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) [10] that facilitates modeling, and verification [11] of logical correctness of the mission controller/AUV. Here we propose a method for simulation/animation of the missions executed.

The control tasks for an autonomous underwater vehicle is be divided into lower level control, concerned with continuous dynamics and a higher-level mission control, which is discrete, either event-driven, or time-driven. Thus the overall system is a hybrid system containing both continuous and discrete dynamics. The mission controller 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. Controllers at each of the layers coordinate those at the lower layers. Each of the mission controller module is modeled as a hybrid system. Hierarchical approach reduces the complexity of design and also facilitates the verification, simulation and potentially the automated synthesis of the highest level mission controller module(s).

Animation/Simulation imitates or estimates how events might occur in a real situation. Usage of a simulation tool offers the advantage of visually inspecting the correctness of design without risking damage to the vehicle or equipments involved. A simulation in addition to verification further strengthens the correctness of a designed system. This is because the verification of a hybrid system is mostly carried out by abstracting the system [11] which might miss situations that might occur. No such abstraction is needed for simulation. So a simulation can capture some interactions which might be missed while carrying out verification as described in [3]. Also an animation/simulation tool aids in the creation of the realistic environments to which the autonomous system must react appropriately. A simulation tool had been developed for the automated highway system in the PATH project at Berkeley [1][2]. The use of such a simulation tool proved to be useful for the reasons explained above. The preliminary simulation tool we developed is a basic tool for establishing the possibility of having an advanced simulation tool in future. The simulation tool is specific to the survey missions for an AUV. OpenGL is used to simulate/animate the missions executed by an AUV.

The mission controller modules are developed using TEJA software tool [5], which supports the design of interacting hybrid state machines and includes automatic real-time code generation allowing for a rapid deployment on the target platform. For verification purposes, the Teja modules specifications are first transformed [6] into a format readable by Uppaal [6], a hybrid system modeling, simulation, and verification tool. For animation, the mission controller modules in Uppaal are further converted to animation modules of OpenGL [9].
II. HYBRID MISSION CONTROLLER ARCHITECTURE

Our mission control architecture is designed with semi-automatic verification in mind. The hybrid automata implemented using Teja can be converted into a format that is readable by available verification tools such as HyTech [8] and Uppaal. 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 hybrid mission controller is organized hierarchically as shown in Figure 1 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 command in the shared database. At the lowest level of the hierarchy is the underwater vehicle (plant) along with the vehicle controllers (VCs).

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.

As shown in Figure 1 mission controller is organized in a three-tier hierarchy. 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. The lowest level of the mission controller is comprised of 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 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. Operations, 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.

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.

An event is initiated by a particular module and its recipients are controlled by an event dependency table which may be static or dynamic. The entire mission controller contains interacting hybrid automata, which is formally defined in a section below.

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.

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. 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 Commander* is used to control sensors and devices on the AUV in response to mission orders; this controller interacts with the Device Controllers. 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.

IV. HYBRID SYSTEM MODEL AND TEJA/UPPAAL TOOLS

Hybrid systems are systems which include continuous as well as discrete signals and states. Hybrid systems [4] [10] have been used as mathematical models for many important applications. Their wide applicability has inspired a great deal of research from both control theory and theoretical computer science [7].

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 evolve continuously as well as in instantaneous jumps. A hybrid automaton is as described below. This type of modeling formalism has been used to model the underwater vehicle control modules.

A. Controlled hybrid automaton

A controlled hybrid automaton is a tuple $\mathcal{G}= (Q, \Sigma, U, Y, F, H, I, E, G, R)$ 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 $\mathcal{H}$.

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

**Outputs:** $Y$ is the output space of $\mathcal{H}$, 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 $F(l, u) = f(l)(u)$.

**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 $I(l) = i \subseteq X$. If no $i$ 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, $E = E_c \cup E_u$, where $E_c$ and $E_u$ represent the controlled and uncontrolled edges, respectively.

**Guard conditions:** $G \subseteq 2^X$ 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

![Figure 2: Survey mission control architecture](image-url)

The sequential coordinator is responsible for executing a timed sequence of operations; and a safety coordinator ensures safe operation.
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 \( r_e : X \to 2^X \) 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.

The semantics of an hybrid automaton can be understood as follows. When in a certain discrete configuration \( l \), the continuous-state \( x \) of the hybrid system evolves according to the controlled vector-field \( F_j(x,\mu) \). The evolution of the continuous-state according to the flow of \( F_j(x,\mu) \) is defined as long as \( x \) lies in the domain specified by the invariant condition \( I_i \).

