Abstract— The paper describes a hierarchical architecture in which mission level controllers based on hybrid systems theory have been, and are being developed using a hybrid systems design tool that allows graphical design, iterative redesign, and code generation for rapid deployment onto the target platform. The goal is to support current and future autonomous underwater vehicle (AUV) programs to meet evolving requirements and capabilities. While the tool facilitates rapid redesign and deployment, it is beneficial to animate/simulate the execution of missions. To this end a basic converter has been developed to convert the mission controller modules to OpenGL code which animates the execution of missions.

I. INTRODUCTION

We have developed a mission control architecture for autonomous underwater vehicles (AUVs) that facilitates modeling, and verification of logical correctness of the mission controller/AUV. Here we propose a method for simulation/animation of the missions executed. The mission controller has been under development at the Applied Research Laboratory at the Pennsylvania State University. The design of the architecture in which the mission controller is being developed has benefited from the discussions with the collaborators from Iowa State University and University of Kentucky.

An autonomous system is an unmanned system which in our case is an underwater vehicle. The control tasks for an underwater vehicle or for an autonomous system can be divided into lower level control, concerned with continuous dynamics and a higher-level mission coordinator/coordinators, which is discrete, either event-driven, or time-driven. The mission coordinators contain both sequence coordinator and timed coordinator for sequential execution and timed execution of various operations of the mission. Thus the overall system is a hybrid system containing both continuous and discrete states.

The mission controller at the higher level is hierarchically organized into three layers. The basic idea is to hierarchically decompose missions into sequence of operations, and operations into sequence of behaviors, and behaviors into sequence of vehicle maneuvers. Then we need to design a behavior-controller for each behavior that does appropriate coordination of appropriate vehicle-manuever controllers, an operation-controller for each operation that does appropriate coordination of appropriate behavior controllers, and a coordinator for each mission specification (untimed, timed, and safety) that does appropriate coordination of appropriate operations controllers. Each of the controllers at each of the layers is modeled as a hybrid system.

In an attempt to manage the complexity of design, we formulate a hierarchical control architecture upon which the mission controller design is based. This architecture not only facilitates the design of a complex mission controller, it also facilitates the verification and potentially, the automated synthesis of the highest level mission coordinator(s).

Animation/Simulation is the imitation of the reality for studying the effect of changing parameters in a model as a means of taking a decision. Animation/Simulation imitates or estimates how events might occur in a real situation. It can involve complex mathematical modeling, role playing without the aid of technology, or combinations. The value lies in placing one under realistic condition, that change as a result of behavior of other variables involved so one cannot anticipate the sequence of events or the final outcome. An animation/simulation tool aids in the creation of the realistic environment using mathematical formulas and algorithms. Usage of a simulation tool gives one the advantage of checking the accuracy of algorithms involved in the experiment without incurring damage to the vehicle or
equipments involved. Due to which we looked into the feasibility of creating an animation/simulation tool for a mission driven AUV which we discuss in this paper.

A simulation tool had been implemented for the automated highway system in the PATH project at Berkley [1][2]. The use of such a simulation tool proved to be advantageous because of the beneficial outcomes. It helped to prove the correctness of the mathematical modeling before actual implementation. A simulation in addition to verification further strengthens the correctness of a modeled system. This is because mostly verification of a hybrid system is carried out by abstracting the system and it might miss out on faulty situation which might occur for the combined system. Animation/Simulation involves both the continuous and discrete dynamics combined together to successfully execute a mission. So a simulation would catch some interactions which might be missed while carrying out verification as described by Godbole, Lygeros and Sastry in [3].

We looked into the feasibility of creating a simulation environment to model the actions executed by the AUV under given mission orders. The preliminary tool developed is a very basic tool which proves the possibility of having a very advanced simulation tool in future. The simulation tool is very specific to the survey missions for an AUV. OpenGL is used to simulate/animate the missions executed by the AUV. The hierarchical model based structure reduced the complexity involved in the conversion of the controller modules to animation modules.

