JUCAV Deck Handling System

Ryan Foley

Tuesday, December 12, 2006

ENPM 643

Table of Contents

Table of Contents 2
Project Overview 3
Background Information 4
JUCAV 4
OmniBird 5
    SOS Processor 6
Controller 6
High Level System Layout 7
Goals and Scenarios 7
Goals and Scenarios 8
UML 9
Use Case Diagram 9
Activity Diagrams 10
    Use Case #1: Collect Data 10
    Use Case #2: Process Image 11
    Use Case #3: Generate Control Commands 12
    Use Case #4: Control Omnibird 14
    Use Case #5: Control JUCAV 16
Class Diagram 17
Requirements 18
Traceability Matrix 19
LTSA 20
Verification Plan 21
LTSA: MSC High Level 22
LTSA: LTS Diagrams 23
LTSA: MSC Traces 24
LTSA: MSC Traces 25
LTSA: Validation and Verification 26
LTSA: Validation and Verification 27
Conclusions 28
References 29
Appendix 30
Final LTSA 30
    XML Code 30

Project Overview

For the past decade the Defense Advanced Research Projects Agency has been developing the first unmanned combat aerial vehicle. The program is co-sponsored by the US Navy and US Air force and has been called the Joint Unmanned Combat Air System of J-UCAS. In order for the Navy to maintain some control over this program the Joint Unmanned Combat Aerial Vehicle (JUCAV) must be able to land and function on an aircraft carrier. My current employer, Genex Technologies, has been functioning as a subcontractor in charge of developing a novel deck taxiing system. Genex has been developing an optical sensor called the Omnibird that will serve as the eyes of the JUCAV on the flight deck.

The deck taxiing system has the ultimate goal of allowing the JUCAV to function exactly as a manned fighter would function. The Navy wants to keep the same flight deck crew and personnel in place for years to come. This means that the Omnibird system must find the flight deck personnel in charge of the JUCAV; capture his image, distinguish the commands he is giving the vehicle, and relay them to the JUCAV.

The scope of this project is to design the deck taxiing system and verify that the JUCAV can be under the deck controllers command at all times. This project also serves as an add-on to the work done at Genex Technologies regarding their Omnibird system.

Background Information

The JUCAS is an extremely complex system and therefore large sub-systems will not be touched in this project. Below is a description of the main sub-systems that are included in this project.

JUCAV

The JUCAV is by far the most interesting and complex sub-system in this system. The JUCAV is intended to enter military service in 2010 under the US Navy and US Air Force. Currently there are two competing contractors working on different versions of the same vehicle. Boeing is developing the X-45 and Northrop Grumman is developing the X-47. Both prototypes are extremely similar and for the purposes of this project will be identified simply as the JUCAV.

Both planes will share the exact same operating system under development by the US Navy. The future goal of the JUCAS system is to provide an unmanned fighter that can effectively do everything. The JUCAV can be used for surveillance, reconnaissance, strategic bombings, air to air refueling, and anything else the Navy and Air Force can dream up. One of the main goals of the system is to allow the JUCAV to deploy munitions deep over hostile territories as a first strike. This way there is no threat to US

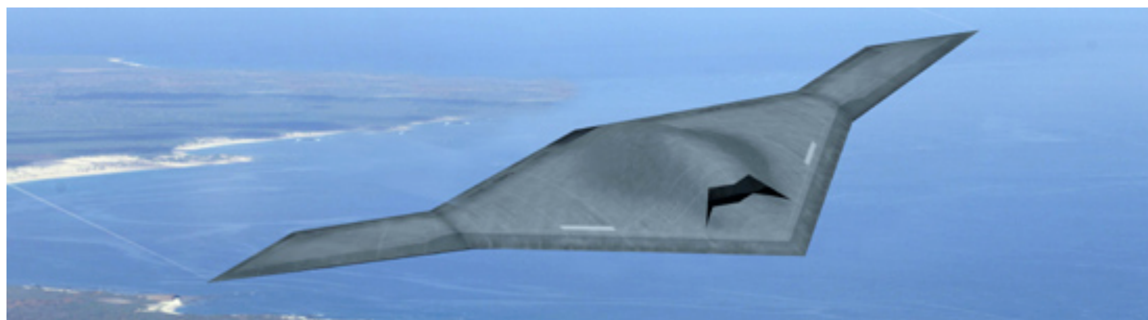

Figure 1: Northrop X-47 Prototype Artist Rendition

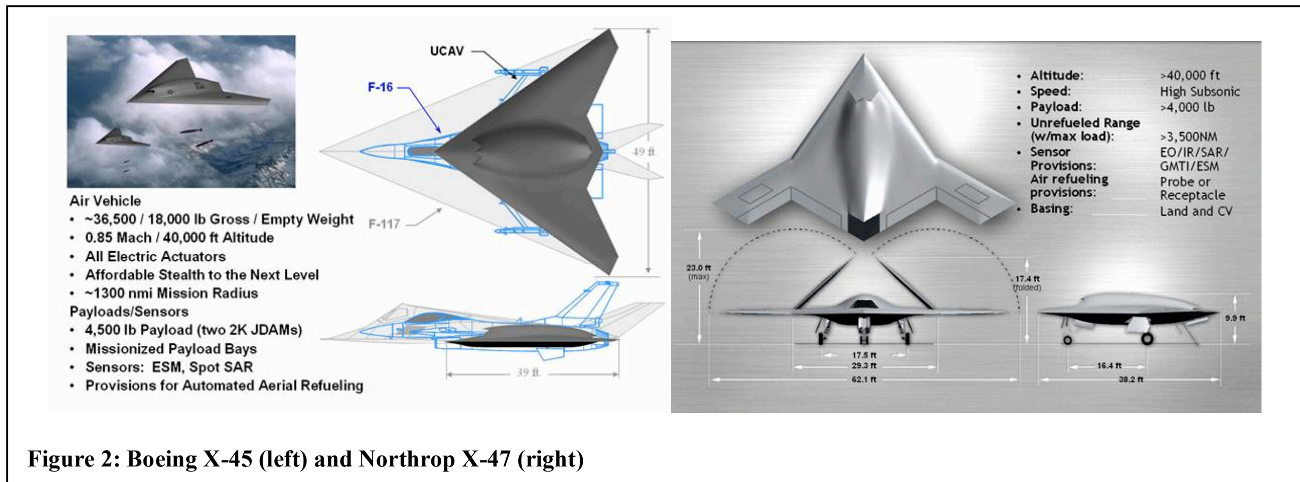

Figure 2: Boeing X-45 (left) and Northrop X-47 (right)

personnel.

Figure 2 below shows a comparison of the current iteration X-45 and X-47 JUCAV's in development. Both JUCAV's maintain similar performance specifications as you can see from Figure 2. Both have an altitude of roughly 40,000 ft, a payload of 4,000 lbs, and fly at speeds just below the speed of sound. They both maintain similar profiles and resemble B-2 Stealth Bombers in their appearance. The flying wing design is used to reduce the JUCAV's radar signature. This feature coupled with the use of radar absorbing materials in the fuselage design will make the JUCAV the most advanced stealth aircraft in the world.

OmniBird

Genex Technologies Omnibird is quite a remarkable design. The biggest accomplishment is its size. It is all fits into a two inch cube! The Omnibird system makes use of a miniature sensor with a 1.3 megapixel resolution. This sensor is mounted to specialized optics that allows the Omnibird to see from ranges of 10 feet to 250 feet, which coincidentally is the distance a controller must maintain to an aircraft, according to Navy standards. The specialized optics allows for a zoom of about 5.2X magnification. This is about one and a half times the standard optical zoom on most recreational hand held cameras. The sensor and optics are also mounted to a miniature stepping motor that allows the Omnibird to see 270 degrees.

