Modeling UASs for Role Fusion and Human Machine Interface Optimization

TJ Gledhill, Jared Joseph Moore, Robert Ivie, Eric Mercer and Michael A. Goodrich

***Abstract*—Currently, a single Unmanned Aerial System (UAS) requires several humans managing different aspects of the problem. Human roles often include vehicle operator, payload expert, and mission manager [?], [?], [?]. 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 the unmanned aerial vehicle. This paper presents models of these roles for UAS-enabled Wilderness Search and Rescue (WiSAR) as well as an encoding of these models in Java. This yields a precise formalization of the individual WiSAR roles and allows behavior of the roles to be model-checked by Java Pathfinder to establish bounds and trends in the models. Results from the modeling activity and tests in Java Pathfinder indicate (a) that it is necessary to clearly identify an appropriate level of modeling abstraction and (b) that the resulting model can be evaluated in such a way that it provides insight into ways to reduce human workload**

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

I. Introduction

Problem Statement: UASs require several human operators to monitor and administer.

Most existing Unmanned Aerial Systems (UASs) require two or more human operators[?], [?]. In Unmanned Aerial Systems (UASs) the standard practice requires 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 that this is a desirable organization [?], there is considerable interest in reducing the required number of humans and reducing human workload using improved autonomy and enhanced user interfaces [?], [?], [?]. This paper approaches this goal through a multi-step 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 completion of the first three steps with an emphasis on the first step. Since the authors have explored UAS-enabled Wilderness Search and Rescue (WiSAR) [?] and since there is a host of modeling information about how WiSAR is currently performed [?], this paper emphasizes this socially relevant application of UAS technology. The UAS-enabled WiSAR systems produced by this research requires three humans, two GUIs, and a single UAV. Modeling is essentially a process of abstraction, choosing which elements of a system are essential and which are not [?]. 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 finite state machines, specifically a Mealy machine, and model different WiSAR roles as state machines. These models explicitly encode key aspects of the various roles, 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. Given the models, along with a discussion of limitations of the models, we can use model checking to verify the model and identify bounds and trends in the models. The model checking is performed using Java Pathfinder (JPF), which is convenient because JPF runs the model checking on the compiled Java code. This means that the model can be run directly by JPF without a complicated conversion process.

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 [?]. To study the Uberlingen collision, an in air collision of two commercial passenger planes, Neha 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. For example, autonomous systems must work in conjunction with humans by either providing true autonomy or simply providing direction to the human operator.

One critical aspect [?] of NASA ARCs work, carries over into our own. Depending on a tasks duration, the task might cause a crash, cause a near miss, or make the situation completely safe. The biggest advantage Brahms has over Java comes from its robust structure. However, Brahms must be translated to Java before using JPF, a step we avoid by implementing our model in Java directly.

Mathew Bolton and Ellen Bass used Enhanced Operator Function Model (EOFM) 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 [?]. 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 [?]. This top down approach makes formulating a large process is easier in EOFM than it is in Java, as well as promoting readability of the code. Our model uses a bottom up approach that reduces readability. It does however provide enhanced malleability through a completely customizable protocol.

Wilderness Search and Rescue is primarily concerned with finding people who have become lost in rugged terrain. Research has shown that unmanned aerial vehicles (UAVs) facilitate this work. In principle the vehicle is used to search for victims in areas that would be difficult for ground teams to reach. Michael Goodrich and his colleagues tested the effectiveness of these types of operations [?]. Their research shows that altitude and video clarity facilitate visual detection. Furthermore, they suppose that these attributes could be enhanced if the roles of the UAV operator and video operator were combined. This situation puts an added drain on the new operator role, because he or she must dedicate energy to a larger amount of tasks.

III. WiSAR UAS Domain

Wilderness search and rescue represents search and rescue efforts in remote, varying, and dangerous terrains. According to [?], there are four core elements of a WiSAR operation.

$$\mathit{Locate} \Rightarrow \mathit{Reach} \Rightarrow \mathit{Stabilize} \Rightarrow \mathit{Evacuate}$$

Fig. 1. Core SAR Elements

The WiSAR UAS operates within this first element, so tasks in this first element dictate the key elements of the modeling exercise.