The mission controller modules are developed using TEJA NP networking software tool [25]. TEJA supports the design of interacting hybrid state machines and includes automatic real-time code generation which allows for rapid deployment on the target platform. For verification purposes, the Teja modules specifications are first represented formally and then transformed into a format readable by UPPAAL [26], a hybrid system modeling, simulation, and verification tool. The mission controller modules in UPPAAL are then converted to animation modules. Section II outlines the hybrid systems model framework that is used to formalize the mission controller modules, section III describes our hybrid mission control architecture, section IV describes the implementation of the modules, section V describes the tool used for animation, section VI describes the procedure for animation/simulation of the missions, section VII shows abstracted code for a module and section VIII concludes the accomplishments.

II. HYBRID MISSION CONTROLLER ARCHITECTURE

Our mission control architecture is designed with semi-automatic (safety and progress) verification in mind. All the levels and modules that make up the hierarchy conform to the interacting controlled hybrid systems model described in Section II; and the tool used to implement the hierarchy allows the conversion of the representation of the hybrid automata into a format that is readable by available verification tools such as HyTech [31] and Uppaal [32]. Interactions between modules are restricted to event synchronizations and shared data. The logic within individual automata is restricted to use clocks as the only continuous variables and all the continuous dynamics are encapsulated in functions. The controller modules in the Uppaal format are then converted into openGL code.

The hybrid mission controller is organized hierarchically as shown in below. Modules within a level may communicate with each other and each level in the hierarchy is restricted to command the level immediately below it and send responses to the level immediately above it. All levels in the mission controller hierarchy may assign vehicle commands directly by placing an appropriate vehicle commands in the shared database. At the lowest level of the hierarchy the vehicle controllers have a hybrid state-space (which might, in some vehicles, be a purely continuous state space), and serve as the plant for the higher level mission controller (MC), which is also hybrid in nature.

The vehicle controller and the mission controller communicate through an interface layer symbolically represented by MC2VC (mission controller to vehicle controller) and VC2MC (vehicle controller to mission controller). The MC2VC block also includes a Command Conflict Manager which is responsible for selecting a specific vehicle level command (when more than one exists) according to a static or dynamic priority list or using other methods (such as optimization). This module is included since all modules in the mission controller hierarchy are allowed to assign vehicle commands directly, and so there is a distinct possibility that multiple vehicle commands can coexist.
**Behavior Controllers**, where a *behavior* may be thought of as a skill or ability that an autonomous system possesses which enables it to perform specific mission tasks (*thrive*) while remaining safe (*survive*). Behaviors directly interface with the vehicle controllers and are therefore vehicle-centric. They require executions of sequences of vehicle maneuvers. The middle level of the mission control hierarchy consists of *Operation Controllers*, where an *operation* represents a mission segment or phase that is integral to the completion of the overall AUV mission, and are user/mission-centric. They are directly commanded by the user via mission orders and, in turn, command/sequence the behavior controllers to achieve their objectives. The highest level of the mission controller consists of the *Mission Coordinators* which are responsible for sequencing and scheduling operations in order to complete the mission while ensuring the safety of the vehicle. Mission coordinators are typically of two types, safety and progress. Progress coordinator may be separated into untimed and timed.

A mission is defined as a coordinated sequence of operations, each of which is a sequence of behaviors, and possibly vehicle controller commands. Each behavior is, in turn, a sequence of commands to the vehicle subsystem controllers via the MC2VC interface. AUV state information is collected by the sensors and transferred by the VC2MC interface periodically to the shared database. This state information is made available to all modules in all levels of the mission controller hierarchy. Similarly, vehicle commands, assigned and manipulated by all levels in the mission controller are stored in the shared database and sent to the AUV by the MC2VC interface.