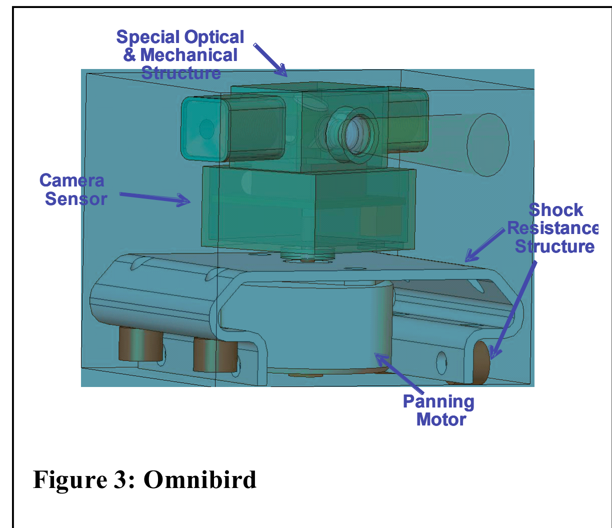

Figure 3: Omnibird

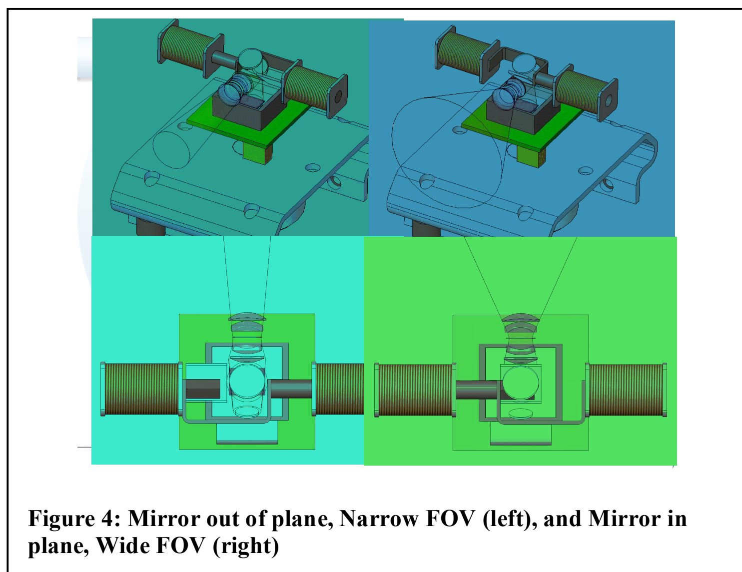

Figure 4: Mirror out of plane, Narrow FOV (left), and Mirror in plane, Wide FOV (right)

Unlike typical zoom optics where a lens is moved closer and farther to the viewing plane to achieve zooming capabilities the Omnibird uses a switching mirror and stationary optics. This allows the system to achieve a large zoom in a very small area. Figure 4, to the left, shows a preliminary Omnibird design a description of how the FOV switch works. As you can

see from the image the FOV switch allows for two FOV's using limited motion and allows for the compact design of the entire system.

SOS Processor

Also included as part of the Omnibird but outside of its packaging is another Genex product called the Smart Optical System (SOS). The SOS is an image processing board that allows for real time video to be processed on board. This will allow the images acquired from the Omnibird to be processed immediately so that only the relevant data is transmitted to the JUCAV Control System and the Omnibird.

Controller

The Navy personnel involved in deck taxiing are critical to the system. The Navy's Controllers are in charge of directing the JUCAV where to go on the flight deck. From the time the JUCAV lands to the time it launches for a mission the Controller is directly responsible for the vehicle.

The Navy has submitted a requirement that no deck handling procedures will change from the current procedures just because the plane has no pilot. This will allow the tried and true practices to continue into the future, and also save the Navy a great deal of time and money because they will not have to develop new procedures, retrain personnel, or add new personnel.

Figure 5: Flight Deck Controller (left), and Filtered image of Markers (right)

In order for the system to function day or night the Controllers must use a special marker system. The controller will have a marker on each hand, chest, and his head. The hand markers will distinguish the command given to the JUCAV, and the roughly fixed distance between the head and chest markers will allow the Omnibird to distinguish the Controller from the JUCAV. These markers will work in conjunction with the Omnibird by incorporating LEDs into their design that transmit light at roughly 940nm wavelength. This will allow for maximum absorption by the Omnibird and minimum interference with other naval equipment, namely night vision goggles which function at about 760nm.

High Level System Layout

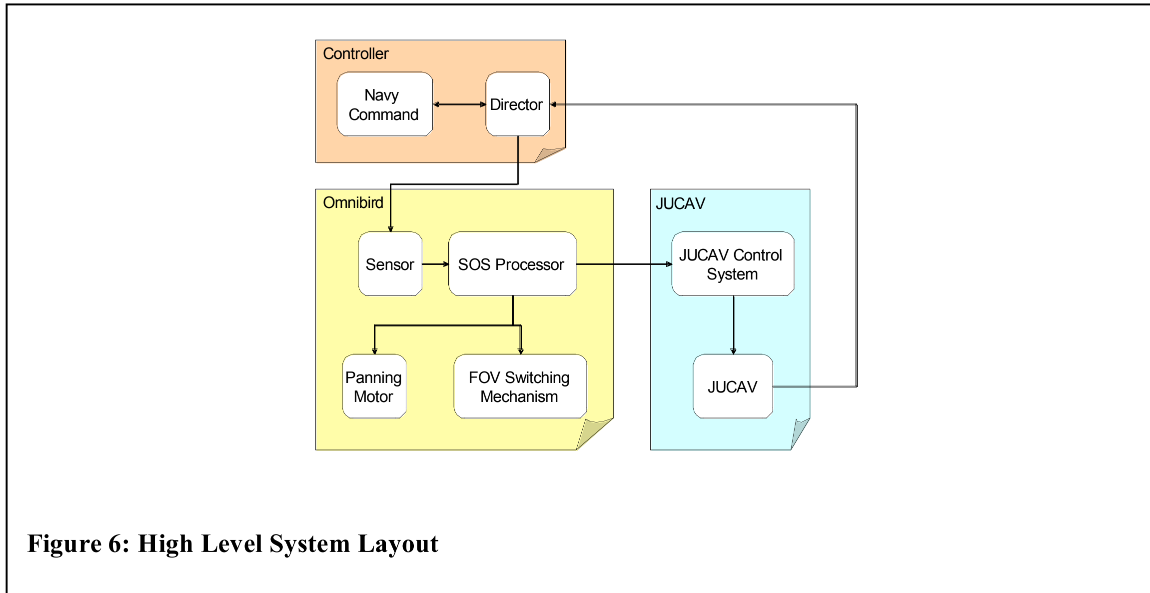

Figure 6: High Level System Layout

An aircraft carrier flight deck is one of the most dangerous and exciting places on earth. The Figure 6 shows that there are many persons and many planes and many obstacles that interfere with a Controller directing a plane. This project focuses on a simplified flight deck consisting of only one JUCAV, one controller, and no other obstacles. In reality this is a much more complicated system to design and safely operate.

The overall system in this project consists of three main sub-systems; the Controller, the Omnibird, and the JUCAV. Figure 6 shows a high level system layout. The three main sub-systems have individual components that will be the basis of the system designed in this project.

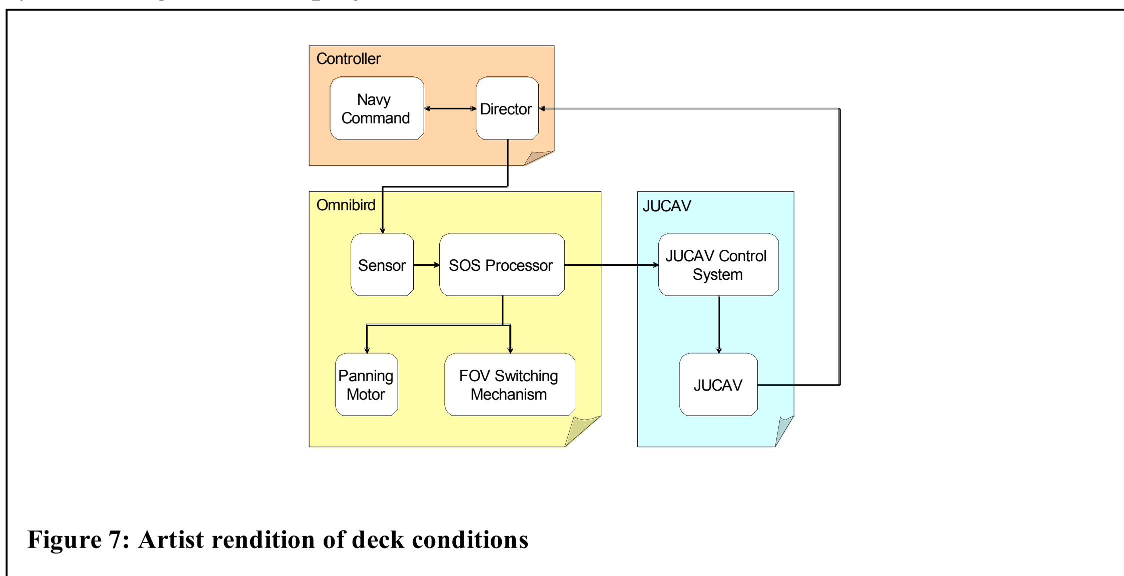

Figure 7: Artist rendition of deck conditions

Goals and Scenarios

The goals for the system were determined based upon the basic need of the system which is to have one controller safely control the JUCAV. The scenarios are based on inherent understanding of the system and the use cases below.

