR_UAVGUI Description: NOTAM has been avoided, return to watching the UAVGUI
Time: 2474
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAVOP sent a new flight
Fired Transition: UAVOP: StartState: AVOID_NOTAM EndState: MONITOR_UAVGUI Description: NOTAM has been avoided, return to watching the UAVGUI
-------------------------------------------------
Load Transition(900): UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_UAVGUI Description: Monitoring the UAVGUI
Time: 3374
Fired Transition: UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_UAVGUI Description: Monitoring the UAVGUI
-------------------------------------------------
Time: 11814
Fired Transition: UAV: StartState: FLYING EndState: LANDING Description: UAV normal flight
-------------------------------------------------
Load Transition(600): UAV: StartState: LANDING EndState: GROUNDED Description: Land the UAV
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to land, show this to the UAVOp
Load Transition(900): UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_UAVGUI Description: Monitoring the UAVGUI
Time: 11815
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV has started to land, show this to the UAVOp
-------------------------------------------------
Load Transition(1): UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_LANDING Description: Operator sees that the UAV is beginning the landing approach
Time: 11816
Fired Transition: UAVOP: StartState: MONITOR_UAVGUI EndState: MONITOR_LANDING Description: Operator sees that the UAV is beginning the landing approach
-------------------------------------------------
Time: 12414
Fired Transition: UAV: StartState: LANDING EndState: GROUNDED Description: Land the UAV
-------------------------------------------------
Load Transition(1): UAS: StartState: NORMAL EndState: NORMAL Description: UAV is GROUNDED, show this to the UAVOp
Time: 12415
Fired Transition: UAS: StartState: NORMAL EndState: NORMAL Description: UAV is GROUNDED, show this to the UAVOp
-------------------------------------------------
Load Transition(30): UAVOP: StartState: MONITOR_LANDING EndState: IDLE Description: Operator sees that the UAV has landed and sets the mission to complete.
Time: 12445
Fired Transition: UAVOP: StartState: MONITOR_LANDING EndState: IDLE Description: Operator sees that the UAV has landed and sets the mission to complete.
-------------------------------------------------
Time: 12445
-------------------------------------------------
```

References

[1] *A Synergistic and Extensible Framework for Multi-Agent System Verification*, AAMAS, Saint Paul, Minnesota, 2013.

[2] J. A. Adams, C. M. Humphrey, M. A. Goodrich, J. L. Cooper, B. S. Morse, C. Engh, and N. Rasmussen. Cognitive task analysis for developing UAV wilderness search support. *Journal of Cognitive Engineering and Decision Making*, 3(1):1–26, 2009.

[3] John R Anderson, Daniel Bothell, Michael D Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036, 2004.

[4] E. Bolton, M. Bass and Siminiceanu R. Using formal verification to evaluate human-automation interaction: A review. In *Systems, Man and Cybernetics (SMC), IEEE International Conference on*, Manchester, England, 2013. IEEE.

[5] Matthew L Bolton and Ellen J Bass. Enhanced operator function model: A generic human task behavior modeling language. In *Systems, Man and Cybernetics (SMC), IEEE International Conference on*, pages 2904–2911, San Anotonio, Texas, 2009. IEEE.

[6] Matthew L Bolton and Ellen J Bass. Evaluating human-human communication protocols with miscommunication generation and model checking. In *NASA Formal Methods*, pages 48–62. Springer, 2013.

[7] G. E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71 (356):791–799, 1976.

[8] William J Clancey, Patricia Sachs, Maarten Sierhuis, and RON VAN HOOF. Brahms: Simulating practice for work systems design. *International Journal of Human-Computer Studies*, 49(6):831–865, 1998.

[9] S.T. Cluff. *A Unified Approach to GPU-Accelerated Aerial Video Enhancement Techniques*. PhD thesis, Brigham Young University, 2009.

[10] J.L. Cooper. Supporting flight control for UAV-assisted wilderness search and rescue through human centered interface design. Master's thesis, Brigham Young University, 2007.

[11] M. L. Cummings, C. E. Nehme, J. Crandall, and P. Mitchell. *Developing Operator Capacity Estimates for Supervisory Control of Autonomous Vehicles*, volume 70 of *Studies in Computational Intelligence*, pages 11–37. Springer, 2007.

[12] Mary L Cummings, Carl E Nehme, Jacob Crandall, and Paul Mitchell. Predicting operator capacity for supervisory control of multiple uavs. In *Innovations in Intelligent Machines-1*, pages 11–37. Springer, 2007.