Command events propagate down the mission controller hierarchy and response events propagate up the mission controller hierarchy via event synchronization. An event is initiated by a particular module and its recipients are controlled by an event dependency table which may be static or dynamic. An event may also initialize parameters within modules in the hierarchy. Command events take the general form $\text{do}^m_n(\text{command, params})$ where $m$ is the requesting controller module, $n$ is the receiving controller module, *command* is the task to be performed and may take on values such as *initialize, abort, etc.*, and *params* are parameters and initial states for the receiving module. Similarly, response events are in the general form $\text{done}^m_n(\text{response, results})$, where *response* is an indication of the completion of the commanded task and may take on values such as *normal, abnormal, etc.*, and *results* are parameters returned to the requesting module on task completion. Referring again to , let $\mathcal{O}$ denote the set of behaviors, $\mathcal{V}$ denote the set of operations, and $\mathcal{M}$ is the set of all possible missions. Similarly, each operation $o_j \in (\mathcal{O} + \mathcal{V})^*$ and each behavior $v_k \in \mathcal{V}^*$. The entire mission controller contains interacting hybrid automata as explained in the next section.

### III. Hybrid Mission Controller for a Survey AUV

Figure 2 shows the details of a specific application of the general AUV mission control architecture to a generic *survey AUV*. The primary mission of a survey AUV is to transit to a user specified location and conduct a survey following a specific pattern in 3D, at a specified speed and depth/altitude. In this example, there are three vehicle controllers (VCs), the *Autopilot* which accepts commands to control the altitude, speed and depth of the AUV; the *Variable Buoyancy System (VBS) Controller* which accepts commands to control the trim and buoyancy of the AUV; and the *Device Controller* which accepts commands to control the various sensors and other devices on board the AUV. Correspondingly, the vehicle state is comprised of the position of the AUV in three dimensions along with the velocity vector, the state of the buoyancy system, and the states of the various sensors and other devices on board.

The lowest level of this mission controller is comprised of two behavior controllers: *Steering* which is responsible for steering the vehicle to a specified location in space and interacts with the Autopilot; *Loiter* which controls the vehicle to loiter at a specific location in space for a specified duration and interacts with the Autopilot and VBS Controller. These behavior controllers issue appropriate commands for vehicle controllers and monitor their responses, via the vehicle state vector, to achieve their control objectives.

The behavior controllers are, in turn, commanded by the operation controllers which correspond directly to mission orders that are specified by the user and are described next. The *Pause* operation controller is used under certain situations to let the vehicle remain at it’s current state for a specified duration. The *Launch* operation controller is responsible for bringing the vehicle off of the surface and running at depth with enough forward speed to achieve controllability. This controller interacts with the Autopilot, the VBS Controller, and the Device Commander controller. The *GPSFix* operational controller sequentially commands the AUV to shut off propulsion, rise to the surface, raise the GPS mast, obtain a GPS-aided position fix retract the GPS mast, and re-launch the AUV. This controller interacts with the Autopilot, behavior controller, the Device Commander, the Device Controllers, and the Launch operation controller. The *WaypointNavigator* operation controller controls the AUV to transit to waypoints specified by the
mission specification. This controller interacts with Steering, Loiter, and the Device Controller. The Device

Finally, at the highest level of the AUV mission controller are the mission coordinators of which there are
two types: Progress and Safety, where the progress coordinator is divide into two parts: Sequential, and Timed. The sequential coordinator is responsible for executing a mission consisting of a sequence of operations; a timed coordinator is responsible for executing a timed sequence of operations; and a safety coordinator ensures safe operation of the vehicle. Timed operations have priority over sequential ones: When a timed operation is due, if necessary, the currently executed sequential operation is suspended until the timed operation has been executed. Sequential operation is then resumed until the next (if any) timed order is due. Safety coordinator has priority over all other coordinators. When an unsafe condition is detected, the commands from the safety coordinator supercede all other commands and seek to move the vehicle into a safe region or abort the mission if necessary. These priorities are implemented by the Command Conflict Manager located in the MC2VC interface. The hybrid model description of the hybrid systems follow next.