Goal #1: The Omnibird collects valid data.
Scenarios
1. Omnibird is fully functional
2. Omnibird is viewing the Controller for the entire time
3. Controller gives valid data

Goal #2: JUCAV only moves as intended
Scenarios
1. System processing data
2. Filtering algorithms eliminates invalid/unnecessary data
3. Filtering algorithms pass on valid commands
4. JUCAV alerts controller of malfunctions

Goal #3: Omnibird Functions at all locations on flight deck
Scenarios
1. Narrow FOV
2. Wide FOV

Goal #4: JUCAV only takes commands from one Controller
Scenarios
1. One controller on deck
2. Two controllers on deck

UML

The role of UML diagrams in this project is to describe the system. I have chosen to use ArgoUML as a modeling program. ArgoUML is an open source Java program that allows for the creation of and cross referencing of all UML diagrams in UML 2.0 format. This was especially useful in creating consistent UML diagrams. The ArgoUML source is provided in the reference section below.

Use Case Diagram

Figure 8: UML: Use Case Diagram

The Use Case Diagram shows the basic functions or use cases of the system. There are five main functions; Collect Data, Process Image, Generate Control Commands, Control Omnibird, and Control JUCAV. The actors are shown so that you can see that the “sub-actors” have a generalized relationship with the primary actors.

Activity Diagrams

Activity Diagrams are created by analyzing the flow of events that are encapsulated in each of the Use Cases from the Use Case Diagram. The below descriptions of the Activity Diagrams are generated directly from their specified Use Case.

Use Case #1: Collect Data

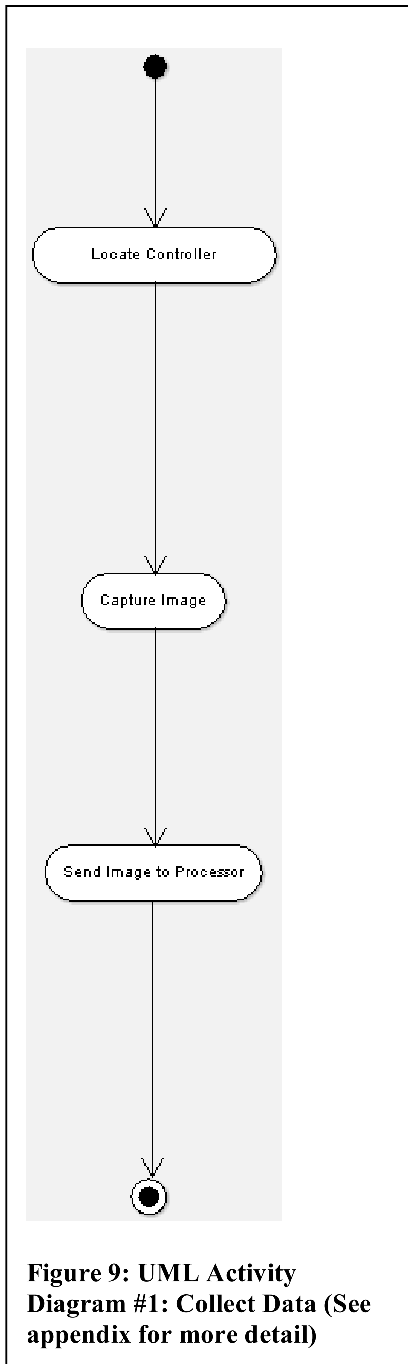

Figure 9: UML Activity Diagram #1: Collect Data (See appendix for more detail)

The Omnibird locates the Controller and captures images of Controller that are sent to the processor.

Primary Actor: Omnibird and Controller
Preconditions: JUCAV has landed, Controller is present, Omnibird is operational.
Post conditions: Data is sent to processor

Flow of Events:

1. Locate Controller
2. Capture Image
3. Send image to processor

Use Case #2: Process Image

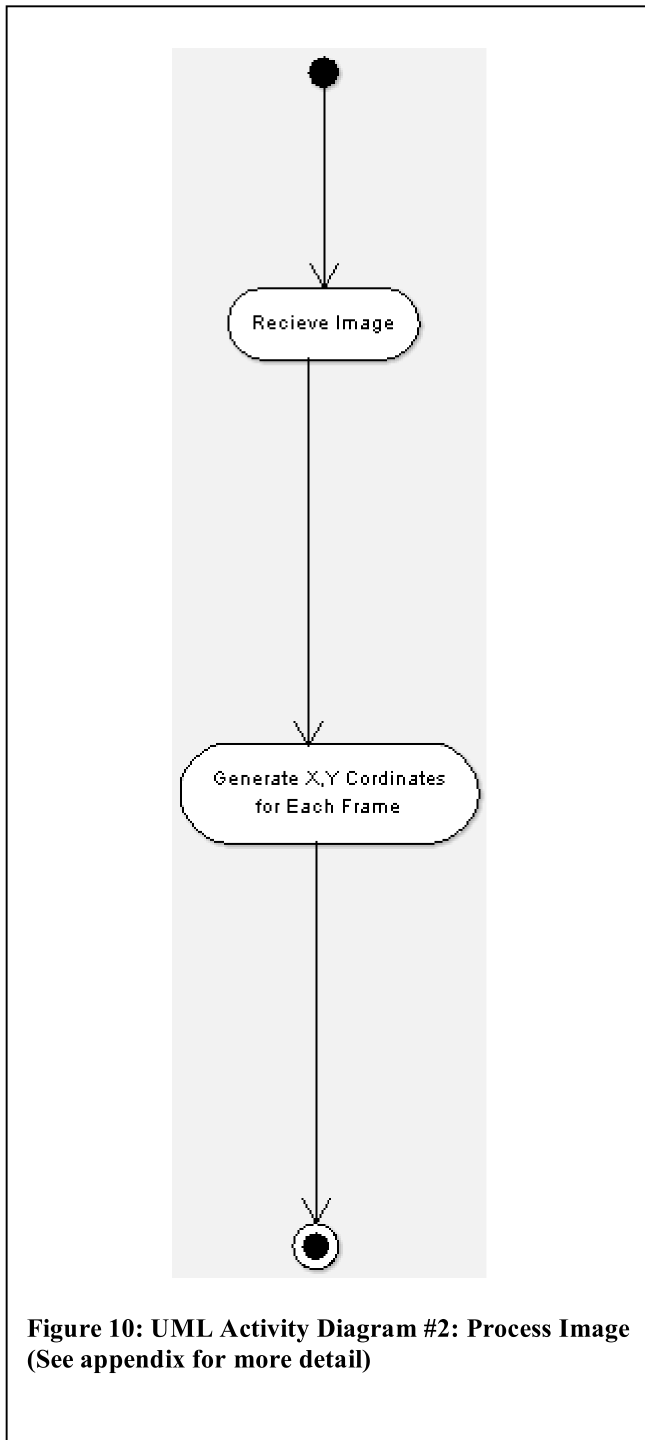

Figure 10: UML Activity Diagram #2: Process Image (See appendix for more detail)

SOS Processor receives the image frames from the sensor and extracts the XY coordinates for each of the four markers on the controller.

Primary Actor: Omnibird
Preconditions: Omnibird has transferred image frames from the sensor to the SOS processor.
Post conditions: Data is collected.

Flow of Events:

1. Receive image (Usable Image)
2. Generate XY coordinates for each marker
3. Data is sent on

Use Case #3: Generate Control Commands

The processed image data is post processed and controls are generated for the FOV switching mechanism, motor controller and JUCAV.

Figure 11: UML Activity Diagram #3: Generate Control Commands (See appendix for more detail)

JUCAV Deck Handling System

Primary Actor: Omnibird, JUCAV
Preconditions: XY coordinates have been processed from image and passed on.
Post conditions: Signal is passed on.

Flow of Events:
1. Receive XY Coordinates
2. Generate FOV control signal
3. Generate motor control signal
4. Pass signal on

Alternate Flow of Events #1:
1. Receive XY coordinates
2. Send XY Coordinates to JUCAV
3. Run tracking coordinates through command database
4. Generate plane control signal (Motion Identified)
5. Pass on signal

Use Case #4: Control Omnibird

The control commands generated by the SOS processor are sent to the FOV switching mechanism and the motor controller. The FOV switch and motor are directed into proper positions and the new positions are updated.

