2013 IEEE International Conference on Systems, Man, and Cybernetics

Modeling UASs for Role Fusion and Human Machine Interface Optimization

TJ Gledhill, Eric Mercer and Michael A. Goodrich
Computer Science Department
Brigham Young University
Provo, UT

***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 [1–3]. 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.**

***Index Terms*—model checking, human machine interfaces, Java PathFinder, Unmanned Aerial Systems, Wilderness Search and Rescue**

I. Introduction

Most existing Unmanned Aerial Systems (UASs) require two or more human operators[1, 2]. 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 [4], there is considerable interest in reducing the required number of humans and reducing human workload using improved autonomy and enhanced user interfaces [3, 5, 6].

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) [6]; and second, there is a host of modeling information about how WiSAR is currently performed [7]. 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 [8]. 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.

II. 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 [9]. To study the Uberlingen collision, an in air collision of two commercial

978-1-4799-0652-9/13 $31.00 © 2013 IEEE
DOI 10.1109/SMC.2013.332

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 [10]. 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 [11]. 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 [12]. 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 [1]. 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. [7]. 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 [13]; 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.

III. WiSAR UAS Domain

Wilderness search and rescue often occurs in remote, varying, and dangerous terrains. According to [14], 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.

IV. 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. [7]. 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 [15].

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 [15] 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:

$$Actor = (S, s_0, \Sigma_A \cup \Sigma, \Lambda_A, T) \tag{1}$$

$$Event = (S, S_0, \Sigma_A \cup \Sigma_S, \Lambda_A, T) \tag{2}$$

$$T : S \times \Sigma \Rightarrow S \times \Lambda_A \tag{3}$$

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.

V. 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.

A. 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.

B. 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.

C. 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.

D. 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.

VI. 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.

A. 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.

The next sections are dedicated to describing the Actor state machines with a generalized description of their relative tran-

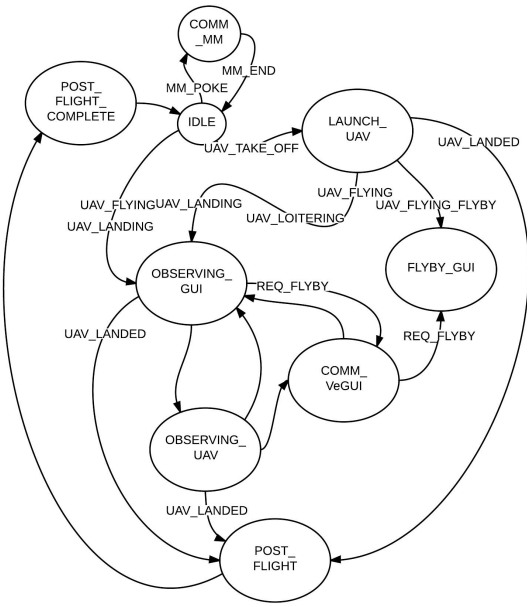

Fig. 1. UAV Operator DiRG. Excludes transition output.

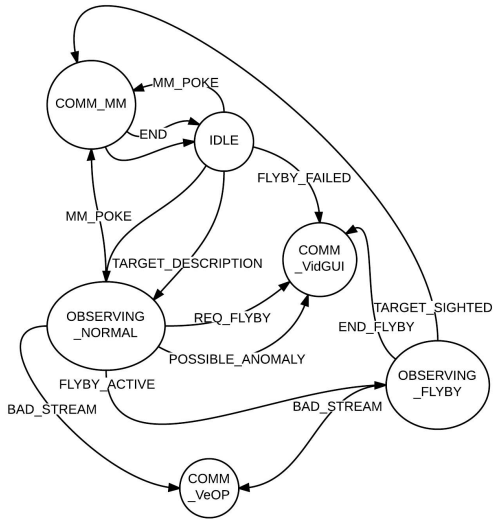

Fig. 2. Video Operator DiRG. Excludes transition output.

sition matrixes. We omit several of the previously mentioned Actors in the interest of space.