[13] C.H. Engh. *A see-ability metric to improve mini unmanned aerial vehicle operator awareness using video georegistered to terrain models.* PhD thesis, Brigham Young University, 2008.

[14] M. A. Goodrich, B. S. Morse, D. Gerhardt, J. L. Cooper, M. Quigley, J. A. Adams, and C. Humphrey. Supporting wilderness search and rescue using a camera-equipped mini UAV. *Journal of Field Robotics*, 25(1-2):89–110, 2008.

[15] M.A. Goodrich, B.S. Morse, D. Gerhardt, J.L. Cooper, M. Quigley, J.A. Adams, and C. Humphrey. Supporting wilderness search and rescue using a camera-equipped mini UAV. *Journal of Field Robotics*, 25(1-2):89–110, 2008.

[16] M.A. Goodrich, B.S. Morse, C. Engh, J.L. Cooper, and J.A. Adams. Towards using unmanned aerial vehicles (UAVs) in wilderness search and rescue: Lessons from field trials. *Interaction Studies*, 10(3):453–478, 2009.

[17] Michael A. Goodrich. On maximizing fan-out: Towards controlling multiple unmanned vehicles. In M. Barnes and F. Jentsch, editors, *Human-Robot Interactions in Future Military Operations*. Ashgate Publishing, Surrey, England, 2010.

[18] Curtis Michael Humphrey. *Information abstraction visualization for human-robot interaction.* PhD thesis, Brigham Young University, 2009.

[19] L. Lin, M. Roscheck, M.A. Goodrich, and B.S. Morse. Supporting wilderness search and rescue with integrated intelligence: Autonomy and information at the right time and the right place. In *Artificial Intelligence, Twenty-Fourth AAAI Conference on*, Atlanta, Georgia, 2010. AAAI.

[20] P. M. Mitchell and M. L. Cummings. Management of multiple dynamic human supervisory control tasks. In *Command and Control Research And Technology, 10th International Symposium on*, Washington DC, 2005.

[21] JJ Moore, R Ivie, TJ Gledhill, E Mercer, and MA Goodrich. Modeling human workload in unmanned aerial systems. In *Formal Verification & Modeling in Human-Machine Systems, AAAI Spring Symposium Series*, Palo Alto, California, 2014. AAAI.

[22] Neville Moray. Subjective mental workload. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 24(1):25–40, 1982.

[23] B.S. Morse, D. Gerhardt, C. Engh, M.A. Goodrich, N. Rasmussen, D. Thornton, and D. Eggett. Application and evaluation of spatiotemporal enhancement of live aerial video using temporally local mosaics. In *Computer Vision and Pattern Recognition, IEEE Conference on*, pages 1–8, Anchorage, Alaska, 2008. IEEE.

[24] R. Murphy, S. Stover, K. Pratt, and C. Griffin. Cooperative damage inspection with unmanned surface vehicle and micro unmanned aerial vehicle at hurrican Wilma. IROS 2006 Video Session, October 2006.

[25] R. R. Murphy and J. L. Burke. The safe human-robot ratio. In M. Barnes and F. Jentsch, editors, *Human-Robot Interaction in Future Military Operations*, chapter 3, pages 31–49. Ashgate Publishing, 2010.

[26] Allen Newell. *Unified theories of cognition*, volume 187. Harvard University Press, 1994.

[27] N.D. Rasmussen, D.R. Thornton, and B.S. Morse. Enhancement of unusual color in aerial video sequences for assisting wilderness search and rescue. In *Image Processing (ICIP), 15th IEEE International Conference on*, pages 1356–1359, San Diego, California, 2008. IEEE.

[28] Neha Rungta, Guillaume Brat, William J Clancey, Charlotte Linde, Franco Raimondi, Chin Seah, and Michael Shafto. Aviation safety: modeling and analyzing complex interactions between humans and automated systems. In *Application and Theory of Automation in Command and Control Systems, 3rd International Conference on*, pages 27–37, Naples, Italy, 2013. ACM.

[29] Dario D Salvucci and Niels A Taatgen. Threaded cognition: an integrated theory of concurrent multitasking. *Psychological review*, 115(1):101, 2008.

[30] Tim J. Setnicka. *Wilderness Search and Rescue.* Apalachian Mountain Club, Boston, MA, 1980.

[31] *Integration of Civil Unmanned Aerial Systems (UAS) into the National Airspace System (NAS) Roadmap.* U.S. Department of Transporation Federal Aviation Administration, first edition edition, 2013.

[32] Christopher D Wickens. Multiple resources and performance prediction. *Theoretical issues in ergonomics science*, 3(2):159–177, 2002.