During the locate phase, 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 (MM), Operator (OP), and Video Operator(VO). These constitute the three human roles in the team. Supporting these human roles are two intelligent user interfaces, the Operator GUI (OGUI) and the Video Operator GUI (VGUI); 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 (PS) model. The parent search model represents the entire command structure for the search and rescue operation. In the next section, we begin modeling 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 environment 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 the literature; these models tend to focus on environmental realism, encoding things like terrain, wind, etc.

IV. Simulating the WiSAR UAS

Do to the complexity of the WiSAR UAS environment we desired to simplify the model simulation as much as possible. To do this we constructed a basic simulation framework. The simulation framework is encapsulated into a single Java class: Simulator which implements the singleton pattern. This section will discuss components of this simulation framework.

In order to simulate critical aspects of the WiSAR UAS model it became necessary to model all of the different roles, interfaces, and objects in a simplified way that would facilitate the desired interactions. We chose to do this using state machines. We refer to these state machines as actors. Actors can be thought of as Mealy machines represented by equations 1, 2. Where S is a set of states, S_0 the start state, Σ_A the set of all Actor inputs, Λ_A the set of all Actor outputs, and T a transition matrix which specifies the outputs for any state transition.

$$\mathit{Actor} = (S, S_0, \Sigma_A, \Lambda_A, T) \tag{1}$$

$$T : S \times \Sigma_A \Rightarrow S \times \Lambda_A \tag{2}$$

Events are Moore State machines that spontaneously change state provided the appropriate conditions of the simulation are met.

Simulating the passage of time provides a metric whereby we can monitor the interactions between Actors in our model. Each action an Actor can take is assigned a range spanning the minimum and maximum time required to complete thereby inserting non-determinism into the system. We control the level of non-determinism through the simulator with a DurationGenerator class. This class has five different duration settings: MIN, MAX, MIN_OR_MAX, $MIN\_MEAN\_OR\_MAX$, and $RANDOM$. MIN and MAX presenting a single option, MIN_OR_MAX presenting two options and so on. The number of state spaces this produces can be seen in table [] in our results.

To help give the appearance of concurrency during the processing of a timestep we implemented an output storage and distribution class called PostOffice. Output is sent to the PostOffice using these methods:

```
addOutput(IData data, String name);
addObservation(IData data,
    String actor_name);
```

The PostOffice uses Actor names to deliver outputs. The PostOffice keeps all output in a temporary hash map. At the desired time the simulator will tell the PostOffice to make the temporary output current. At this point the current output is destroyed and the temporary output is emptied into the current output. This prevents output received during the same processing step from being received before its due time. An Actor may also post observations which are output that must be requested by other Actors. Actors pull input and observations from the simulator using these methods:

```
ArrayList<IData> getInput(String);
ArrayList<IData> getObservations(String);
```

The PostOffice also has the ability to link output and observations for multiple actors by mapping Actor names to one another. This feature was added to allow sub-actors to automatically share input with parent Actors and to allow observation of a parent Actor to return the observations of all of its children.

Each Actor implements the IActor interface with the methods: processNextState and processInputs. The processNextState moves the actor into its next state, generating outputs and setting the next state to a default value and duration. The processInputs method then looks at its inputs and determines what the next state should be and when it should occur. This method also generates additional outputs.

```
public interface IActor {
    public int nextStateTime();
    public void processNextState();
    public void processInputs();
    public String name();

}
```

Each Event implements the IEvent interface. The IEvent interface is fairly similar to the IActor interface except it has the method getCount. This makes it possible to insert multiple events of the same type without having to instantiate entirely new objects. Events are also given higher priority than Actors ensuring that all applicable events execute prior to any actors on a given time-step.

```
public interface IEvent {
    public int getNextTime();
    public void processNextState();
    public void processInputs();
    public int getCount();
}
```

Each Actor has a method getNextStateTime. This returns the global time-step when the actor will change state. This assumes that everything in the system requires at least one time step to complete. The simulator takes the minimum value of all the nextStateTimes and advances the global clock to that time. If however the time outputted is 0 then the simulator treats that as a signal that the model is done processing and terminates. This eliminates unnecessary processing by ensuring the simulator only processes time steps when the system is changing..