Figure 12: UML Activity Diagram #4: Control Omnibird (See appendix for more detail)

Primary Actor: Omnibird
Preconditions: Control signals are generated and passed on appropriately.
Post conditions: Updated conditions ore recorded

Flow of Events:

1. Receive Control Commands From SOS Processor
2. Direct Motor into Position (if new position)
3. Send new position data to SOS Processor

Alternative Flow of Events #1:

1. Receive Control Commands From SOS Processor
2. Select Appropriate FOV (if new position)
3. Send new position data to SOS Processor

Use Case #5: Control JUCAV

The control signal directs the engine/brakes to carry out the Controllers intended motion

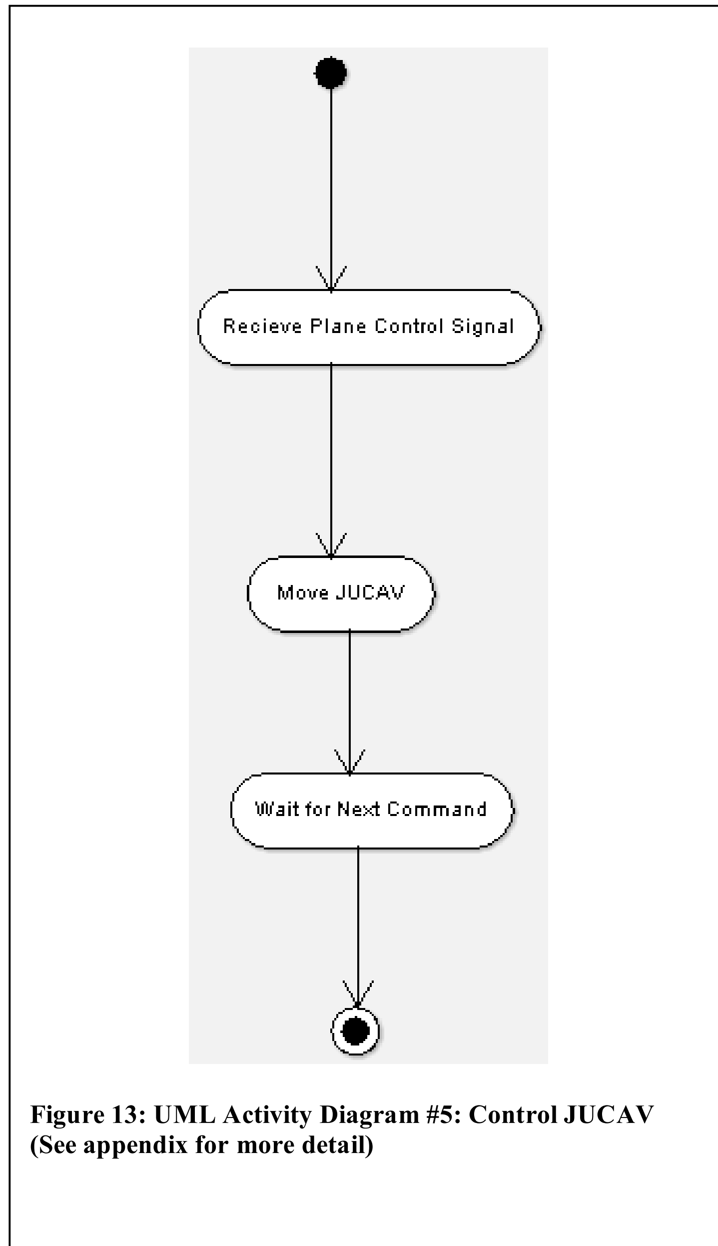

Figure 13: UML Activity Diagram #5: Control JUCAV (See appendix for more detail)

Primary Actor: JUCAV
Preconditions: A motion is identified and a control signal is generated and passed onto the JUCAV.
Post conditions: JUCAV is awaiting next command from Controller.

Flow of Events:

1. Receive plane control signal
2. Move JUCAV
3. Wait for next command

Class Diagram

Figure 14: UML: Class Diagram (See appendix for more detail)

The class diagram above shows the interdependent relationships between the classes and sub classes of the system. As you can see from the diagram the Omnibird, Controller, and JUCAV have their inner classes packaged in an outer envelope as shown. This allows the system to be very organized but also allows you to see how the inner classes interact amongst each other inside the package envelope, and out.

Requirements

1 User Requirements
1.1 Controller shall understand when he has control of JUCAV.
1.2 Controller must understand flight deck procedures.
1.3 Controller does not pass false instructions to JUCAV.
1.4 Controller must be able to direct JUCAV into position
1.5 Controller must quickly control JUCAV upon landing.
2 Performance Requirements
2.1 Sensor is calibrated appropriately.
2.2 Omnibird shall be connected to JUCAV.
2.3 Sensor shall be connected to SOS processor.
2.4 Image processing accurately determines XY coordinates of markers.
2.5 Data processed must be reliable.
2.6 Data must be able to be obtained when the Controller is at both ends of the flight deck.
2.7 Data must be obtained while JUCAV is in motion.
2.8 Omnibird recognizes all instructions from Controller.
2.9 Omnibird must be able to keep Controller inside of its FOV.
3 Functional Requirements
3.1 Omnibird must meet requirements of the JUCAV.
3.2 Omnibird must not need regular maintenance.
3.3 Controller must meet requirements of the Flight Deck.
3.4 Omnibird must be able to function on the Flight Deck.
3.5 Omnibird must alert Controller of a malfunction.
3.6 Omnibird does not give false directions to JUCAV.
3.7 Omnibird must function 24/7.
4 Safety Requirements
4.1 JUCAV must remain on Flight Deck while being directed by controller.
4.2 Controller must remain on Flight Deck.
4.3 JUCAV must not move with out a Controller directing it.
4.4 Omnibird must be able to locate a Controller.
4.5 Omnibird must only take directions from one Controller.
4.6 JUCAV shall not be in motion if there is no data being collected.
4.7 JUCAV can alert Controller of Omnibird malfunction.
5 Test Requirements
5.1 Omnibird tested to meet conditions on Flight Deck.
5.2 Omnibird tested to meet user alert requirements.
5.3 Tested to determine what data is observed from Controller.
5.4 Tested to determine what data is passed to JUCAV.
5.5 Test to verify image processing algorithm.
5.6 Test to verify complete database of control commands.

Traceability Matrix

Figure #15:

#	1.1	1.2	1.3	2.1	2.2	2.3	2.4	3.1	3.2	4.1	4.2	Scenario
1												
1.1							■			■		4.1,2.4
1.2			■									1.3
1.3			■				■					1.3
1.4	■	■	■	■	■	■	■	■	■	■	■	all
1.5		■										1.2
2												
2.1	■											1.1
2.2	■					■						1.1,2.3
2.3	■											1.1
2.4					■	■						2.2,2.3
2.5					■	■	■					2.2,2.3,2.4
2.6								■	■			3.1,3.2
2.7	■											1.1
2.8		■				■						1.2,2.3
2.9		■										1.2
3												
3.1	■											1.1
3.2	■											1.1
3.3			■									1.3
3.4	■											1.1
3.5							■					2.4
3.6			■									1.3
3.7	■											1.1
4												
4.1		■										1.2
4.2		■										1.2
4.3								■				3.1
4.4			■									1.3
4.5								■				3.1
4.6					■							2.2
4.7							■					2.4
5												
5.1	■											1.1
5.2	■						■					1.1,2.4
5.3					■	■						2.2,2.3
5.4					■	■						2.2,2.3
5.5				■								2.1
5.6						■						2.3

LTSA

According to the LTSA developer's webpage:

"LTSA is a verification tool for concurrent systems. It mechanically checks that the specification of a concurrent system satisfies the properties required of its behavior. In addition, LTSA supports specification animation to facilitate interactive exploration of system behavior.

A system in LTSA is modeled as a set of interacting finite state machines. The properties required of the system are also modeled as state machines. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties. More formally, each component of a specification is described as a Labeled Transition System (LTS), which contains all the states a component may reach and all the transitions it may perform. However, explicit description of an LTS in terms of its states, set of action labels and transition relation is cumbersome for other than small systems [3]."