IV. HYBRID SYSTEM MODEL

Hybrid systems are systems which include continuous as well as discrete signals and components. Hybrid systems [19][24] have been used as mathematical models for many important applications, such as automated highway systems [4], [5], [6], air-traffic management systems [7][8], [9], embedded automotive controllers[10][11], manufacturing systems [12], chemical processes [13], robotics [14][15], real-time communication networks, and real-time circuits [16].Their wide applicability has inspired a great deal of research from both control theory and theoretical computer science [17 - 24], [13]. An AUV is a hybrid dynamical system with both discrete and continuous states. Hybrid systems can be modeled as hybrid automata. A hybrid automaton model captures the evolution of variables over time. The variables will either evolve continuously or in instantaneous jumps. A hybrid automaton is as shown below. This type of modeling formalism will be used to develop underwater vehicle modules.

Controlled hybrid automaton

A controlled hybrid automaton is a tuple $\mathcal{A}=(Q, \Sigma, \mathcal{U}, Y, F, H, I, E, G, R)$ consisting of the following components:

State space: $Q=L \times X$ is the state space of the hybrid automaton, where $L$ is a finite set of locations and $X = \mathbb{R}^n$ is the continuous state space. Each state $q \in Q$ can be described by $(l, x) \in Q$, where $l \in L$ and $x \in \mathbb{R}^n$.

Events: $\Sigma$ is the finite alphabet or event set of $\mathcal{A}$.

Continuous Controls and Parameters: $\mathcal{U} = \mathbb{R}^m$ is the continuous control space consisting of control signals and exogenous continuous-time parameters. $u : [0, \infty) \rightarrow \mathcal{U}$ denotes a control vector comprised of these parameters.

Outputs: $Y$ is the output space of $\mathcal{A}$, which may consist of both continuous and discrete elements.

Continuous Dynamics: $F$ is a function on $L \times U$ assigning a vector field or differential inclusion to each location and continuous control vector. We use the notation $F(l, u) = f_J(u)$.

Output Functions: $H$ is a set of output functions, one for each location $l \in L$. We use the notation $H(l) = h_l$, where $h_l : X \times U \rightarrow Y$ is the output function associated with location $l \in L$.

Invariant conditions: $I \subseteq 2^X$ is a set of invariant conditions on the continuous states, one for each location $l \in L$. We use the notation $I(l) = i_l \subseteq X$. If no $i_l$ is specified for some $l \in L$, then it's default value is taken to be $X = \mathbb{R}^n$.

Edges: $E \subseteq L \times \Sigma \times L$ is a set of directed edges. $e = (l, \sigma, l') \in E$ is a directed edge between a source location $l \in L$ and a target location $l' \in L$ with event label $\sigma \in \Sigma$. In addition, $E = E_i \cup E_c$, where $E_i$ and $E_c$ represent the controlled and uncontrolled edges, respectively.

Guard conditions: $G \subseteq 2^E$ is the set of guard conditions on the continuous states, one for each edge $e \in E$. We use the notation $G_e = g_e \subseteq X$. If no $g_e$ is explicitly specified for some edge $e \in E$, then it's default value is taken to be $X = \mathbb{R}^n$.

Reset conditions: $R$ is the set of reset conditions, one for each edge $e \in E$. We use the notation $R(e) = r_e$, where $r_e : X \rightarrow 2^X$ is a set-valued map. If no $r_e$ is explicitly
specified for some edge $e \in E$, then the default value is taken to be the identity function.

**Interacting Controlled Hybrid Automata**

In order to cope with complexity of real-life applications it is often convenient to model a hybrid system in a modular fashion as a set of interacting hybrid automata, \( \mathcal{H} \). Each hybrid automaton in the set is a tuple as before, \( \mathcal{H} = \{ \mathcal{Q}, \Sigma, U', Y', F', H', I', E', G', R' \} \).

The interaction among various hybrid autonomous modules takes place through event synchronization and sharing of variables in invariant and guard conditions, as follows.