Figure 2 portrays the execution of a time step. The simulator first processes the next state of any events that undergo a state change in that time. The outputs of those events are stored in the Post Office to be accessed by the Actors when applicable. Next the simulator calls processInboundData on the PostOffice to move the temporary output to the current output. The Actors then each call processInputs. Once each Actor has evaluated its inputs and made any changes the simulator checks if there are any future time-steps to evaluate and if not it terminates the imulation.

Fig. 2. Basic simulation timestep

To help facilitate the simulation three other interfaces exist. They are the ITeam, IData, and IState interfaces. The ITeam interface acts as a wrapper around a team of Actors. The team represents all of the Actors that the simulator will interface with during the simulation and allows the simulator to call a single method on the team to perform actions on all of the Actors. The IData and IState interfaces are used to define common data types for passing information between Actors. We use enumerations to define outputs and states specific to an Actor, these enumerations implement the correct interface for the type of information they carry.

```
public interface ITeam {
    public int getNextStateTime(int);
    public void processNextState();
    public void processInputs();
}

public interface IData {
    public String name();
}

public interface IState {
    public String name();
}
```

The Simulator class provides the following helper methods that are accessible to the Actors and Events.

```
int getTime();
void setTime(int time);
int getNextStateTime();
void addOutput(String actor_name,
    IData input);
void addOutputs(String actor,
    ArrayList<IData> inputs);
ArrayList<IData> getInput(String actor);
void linkInput(String parent,
    String child);
void linkObservations(String parent,
    String child);
ArrayList<IData> getObservations(String);
void addObservation(IData data,
    String actor);
void addObservations(ArrayList<IData> data,
    String actor);
void addEvent(IEvent event);
int duration(Range range) ;
```

This modeling framework imposes no other constraints on the model besides the simulation structure defined by these interfaces. The model implementation has the full power of the Java language for all internal decision making. We believe that this will ensure that the framework will be capable of simulating any and all aspects of the UAS model. This contrasts with semantically constrained modeling languages such as Brahms which, due to the semantics, may not fully map to the model.

V. WiSAR UAS Model

This section describes how we modeled the WiSAR UAS using the framework that was previously mentioned. We have broken down this detailed description into two main categories, Actors and Events, followed by a brief discussion of Java asserts and a case study drawn from WiSAR.

A. Actors

Choosing the core actors was not a trivial task, because we were looking for a level of abstraction that gave results without adding unwanted complexity to the model. The final model simulates the following team of actors: the parent search (PS), mission manager (MM), UAV operator (OP), video operator (VO), operator GUI (OGUI), video operator GUI (VGUI), and the UAV.

Our models of the human roles use specific states for their communication. We will describe these states once and then refer to them as a single communication state. Typically before a human will begin to communicate they have received some input, direct or indirect, that their communication is being received. We have chosen to model this as a POKE state. When communicating a human role will enter a poke (POKE) state. In this state the human waits until it receives an acknowledgement. If the acknowledgement is not received then the communication does not occur. After the acknowledgement the human will move 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 human role receives a poke then it will immediately respond with a busy or an accept. If the human accepts the poke then it enters the receive (RX) state. The human will not leave this state until the end input is received or 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. For this reason we use multiple communication routines to communicate multiple items.

1) Parent Search (PS): This Actor represents the IC and is heavily abstracted in our model. The PS has two main states $IDLE$ and communicating with the MM. It contains no logic it simply passes output from events to the MM and accepts input from the MM. The events that it listens to are NewSearchAOIEvent, TargetDescriptionEvent, and Terminate-SearchEvent.