This project uses an LTSA plug-in that enables the use of Message Sequence Charts (MSC) to be used to graphically represent the JUCAV Deck Handling System as a set of interlocking message sequence charts. A MSC is very similar to a sequence diagram in UML. By using the interlocking MSCs we are able to describe the sequence the system sends messages so that we are able to verify that the system functions as intended.

Verification Plan

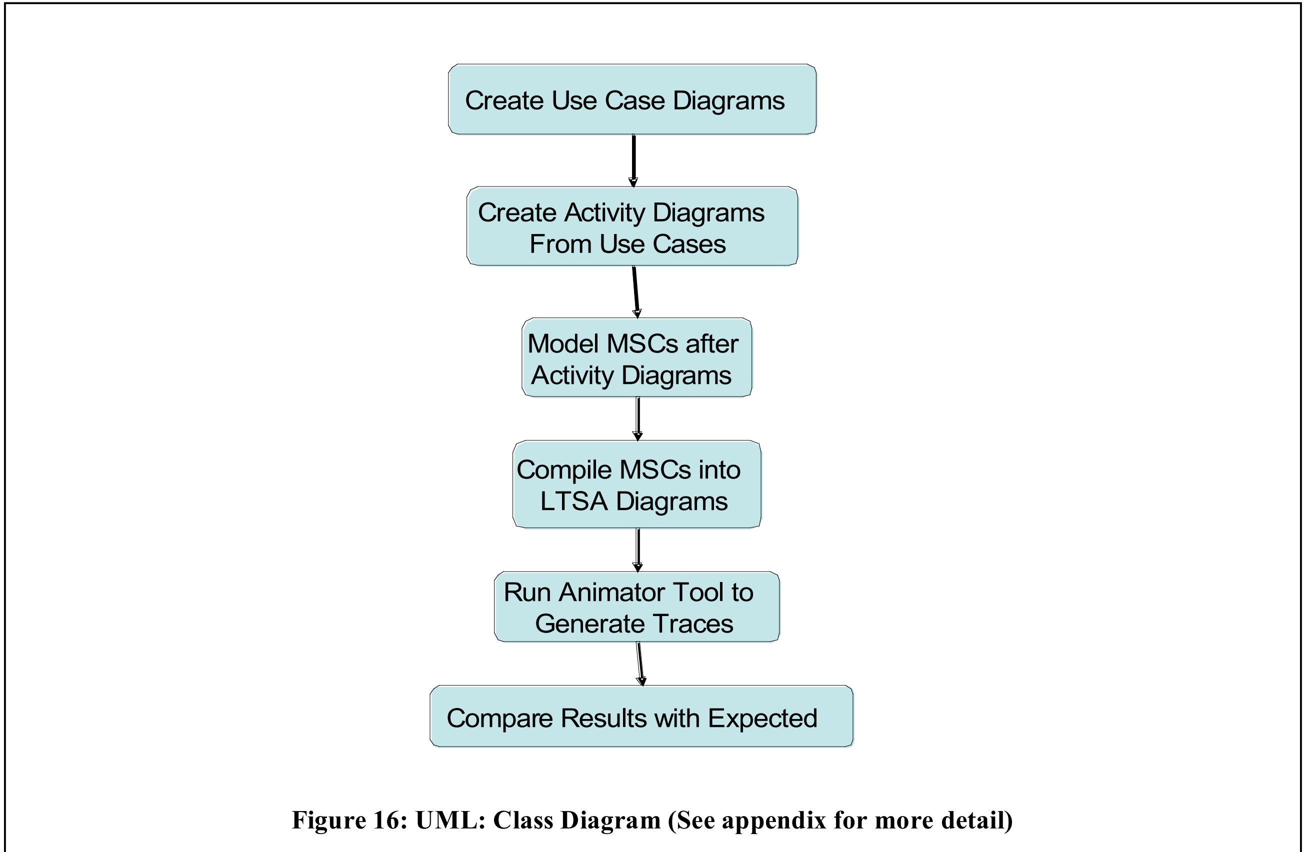

Figure 16: UML: Class Diagram (See appendix for more detail)

Figure 15 shows the flow of events to verify that the established design is a valid design. The basic sequence is to start by creating Use Cases, and then create Activity Diagrams from these Use Cases. You can them model the Activity Diagrams into the MSC plug-in. By using LTSA you can establish a trace of the messages being passed. This trace should match exactly what was expected from your Activity Diagrams.

The weak links in this process are the conversion from the Activity Diagrams to the MSC, and the human verification of symmetry between the Traces and the Activity Diagrams. A future project could be the development of a plug-in that automatically verifies the Traces.

LTSA: MSC High Level

init

PositionO...

ProcessIm...

ValidComm...

InvalidCo...

Figure 17: High Level Message Sequence Chart

Figure 17 shows the High Level MSC. This is basically a flow chart for the entire system. In fact it looks exactly like a basic high level flow chart. The arrows represent the sequence of progression through the design. These boxes represent each of the Lower Level MSC diagrams. The output of the Lower Level MSC must be the input of the next Lower Level MSC based upon the arrows in the Higher Level MSC.

LTSA: LTS Diagrams

The LTS diagrams below show the individual actor's message passing options. These diagrams will be better viewed in the LTSA program with the program attached to the report.