1) UAV Operator (VeOp): As illustrated in Figure 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.

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.

2) Video Operator (VidOp): As illustrated in Figue 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.

3) 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.

4) 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.

B. 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]

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

C. 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 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.

D. Case Study: Anomaly Detection

The scenario illustrated in figure 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.

VII. Results

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

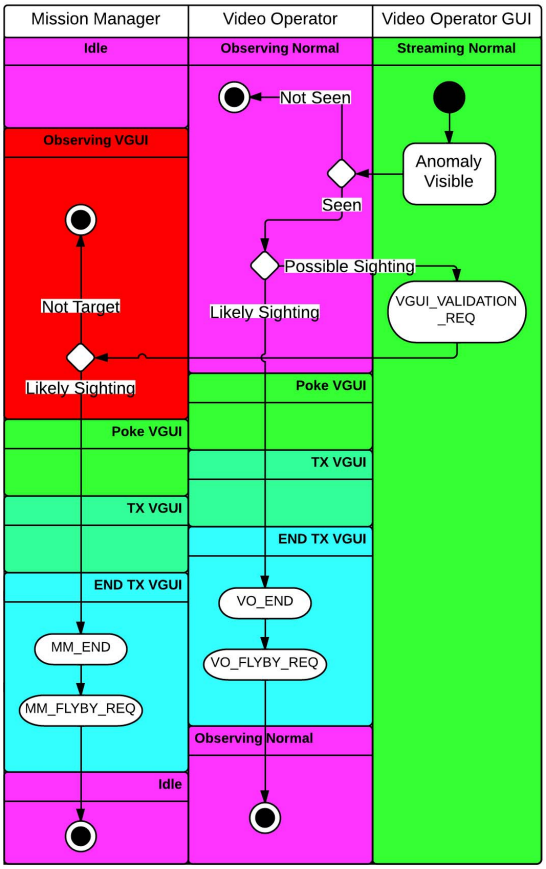

Fig. 3. Anomaly Detection Model: Swim lanes represent actors. Arrows represent input/output. Colored sections represent actor states.

A. 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.

B. 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.

VIII. 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.

References

[1] 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*, vol. 25, no. 1-2, pp. 89–110, 2008.

[2] 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.

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

[4] R. R. Murphy and J. L. Burke, "The safe human-robot ratio," in *Human-Robot Interaction in Future Military Operations*, M. Barnes and F. Jentsch, Eds. Ashgate Publishing, 2010, ch. 3, pp. 31–49.

[5] P. M. Mitchell and M. L. Cummings, "Management of multiple dynamic human supervisory control tasks," in *10th International Command and Control Research And Technology Symposium*, 2005.

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

[7] 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 unmanned aerial vehicle wilderness search support," *Journal of cognitive engineering and decision making*, vol. 3, no. 1, pp. 1–26, 2009.

[8] G. E. P. Box, "Science and statistics," *Journal of the American Statistical Association*, vol. 71, no. 356, pp. 791–799, 1976.

[9] *Aviation Safety: Modeling and Analyzing Complex Interactions between Humans and Automated Systems*, ser. ATACCS, 2013.

[10] *A Synergistic and Extensible Framework for Multi-Agent System Verification*, ser. AAMAS, 2013.

[11] *Enhanced operator function model: a generic human task behavior modeling language*, ser. SMC'09. IEEE Press, 2009.

[12] *Evaluating Human-human Communication Protocols with Miscommunication Generation and Model Checking*, 2013.

[13] C. M. Humphrey, "Information abstraction visualization for human-robot interaction," Ph.D. dissertation, 2009.

[14] T. J. Setnicka, *Wilderness Search and Rescue*. Appalachian Mountain Club, 1980.

[15] *Using Foirmal Verification to Evaluate Human-Automation Interaction: A Review*, 2013.