**Invariant Conditions:** For each 
\( l \in L', L'(l) \subseteq X' \times \Pi, Y' \).

**Guard Conditions:** For each 
\( e \in E', G'(e) \subseteq X' \times \Pi, Y' \).

The other components of the tuple are analogous to those of the single hybrid automaton defined above.

For an event \( \sigma \in \Sigma = \bigcup \Sigma^j \), let \( In(\sigma) = \{ j : \sigma \in \Sigma^j \} \) be the set of indices of the event sets that contain the event \( \sigma \). Then each \( \sigma \) -step must be taken synchronously by each of the hybrid automata \( \mathcal{H}^j \) such that \( j \in In(\sigma) \). In other words, for each \( j \in In(\sigma) \), \( (l'_j, x'_{i}) \xrightarrow{\sigma}(l'_j, x'_{i}) \) if and only if

(a) \( e' = (l'_j, \sigma, l'_i) \in E' \)
(b) \( x'_{i} \in g_{i,j} \cap i'_{l,j} \)
(c) \( x'_{i} \in l'_{i,j} (x'_{i}) \cap i'_{l,j} \mathcal{E} \).

In each of the hybrid systems a state consists of a location and a value of continuous states. Based on inputs received, continuous states evolve according to a differential equation (flow) associated with the location and outputs are produced all along such evolution. When system states and applied inputs satisfy a guard condition associated with one of the transitions defined at the location that transition is executed leading the system to a new location and resetting the continuous states. Interaction occurs by sharing inputs/outputs and transition labels.

A. **Teja: Hybrid system modeling tool**

Teja allows the creation of a system architecture where all the modules required for a particular mission controller are instantiated and initialized, and their interactions are specified via an event dependency table which may be dynamically reset. Automatic code generation ensures that the real-time scheduling needs are met to tolerances far exceeding the mission control application.

Teja allows for abstract class definitions and inheritance so that, when appropriate, generic controller classes may be defined and subclasses may be used to refine and customize the generic controllers to specific applications. Utilities are provided to handle useful functionality such as communications and data handling and parsing. Libraries and utilities are provided for a variety of commonly used platforms and operating systems including Window, Linux, and Solaris. All of these features make Teja an ideal tool for rapid prototyping, testing, and deployment of mission controllers on target vehicle platforms.

Figure 3-5 shows the hybrid automaton representation of the GPSFixer, Launch operation controller and steering behavior controller modeled using the Teja NP tool [24]. Transitions between states may be proactions, where the transition fires when the guard condition is true, or responses which fire on event synchronization from another hybrid automaton. In Teja, the first portion of an event label is either a local label in the case of a proaction, or a synchronization label in the case of a response. The second portion, after the \( / \), represents an output event label that is used to fire enabled response transitions in other modules that are specified in a (static or dynamic) event dependency table for that particular event label. Resets and other initializations may be performed on transitions between states. The hybrid automata modules that make up a particular application therefore interact through event synchronization. An initial state is specified and the Teja tool allows constructors and destructors to initialize and finalize the state variables, and parameters of each automaton. Vehicle state values, on receipt from the interface level, are used to populate Teja data structures which are available to all modules that require access to them. Links to other automata are provided so that public data within them may be accessed and set. These links are used to pass parameters and initial conditions, and retrieve results on event transitions.