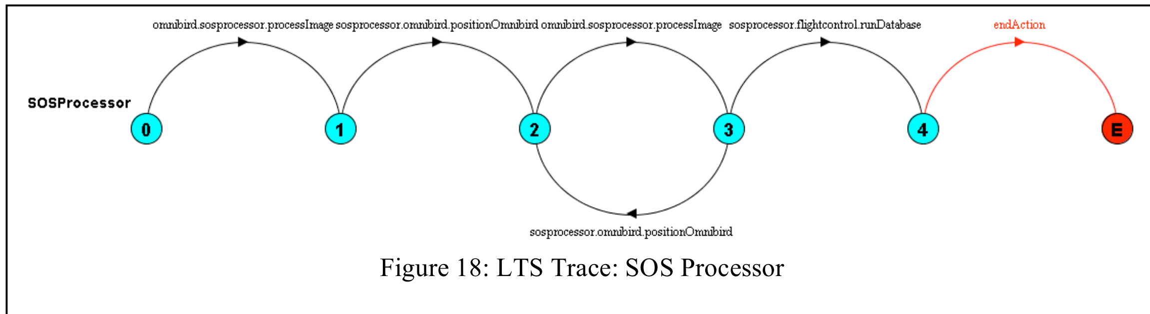

Figure 18: LTS Trace: SOS Processor

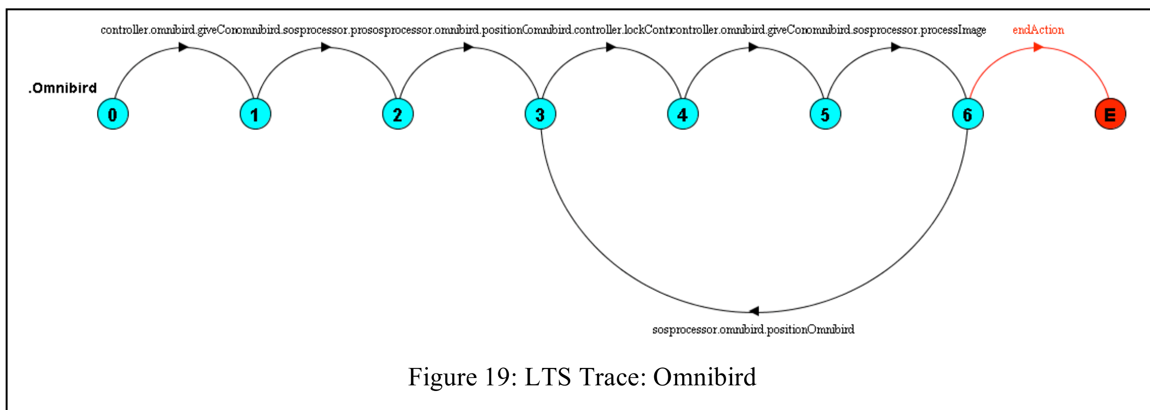

Figure 19: LTS Trace: Omnibird

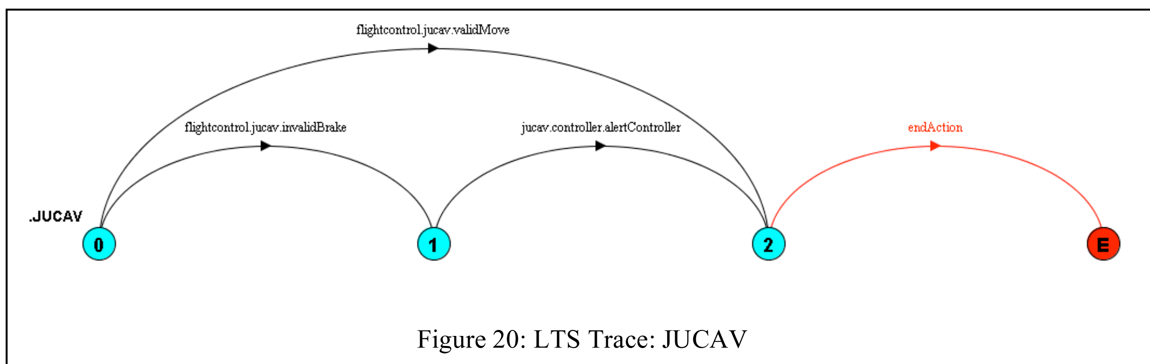

Figure 20: LTS Trace: JUCAV

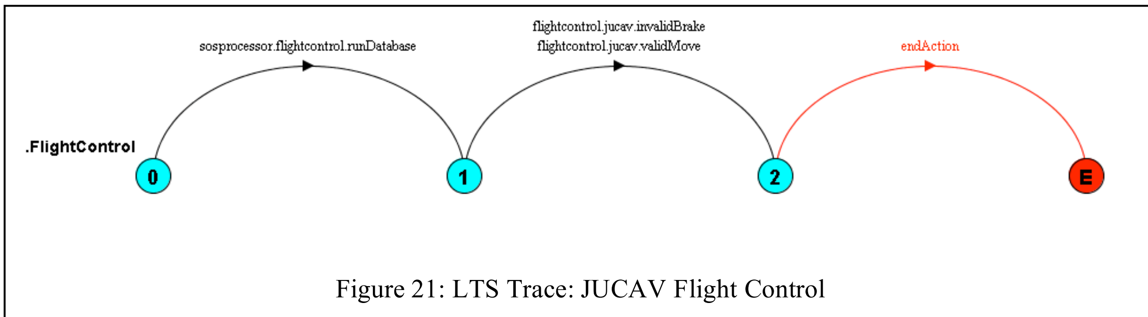

Figure 21: LTS Trace: JUCAV Flight Control

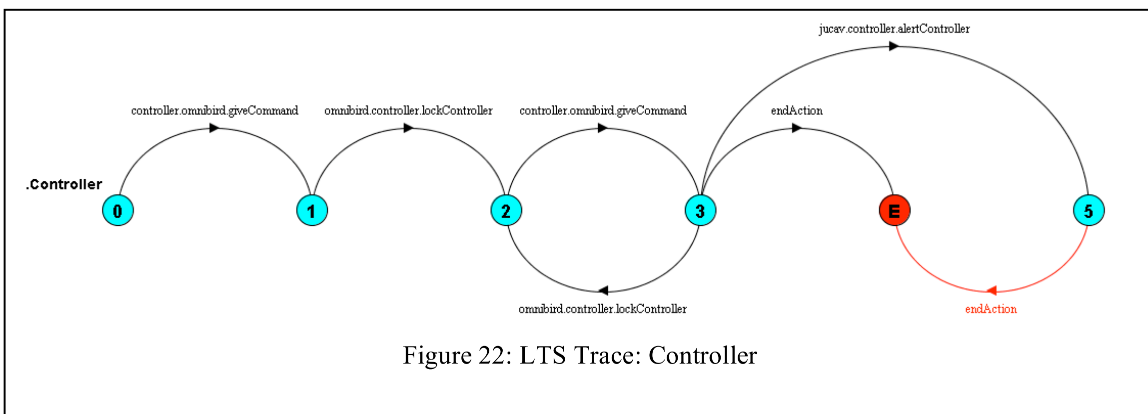

Figure 22: LTS Trace: Controller

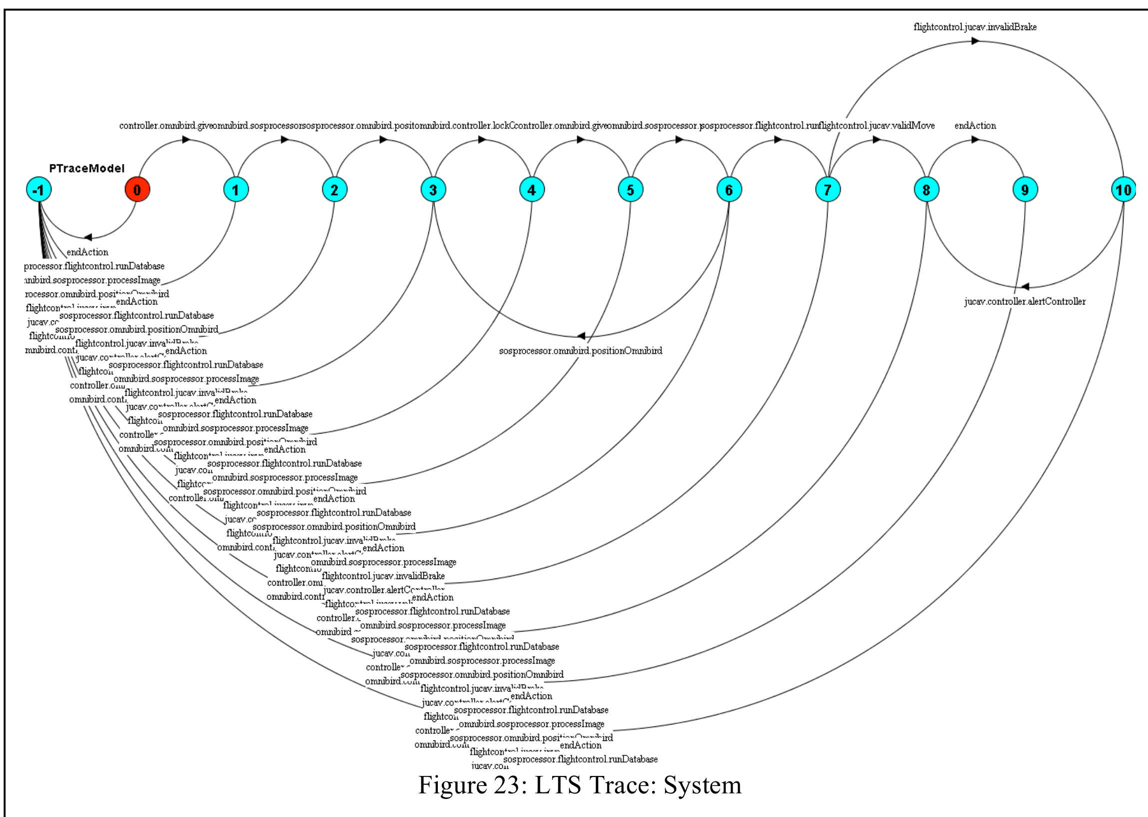

Figure 23: LTS Trace: System

LTSA: MSC Traces

The below traces show the possible ways that the messages can be passed. Figure @@ shows the trace of a valid command. These traces allow us to follow the progress of the system and verify that the messages can make it through to the end of the JUCAV actions.