2) Mission Manager (MM): The MM Actor is made up of an $IDLE$, $OBSERVING\_VGUI$, and four communication states: PS, OP, VO, and VGUI. Initially the MM is idle. After receiving a new search command and target description the MM distributes the received information. During the search the MM will periodically check the VGUI to see if there are anomaly verification requests. For each of these requests the MM will decide if it deserves a flyby or if it is a false positive. While watching for anomaly verification requests the MM is also listening for input from either the PS, OP, or VO.

3) UAV Operator (OP): The OP Actor has the following states: $IDLE$, $OBSERVING\_GUI$, $FLYBY\_GUI$, $OBSERVING\_UAV$, $LAUNCH\_UAV$, $POST\_FLIGHT$, and communication. The OP communicates with the MM, VO, and OGUI. Initially the OP is idle. After receiving a new search command from the MM the OP will construct a flight plan on the OGUI. When this is complete the OP will then launch the UAV. While in the launch state the OP is observing the UAV, when the UAV completes its take off the OP moves to observing the GUI. While the UAV is airborne the OP will continually move between observing the UAV and observing the GUI. Different UAV output is available to the OP depending on the form of observation. The OP will respond through the GUI to any problems that are noticed. During the flight the OP is also listening for input from the PS and VO. If there are flyby requests on the OGUI then the OP may choose to enter a flyby mode. This implies a high cognitive load on the OP while positioning the UAV over the specified anomaly. The OP will remain in flyby mode until the VO 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.

4) Video Operator (VO): The VO Actor has the following states: $IDLE$, $OBSERVING\_VGUI\_NORMAL$, $OBSERVING\_VGUI\_FLYBY$, and communication. The VO communicates with the MM, OP, and VGUI. Initially the VO is idle. After receiving a target description and the search information the VO moves to normal GUI observation. While observing the GUI the VO watches for anomalies, each time an anomaly is visible the VO decides if the anomaly is seen. If it is seen the VO decides if it is an unlikely, possible, or likely sighting. This is done with probabilities related to the type of anomaly. A false positive anomaly has a lower probability of being a likely sighting while a true positive anomaly has less probability of being an unlikely sighting. If the anomaly is classified as possible then the VO makes a validate sighting request for the MM. If the anomaly is a likely sighting then the VO requests a flyby from the OP. When the OP begins a flyby request the VGUI will signal this to the VO who will enter the flyby state. In this state the VO watches for the anomaly, due to the nature of the flyby the VO can now make an informed decision which has much higher probabilities of being correct. After deciding if it is the target the VO signals through the VGUI that the flyby is finished. If the sighting is confirmed the VO reports to the MM, otherwise the VO returns to normal GUI observation.

5) Operator GUI (OGUI): The OGUI Actor has two states: $NORMAL$ and $ALARM$. The OGUI communicates directly with the UAV and VGUI Actors directly without the need for poke, tx, and end. The default function of the OGUI is to observe the UAV. The OGUI keeps internal variables of all the UAV and VGUI data that it tracks, all of this data is available through observation of the OGUI. If it detects an error with the UAV outputs such as low battery, no flight plan, low height above ground, or lost signal the OGUI will enter the alarm state. This state implies that there are visible warnings on the screen to alert the OP of the problem. The OGUI listens for OP input or changes in the UAV output to signal that the problem has been dealt with before moving into the normal state.

6) Video Operator GUI (VGUI): The VGUI Actor has two states: $NORMAL$ and $FLYBY$. The VGUI has the same communication model as the OGUI. The default function of the VGUI is to present anomalies to the VO. The VGUI presents anomalies by listening for input from TruePositiveAnomalyEvent and FalsePositiveAnomalyEvent. The VGUI tracks the visible anomalies, flyby requests, and anomaly verification requests. The VGUI moves to the flyby state after the OP initiates a flyby. In this state the VO can observe that the UAV attempting to relocate and obtain better video quality of a previously seen anomaly. The VGUI will remain in this state until it receives the finished command from the VO.