On initialization GPSFixer transitions to the **Idle** state from **Start**. The GPSFixer transitions to **GoToSurface** on receiving **TakeGPSFix** from the higher level controllers. If the AUV fails to reach the surface in time the GPSFixer transitions to **ReportTO** state the event being **TimeOut**. If the GPSFixer reaches the surface successfully the GPSFixer transitions to the **RaiseMast** state on the event **OnSurface**!. If the mast is not raised in time the GPSFixer transitions to the **ReportTO** state the event being **TimeOut**. When the mast is raised the GPSFixer transitions to the **TakeFix** state on the event **MastUp**. After updating the navigation system the GPSFixer passes control to the Launch controller to lower the mast and the AUV by outputing the event **Launch** and transitions to the **ComeOffSurface** state. The Launch controller then goes through its sequence of events to lower the mast and bring the AUV below the surface of water as shown in Figure 4. Then the Launch controller passes control to the GPSFixer controller on the event **LaunchDone**. The GPSFixer transitions to the **Decide** state where it decides whether to return back to the original location before starting GPSFix mission or to just go to a particular depth. If the AUV needs to return to the original location the GPSFixer passes control to the Steer controller by outputing the event **Steer**. The Steer controller then
executes its sequence of events (Figure 5). Once the AUV reaches the destined location the Steer controller passes control to the GPSFixer by outputting the event SteeringDone. The GPSFixer transitions to Decide state. The GPSFixer finally ends the mission by transitioning to the Idle state by sending the output GPSFixDone to the concerned controller which can be TC or SC.

Figure 3: The GPSFixer Operation Controller

Figure 4: The Launch Operation Controller

Figure 5: The Steering Behavior Controller

V. ANIMATION

Animation in our case deals with animating the sequence of operations and behaviors the survey AUV executes to successfully complete a mission. A mission consists of several combinations of discrete and continuous parameters which might damage the AUV in the real environment. But in an animated environment several combinations of parameters can be considered within a mission without damaging the actual AUV. The animation shows the effects of such parameter changes on the actions of the AUV which can then be implemented or discarded.

The animations involve the interaction of all the modules involved in a mission without any abstraction. Thus animation further strengthens the correctness of the modules and their interaction. These reasons motivated the research to develop an automated method to animate the mission executed by the AUV. The graphic tool used and the proposed method follows next.

A. OpenGL: Tool for Animation/Simulation

OpenGL [34] is a software interface to the graphics hardware. OpenGL is a hardware independent interface to be implemented on many different hardware platforms. OpenGL contains commands to draw geometric primitives like points, lines, and polygons to build the desired model. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered into the frame buffer. OpenGL is like a state machine, the state being defined by color, current viewing, projection transformation, polygon drawing mode, characteristics of light etc. OpenGL also supports animation of graphical models drawn. Thus using OpenGL we can move or rotate or involve translation of an object the way we want.

GLUT (OpenGL Utility Toolkit) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres.

A typical program that uses OpenGL begins with calls to open a window into the frame buffer into which the program will draw. Then, calls are made to allocate a GL context and associate it with the window. Once a GL context is allocated, OpenGL commands can be issued. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user's two- or three-dimensional model space to the two-dimensional screen.

B. Algorithm for conversion

We created a converter coded in Perl. The inputs to the converter are the coordinator modules involved in a specific mission. The coordinator modules are modeled using real time verification tool Uppaal (explained in section 4) which are .xml files. After opening the coordinator modules the converter extracts important information from the UPPAAL files and generates a graphics file in OpenGL. The information extracted are the different events received or sent, and the variables used.

The converter searches the sequential/timed coordinator file, extracts the operation name which is an event on a transition within the sequential/timed coordinator. Then the converter searches the file among the set of input files which models a transition on the same event. The converter then keeps extracting and expanding the sequence of events. This way we can incorporate the interaction between the modules working together. The expansions model the sequence of actions (algorithms) executed by the concerned controller modules.
The code that is generated can then be run using the commands used to run an OpenGL program. The parameters required for an operation can be changed within the files which changes certain actions executed by the AUV. A brief description of OpenGL, converter code and explanation for a simple module is explained next.

We use a bottom up approach for simulation. We first simulate the actions implemented by the lowest level controllers. Once the sequences executed by the lower level controllers are simulated we combine the higher level controllers with the lower ones. We used this approach because the parts of mission executed by the lower level controllers are called for by the higher level ones. So this gives us an organized way to build up the simulation of the model and validates the correct operation of all the modules working together.