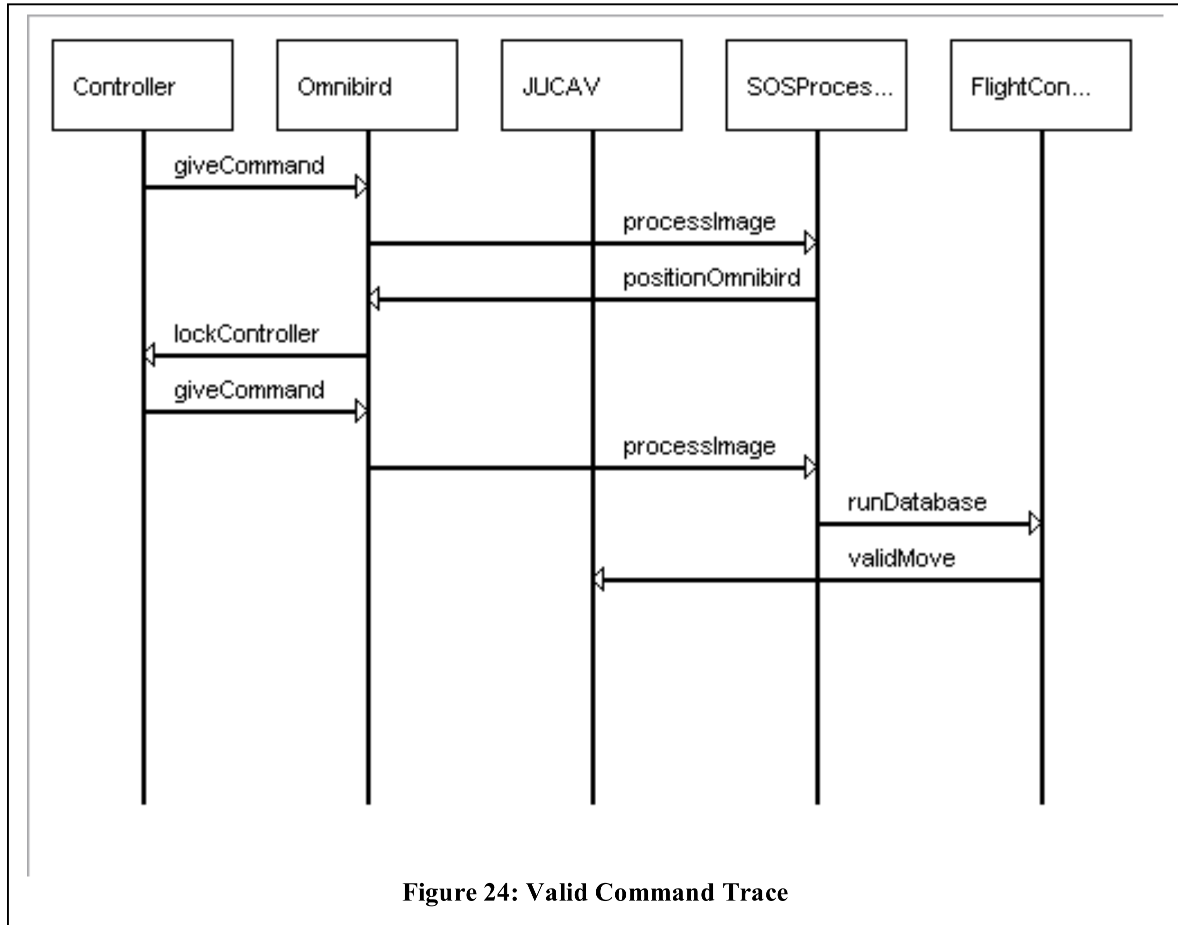

Figure 24: Valid Command Trace

Figure @@, below, shows the message trace for an invalid command. As you can see the JUCAV is automatically commanded to apply the brakes for an invalid command.

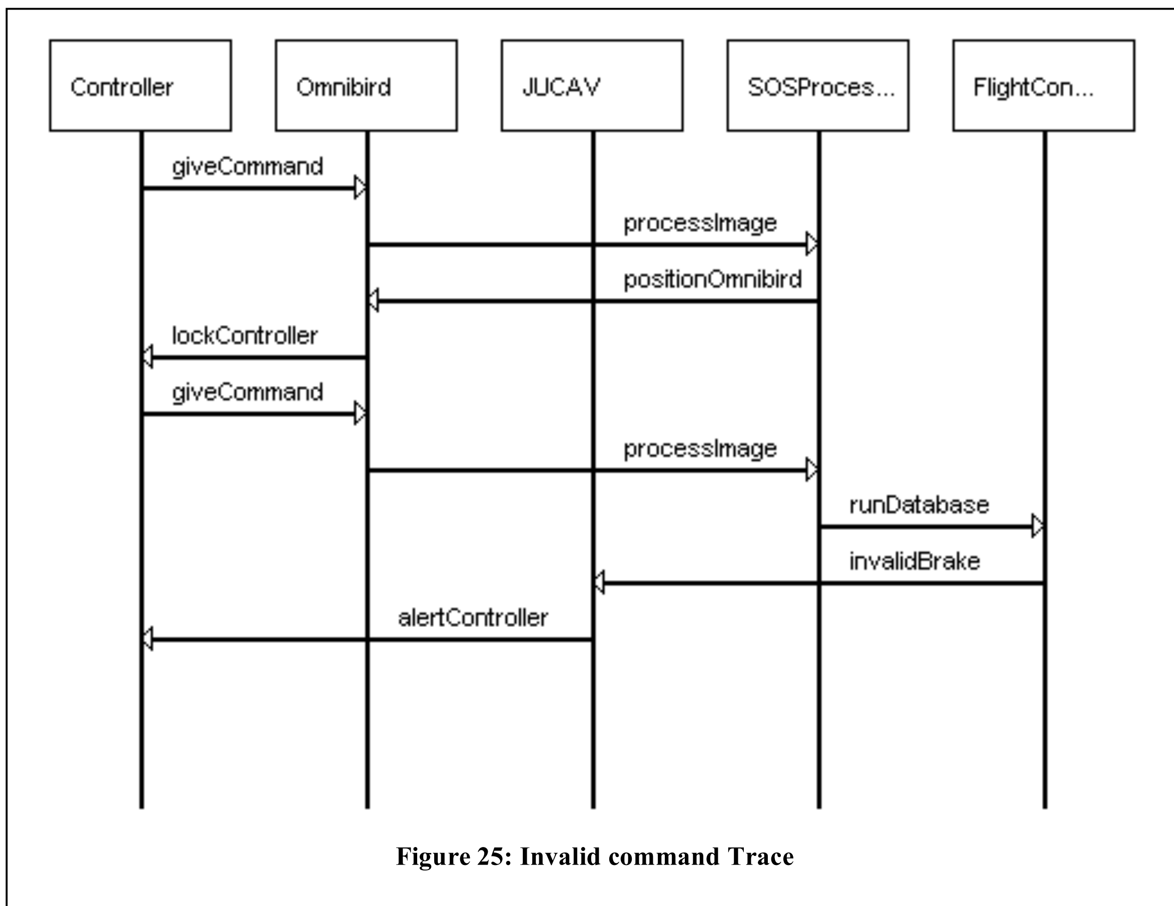

Figure 25: Invalid command Trace

LTSA: Validation and Verification

LTSA also offers several things that allow us to verify the system. Two of these features are the Progress Check and the Safety Check. The progress check maintains that there are no deadlocks in the system. The LTSA print out of the Safety Check is shown in Figure @@, below. It shows that there are no safety violations in the system.

```
Composition:
ImpliedScenarioCheck = PTraceModel ||
ConstrainedArchitectureModel.ArchitectureModel.SOSProcessor ||
ConstrainedArchitectureModel.ArchitectureModel.JUCAV ||
ConstrainedArchitectureModel.ArchitectureModel.Omnibird ||
ConstrainedArchitectureModel.ArchitectureModel.FlightControl ||
ConstrainedArchitectureModel.ArchitectureModel.Controller ||
ConstrainedArchitectureModel.ConstraintModel
State Space:
 11 * 6 * 4 * 8 * 4 * 6 * 1 = 2 ** 17
Analysing...
Depth 10 -- States: 13 Transitions: 14 Memory used: 14245K
No deadlocks/errors
Analysed in: 0ms
```

Figure 25: Safety Check

The Progress Check, on the other hand, reports several violations. However these violations are what is expected because they occur at the end of the system trace. When the JUCAV moves or brakes for an invalid command it is automatically given the next signal from the controller. The process of generating new commands is a repetitive process.

Conclusions

Several conclusions can be drawn from this report. The first conclusion that is not readily apparent from the report is the inherent limitations of LTSA. The MSC editor plug-in does not allow many actors to be used. When experimenting with the program it became quickly apparent that once more than five actors were included in the bMSC diagrams the program would not compile the diagrams appropriately or at all.

This limitation is a major cause of concern when modeling a system with many actors and sub-actors. I was able to experiment with using limited actors and many messages; the program functioned with little problem. This leads me to believe that the best way to model these complex types of systems is to model sub systems and then treat each sub system as its own bMSC when compiling a high level MSC for the overall system. This will allow for better modeling of larger systems. This will be a problem if the messages between the subsystems are dependant on the messages within the subsystems themselves. One way around this problem is a better tool that allows for a more robust analysis of finite state machines. A program that makes better use of available memory will better suit this application.

The trace diagrams for the MSC results do match the activity diagrams generated for the specified segments of the system. These traces are a bit hard to distinguish because the activity diagrams are basically these traces in piecemeal form. Also I find that a tool is needed to automate the conversion between activity diagrams and the MSC diagrams would be especially useful to verify that the MSC and activity diagrams are exactly alike. Basically the user could easily manipulate the way the MSC diagrams are compiled compared to the activity diagrams and achieve any result that they want. A completely automated program would be much better at accomplishing this. Perhaps an XML exporting of code from ArgoUML could be the input for the MSC plug in and thus eliminating the chance of errors being inserted accidentally.