7) 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. We also defined several sub-Actors for the UAV which we felt must be modeled. The first of these is the UAVBattery which has 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 $current\_time + low\_battery\_threshold$. The next 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 OGUI 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.. When the UAV has flown the flight plan for the specified duration then the flight plan enters the complete state.

The next sub-Actor is the UAVHeightAboveGround which has the following states: $INACTIVE$, $GOOD$, LOW, and $CRASHED$. Its initial state is inactive.. This Actor represents the UAVs height above the ground (HAG). Its main purpose is to listen for low height above ground events and to crash the UAV if a response is not received in time. Once the UAV is airborne the HAG will remain good unless input is received from the LowHAGEvent. The event moves the UAVHeightAboveGround into the low state. If the UAV receives a modified flight plan then the state is changed back to good, otherwise the UAV will crash when it receives the second output from the LowHAGEvent..

The next sub-Actor is the UAVSignal which has the folowing states: OK and $LOST$. Initially it is set to OK. This represents the UAVs communication with the OGUI. Its main purpose is to handle input from LostSignalEvent. When this event occurs the signal enters the lost state until receiving input to return to the ok state.

The next sub-Actor is the UAVVideoFeed which has the following states: $IDLE$, OK and BAD. Initially it is set to IDLE. This represents the quality of the UAV video feed. Its purpose is to handle BadVideoFeedEvents. When the UAV takes off the video feed moves to OK. When event input is received the video feed moves to bad. This is only returned to OK after the UAV has landed and entered the ready state again.

The last sub-Actor is the FlybyAnomaly which has the following states: $IDLE$, $ANOMALY\_NOT\_SEEN$, $ANOMALY\_SEEN$, $PAUSED$, and $FLYBY\_COMPLETE$. Initially it is set to idle. This represents when the UAV is performing a flyby of a previously seen anomaly. When a flyby is begun the FlybyAnomaly Actor moves into the anomaly not seen state with a next state of anomaly seen after a specified duration. When it moves into the anomaly seen state, where it remains until the end flyby command is received, we assume that the anomaly is visible from the UAV and that it will remain visible until the flyby is complete. When the end flyby command is received the FlybyAnomaly briefly enters the flyby complete state before returning back to idle. If the UAV needs to land or loiter for any reason during the flyby it will enter a paused state.

B. Events

We used events for several different abstractions. The first class of abstraction is the common event. These events are common and expected within the WiSAR UAS. The second class of abstraction is the uncommon event. These are events which are outside normal operation. They represent unforeseen problems that can occur at any time. In essence these types of events push the actors into state spaces that they would never reach during normal operation. Here is our list of events:

1) NewSearchAOIEvent: This event outputs a new search area of interest (AOI) to the PS. The OP cannot perform their duties until this output is received.

2) TargetDescriptionEvent: This event outputs a target description to the PS. The VO cannot perform their duties until this output is received.

3) TerminateSearchEvent: This event outputs a terminate command to the PS. The parent search may terminate for a multitude of reasons and at anytime.

4) LowHAGEvent: This event is only available while the UAV is airborne, it outputs height above ground is low. This event sends a second output after a specified period indicating that the UAV crashed. This output is ignored if the flight plan was modified.

5) LostSignalEvent: This event is only available while the UAV is airborne. It first sends a lost signal output. After a specified duration it then sends a signal ok output.

6) TruePositiveAnomalyEvent: Only available while the UAV is flying its flight plan. Outputs an initial signal that the anomaly is visible. Outputs a second signal that the anomaly is no longer visible.

7) FalsePositiveAnomalyEvent: Same process as TruePositiveAnomalyEvent.

8) BadVideoFeedEvent: Only available while the UAVVideoFeed is ok. Outputs that the video quality is now too poor to be effective.

C. Asserts

The more complicated the model the more things that can go wrong. Some errors are caused by coding bugs and some are flaws in the model. Flaws in the model are extremely valuable, but it can be challenging to tell them apart. To catch these errors 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.