The OpenGL modules contain codes which expand each of the sequences to be executed to successfully complete an operation within a mission. In the present simulation model sensor values are stored in common files. The modules collect the sensor information and other parameter changes from within the common files accessed. The modules then execute the sequence of actions according to the inputs received. After completing the operations the changed parameters are then written back to the common files like time, position etc. this way the information gets updated. The next operation to be executed gathers information from the common files before starting to execute the sequence of actions. The algorithm used for the conversion of the Uppaal modules to the OpenGL code is as given below. Then follows a flowchart which shows the way the converter works and then follows a few screen shots like the AUV with the mast underwater (Figure 6) raising mast after reaching surface of water (Figure 7) and the abstracted converter code for steering module.

- For i = n to 1 (where n is the lowest level)
  - For k = 1 to m (where m is the total number of modules within a level)
    - Input the hybrid model \textbf{H} at the Level, to the converter
    - The converter extract events \( \sigma \), guard \( g \), and variables \( \textit{Vars} \)
    - Generate OpenGL code to model the events, guards using the variables involved.
  - Next k
- Next i

The abstracted code for the Steer module follows next.

VI. ABSTRACTS OF CONVERTER CODE FOR STEERING MODULE

This section contains abstracted code used for conversion of the steering controller module. All the conversion modules have a similar structure only differing based on the events and guards. All the modules have an initialization phase in which all the variables are initialized and the initialization code for the animation is generated. Then follows the extraction-expansion phase during which events, variables and guards are extracted from the controller modules and expanded to incorporate the sequence of actions executed for an operation. Finally the animation code for graphical representation and keyboard/mouse association is generated. The abstracted code for the steering module is as given next.

- **Initialization** (initializes the variables used in converter)
- **Generate the initialization code**
  - print OUTFILE "\#include <stdlib.h>";
  - print OUTFILE "\#include<stdio.h>";
  - print OUTFILE "\#include<time.h>";
  - print OUTFILE "\#include<math.h>"
- **Input the steering module check for the number of lines of declaration of variables**
  - while($input = \!<STEERFILE>\>)
    - if ($input =~ /\sint\s/)  
      - $numbertimesint++; # Keeps track of iterations
      - print"Integers = $numbertimesint\n";
    -
  - **Generate all the variables extracted**
    - if($in =~ /\sint\s+:\d/)  
      - $stringofdecl = $&; # Store the pattern in a string
        @arrayofdecl = split(’,’, $stringofdecl); # Get rid of int
        $stringofvar = $arrayofdecl[1]; # Get the element of the array which contains the string of variables
        @actualvar = split(‘,’,$stringofvar); # split it based on commas to get the total no of variables
        for ($count = 0; $count<=$#actualvar; $count++) 
          - $stringid = $actualvar[$count]; # Store each variable in a string
            %sepvalue = split(‘:’,$stringid); # Split variables based on colon
            print OUTFILE "static GLfloat "; print OUTFILE "%s"; print OUTFILE ":"; print OUTFILE "\n";
    - $getvariable++;
- **Generate the initialization modules and initialization variables**
  - print OUTFILE "\nvoid init(void){...}
  - print OUTFILE "\nvoid display(void){...}
- **Extract information for the Steer operation**
  - if($in =~ /\!Steer\W\!\<\)  
  - **Extracting event abort and expanding its sequence**
    - if($in =~ /\!Abort\W\!\<(&& $abortNumber == 0) {...}
VII. CONCLUSION

Animation/Simulation of missions executed by AUVs modeled using hybrid, model-based architecture for mission control is presented. About 4500 lines of converter code has been written for the present design and missions. A bottom up approach to animation/simulation is accomplished. A generic survey AUV mission controller conforming to this architecture, had been developed which was successfully converted to animation code. Development of advanced animation tool incorporating sensor feedback from the AUV is part of future research work. Automated synthesis of the highest-level coordinators is also envisioned and planned for future study.

REFERENCES

[26] www.uppaal.com
[34] OpenGL programming guide – Redbook version 1.4.