References

1. Class Notes 2006
2. http://argouml.tigris.org/
3. http://www.doc.ic.ac.uk/ltsa/
4. http://www.doc.ic.ac.uk/ltsa/msc/
5. http://www.doc.ic.ac.uk/ltsa/msc/tutorial/
6. Jeff Magee and Jeff Kramer, Concurrency: State Models and Java Programs (2nd Edition), John Wiley and Sons, 2006.
7. Class Projects 2004, 2006
    a. Validation, Verification, and Behavior Modeling of a Canal System by Noosha Haghani and Nazanin Alborni
    b. Validating Behavior of an Airport Control Tower with LTSA by Kerin Thornton
    c. Prognostics System on a Military Wheeled Vehicle by Craig Hershey

Appendix

Final LTSA

XML Code


```
  <?xml version="1.0" encoding="UTF-8" ?>
- <specification>
- <hmsc>
  <bmsc name="init" x="50" y="30" />
  <bmsc name="PositionOmnibird" x="170" y="130" />
  <bmsc name="ProcessImage" x="170" y="230" />
  <bmsc name="ValidCommand" x="290" y="230" />
  <bmsc name="InvalidCommand" x="170" y="330" />
- <transition>
  <from>init</from>
  <to>PositionOmnibird</to>
    </transition>
- <transition>
  <from>PositionOmnibird</from>
  <to>PositionOmnibird</to>
    </transition>
- <transition>
  <from>ProcessImage</from>
  <to>ValidCommand</to>
    </transition>
- <transition>
  <from>ProcessImage</from>
  <to>InvalidCommand</to>
    </transition>
- <transition>
  <from>PositionOmnibird</from>
  <to>ProcessImage</to>
    </transition>
    </hmsc>
  <bmsc name="init" />
- <bmsc name="PositionOmnibird">
- <instance name="Controller">
- <input timeindex="6">
  <name>omnibird,controller,lockController</name>
  <from>Omnibird</from>
    </input>
- <output timeindex="2">
  <name>controller,omnibird,giveCommand</name>
```

```
<to>Omnibird</to>
  </output>
  </instance>
- <instance name="Omnibird">
- <input timeindex="5">
<name>sosprocessor,omnibird,positionOmnibird</name>
<from>SOSProcessor</from>
  </input>
- <output timeindex="6">
<name>omnibird,controller,lockController</name>
<to>Controller</to>
  </output>
- <output timeindex="3">
<name>omnibird,sosprocessor,processImage</name>
<to>SOSProcessor</to>
  </output>
- <input timeindex="2">
<name>controller,omnibird,giveCommand</name>
<from>Controller</from>
  </input>
  </instance>
<instance name="JUCAV" />
- <instance name="SOSProcessor">
- <output timeindex="5">
<name>sosprocessor,omnibird,positionOmnibird</name>
<to>Omnibird</to>
  </output>
- <input timeindex="3">
<name>omnibird,sosprocessor,processImage</name>
<from>Omnibird</from>
  </input>
  </instance>
<instance name="FlightControl" />
  </bmsc>
- <bmsc name="ProcessImage">
- <instance name="Controller">
- <output timeindex="1">
<name>controller,omnibird,giveCommand</name>
<to>Omnibird</to>
  </output>
  </instance>
- <instance name="Omnibird">
- <input timeindex="1">
<name>controller,omnibird,giveCommand</name>
<from>Controller</from>
```

```
    </input>
- <output timeindex="2">
  <name>omnibird,sosprocessor,processImage</name>
  <to>SOSProcessor</to>
    </output>
    </instance>
  <instance name="JUCAV" />
- <instance name="SOSProcessor">
- <input timeindex="2">
  <name>omnibird,sosprocessor,processImage</name>
  <from>Omnibird</from>
    </input>
    </instance>
  <instance name="FlightControl" />
    </bmsc>
- <bmsc name="ValidCommand">
  <instance name="Controller" />
  <instance name="Omnibird" />
- <instance name="JUCAV">
- <input timeindex="3">
  <name>flightcontrol,jucav,validMove</name>
  <from>FlightControl</from>
    </input>
    </instance>
- <instance name="SOSProcessor">
- <output timeindex="2">
  <name>sosprocessor,flightcontrol,runDatabase</name>
  <to>FlightControl</to>
    </output>
    </instance>
- <instance name="FlightControl">
- <input timeindex="2">
  <name>sosprocessor,flightcontrol,runDatabase</name>
  <from>SOSProcessor</from>
    </input>
- <output timeindex="3">
  <name>flightcontrol,jucav,validMove</name>
  <to>JUCAV</to>
    </output>
    </instance>
    </bmsc>
- <bmsc name="InvalidCommand">
- <instance name="Controller">
- <input timeindex="3">
  <name>jucav,controller,alertController</name>
```

```
  <from>JUCAV</from>
    </input>
    </instance>
  <instance name="Omnibird" />
- <instance name="JUCAV">
- <input timeindex="2">
  <name>flightcontrol,jucav,invalidBrake</name>
  <from>FlightControl</from>
    </input>
- <output timeindex="3">
  <name>jucav,controller,alertController</name>
  <to>Controller</to>
    </output>
    </instance>
- <instance name="SOSProcessor">
- <output timeindex="1">
  <name>sosprocessor,flightcontrol,runDatabase</name>
  <to>FlightControl</to>
    </output>
    </instance>
- <instance name="FlightControl">
- <input timeindex="1">
  <name>sosprocessor,flightcontrol,runDatabase</name>
  <from>SOSProcessor</from>
    </input>
- <output timeindex="2">
  <name>flightcontrol,jucav,invalidBrake</name>
  <to>JUCAV</to>
    </output>
    </instance>
    </bmsc>
- <bmsc name="Trace">
- <instance name="Controller">
- <output timeindex="1">
  <name>controller,omnibird,giveCommand</name>
  <to>Omnibird</to>
    </output>
- <input timeindex="4">
  <name>omnibird,controller,lockController</name>
  <from>Omnibird</from>
    </input>
- <output timeindex="5">
  <name>controller,omnibird,giveCommand</name>
  <to>Omnibird</to>
    </output>
```

```
- <input timeindex="9">
  <name>jucav,controller,alertController</name>
  <from>JUCAV</from>
    </input>
    </instance>
- <instance name="Omnibird">
- <input timeindex="1">
  <name>controller,omnibird,giveCommand</name>
  <from>Controller</from>
    </input>
- <output timeindex="2">
  <name>omnibird,sosprocessor,processImage</name>
  <to>SOSProcessor</to>
    </output>
- <input timeindex="3">
  <name>sosprocessor,omnibird,positionOmnibird</name>
  <from>SOSProcessor</from>
    </input>
- <output timeindex="4">
  <name>omnibird,controller,lockController</name>
  <to>Controller</to>
    </output>
- <input timeindex="5">
  <name>controller,omnibird,giveCommand</name>
  <from>Controller</from>
    </input>
- <output timeindex="6">
  <name>omnibird,sosprocessor,processImage</name>
  <to>SOSProcessor</to>
    </output>
    </instance>
- <instance name="JUCAV">
- <input timeindex="8">
  <name>flightcontrol,jucav,invalidBrake</name>
  <from>FlightControl</from>
    </input>
- <output timeindex="9">
  <name>jucav,controller,alertController</name>
  <to>Controller</to>
    </output>
    </instance>
- <instance name="SOSProcessor">
- <input timeindex="2">
  <name>omnibird,sosprocessor,processImage</name>
  <from>Omnibird</from>
```

```
   </input>
- <output timeindex="3">
  <name>sosprocessor,omnibird,positionOmnibird</name>
  <to>Omnibird</to>
   </output>
- <input timeindex="6">
  <name>omnibird,sosprocessor,processImage</name>
  <from>Omnibird</from>
   </input>
- <output timeindex="7">
  <name>sosprocessor,flightcontrol,runDatabase</name>
  <to>FlightControl</to>
   </output>
   </instance>
- <instance name="FlightControl">
- <input timeindex="7">
  <name>sosprocessor,flightcontrol,runDatabase</name>
  <from>SOSProcessor</from>
   </input>
- <output timeindex="8">
  <name>flightcontrol,jucav,invalidBrake</name>
  <to>JUCAV</to>
   </output>
   </instance>
   </bmsc>
   </specification>
```