In our model we have used asserts in two ways. The first is detection of 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 requiring a specific state and input before the next task can be performed. By looking at inputs received we are able to tell if actors are out of sync with one another. An example of this is the OP_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.

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

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

This scenario represents what should occur when the video operator believes they see the target on the video GUI. 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.

	1	2	3	4
Time	00:00:01	00:00:00	00:00:06	00:00:52
New States	124	16	3911	51344
Visited States	0	0	240	5193
Backtracked	124	16	4151	56537
End	1	1	72	828
Max Depth	124	16	129	129

VI. Results

In this section we present empirical results to demonstrate that WiSAR UAS is modeled effectively using the approach we have taken. We will demonstrate this using a brief case study extracted from WiSAR UAS. We will then give an analysis of the JPF results we obtained when we verified the model.

A. Model Construction

As mentioned in section V we use seven core actors in our model. The anomaly detection effects each of these actors. Because we already had the core actor classes setup before we defined this part of the model it was a simple process of adding additional states and transitions to our actors. For the anomalies themselves we created an anomaly event which sent output directly to the video GUI. We wanted the operator to non-deterministically choose if an anomaly was not seen, seen as a possible target, or seen as a likely target. We were able to acheive this by generating a random probability within a designated range. This means that when run by JPF each of those probabilities will be explored. A small portion of the model can be seen in figure 3.

B. JPF Results

We were very pleased with the integration into JPF. We found that using JPF made it much easier to find bugs in the model. As mentioned previously we performed our verification with different levels of non-determinism through task durations. This data can be seen in table **??** The New States represent... Visited States represent.... Backtracked means... End means... Max Depth means...

When we began to verify the model one of the first outputs we discovered was infinite loops. This told us that something was wrong. Typically this was caused by a failure to send the correct output, in one case the infinite loop occurred when the actor would change states too early. We discovered that the default maximum duration for that state was set too low. This represented a flaw in our model.

C. Modeling Challenges

Another example was the addition of sub-actors to the UAV actor. Sub-actors are parts of a more complex actor, in this case the UAV. The sub-actors for UAV are the battery, flight plan, height above ground, and the signal. Their inputs and outputs are handled differently than a normal actor. These sub-actors inputs are linked within the simulator to the UAV. The UAV also has specific modifications that allow it to process all sub-actors before completing its own processing. This level of abstraction reduced the very complex UAV actor into multiple, simpler actors. This gave us confidence that this modeling framework is robust enough to model anything we may need. It also gave us a pattern for modeling the different levels of abstraction that we desire.

While constructing this portion of our model we used mock actors to represent some of the logic which belonged to the core classes. Mock actors are actor classes which always give the same output. This allowed us to get the model working before we had finished coding. This proved to be very useful and we can see how it can be an effective tool for future verification. If we are not interested in the non-deterministic states of one actor, or a portion of that actors state space, that portion of the model can be replaced with a mock actor.

We did run into a few issues while coding the model. One problem we found was inconsistency in our human memory modeling. Without a strict grammar it is possible to do the same thing in many different ways which leads to flaws in the model. We also found that if we did not move logic into sub-actors the complex actors began to be unmanageable. In some cases moving logic into a sub-actor can be a difficult task. Another problem we found was the lack of asserts for invalid states and inputs. These asserts are a crucial part of determining if the model is performing correctly yet we had no way of enforcing that these asserts were placed into the code.

VII. Future Work

As stated previously this work is the basis for research on human machine interfaces to support combined human roles that reduce operator workload. To achieve this end we feel that the modeling framework we have presented would be more beneficial if it formalized some of the practices we found to be useful while modeling. We would also like the simulation to check the connectivity of the model before running the simulation to ensure that the transition matrix for each actor is complete.

Acknowledgment

The authors would like to thank Neha Rungta of NASA Ames Intelligent Systems Division for her help with JPF and Brahms.

References