If at anytime during its evolution the continuous-state acquires a value that satisfies a guard condition \( g_e \) for some edge \( e \in \{l, \sigma, l'\} \) of the hybrid automaton, the system can transition from configuration \( l \) to \( l' \). The transition is labeled by an "event" \( \sigma \), and the continuous-state in the new configuration acquires a value specified by the reset condition \( r_e \). When in new configuration \( l' \) the continuous-state evolves according to the controlled vector-field \( F_j(x,\mu) \).

### B. Interacting Controlled Hybrid Automata

It is convenient to model a hybrid system in a modular fashion as a set of interacting hybrid automata, \( \{H_i\} \). Each hybrid automaton in the set is a tuple as before:

\[
\mathcal{H} = \{Q_i, \Sigma_i, U_i, Y_j, F_j, I_i(j), E, G_i, R_i\}
\]

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 \( l \in L_i, I_i(l) \subseteq X_i \times \prod_{k} Y_k \), where \( k=1\ldots j-1, j+1\ldots n \).

**Guard Conditions:** For each \( e \in E_i \),

\[
G_i(e) = g_{e,l} \subseteq X_i \times \prod_{k} Y_k
\]

where \( k=1\ldots j-1, j+1\ldots n \).

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

**Event Synchronization:** For an event \( \sigma \in \Sigma = \bigcup_j \Sigma_j \), let

\[
\text{In}(\sigma) = \{ j \mid \sigma \in \Sigma_j \}
\]

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 \( \mathcal{H}_j \) if \( j \in \text{In}(\sigma) \), the corresponding guard condition \( g_{e,l} \) is satisfied, and the invariant condition \( I_i(l) \) of the accepting state is satisfied. In other words, for each \( j \in \text{In}(\sigma) \), \( (l_i^1, x_i^1) \xrightarrow{\sigma}(l_i^2, x_i^2) \) if and only if (a) \( e^j = (l_i^1, \sigma, l_i^2) \in E_i \) (b) \( x_i^j \in g_{e,l} \cap I_i \) and (c) \( x_i^j \in r_{e,l}((x_i^1) \cap I_i^2) \).

### C. Modeling Mission Modules in TEJA

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.

Figure 3-4 shows the hybrid automaton representation of the GPSFixer, Launch operation controller and steering behavior controller modeled using the Teja NP tool. On initialization the modules go to the **Idle** state from **Start**. The GPSFixer (Figure 3) then goes to the surface, raises mast, updates the navigation system and then passes control to the Launch controller to lower the mast and then on the event **Launch** transitions to the **ComeOffSurface** state. The Launch controller then goes through its sequence of events shown in Figure 4 to lower the mast and lower the AUV below the surface of water. 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 outputting the event **Steer**. The Steer controller then executes its sequence of events (Figure 4). Once the AUV reaches the destined location the Steer controller passes control to the GPSFixer by outputting the event **GPSFixDone**. The GPSFixer finally ends the mission by sending the event **GPSFixDone** to the TC or SC.
D. Verification using Uppaal

The hybrid system models in Teja are converted into timed models supported by Uppaal. Uppaal is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types. The tool is designed to verify systems that can be modeled as networks of timed automata extended with integer variables, structured data types, and channel synchronisation. It contains two parts a graphical user interface and a model check engine. The user interface is implemented in java and executed on users work station. The model-checker Uppaal is based on the theory of timed automata and its modeling language offers additional features such as bounded integer variables and urgency. A timed automaton is a finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. The query language of Uppaal used to specify properties to be checked, is a subset of CTL (computation tree logic). The tool consists of an editor, a simulator and a verifier.

V. ANIMATION

Animation in our case deals with animating the sequence of operations and behaviors the survey AUV executes to successfully complete a mission. The graphic tool used and the proposed method for animation follows next.

A. OpenGL: Tool for Animation/Simulation

OpenGL is a hardware independent interface that can be implemented on many different graphics 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.

B. Algorithm for conversion from Uppaal to OpenGL

We created a converter coded in Perl. It takes as input a controller module .xml file that is created for performing verification using the tool, Uppaal. The conversion of Teja modules to Uppaal-readable .xml files is also done automatically using another Teja2Uppaal converter that has been created at ARL-PSU.

Starting from the coordinator module files the converter extracts important information 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, and extracts the operation name which is a transition-label within the sequential/timed coordinator. Then the converter searches the file among the set of input files which models the named operation. In this manner the converter keeps extracting and expanding the sequence of transition-labels from one module down to another module. The expansions capture the sequence of actions (algorithms) executed by the concerned controller modules and maintain the interactions of the various 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.

We follow a bottom up approach for simulation/animation. 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 correct simulation of the model. 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. The modules then execute the sequence of actions according to the inputs received. After completing the operations the changed parameters, such as time, position, etc., are then written back to the common files. The next operation to be executed gathers information from the common files before starting to execute simulation.

The algorithm used for the conversion of the Uppaal modules to the OpenGL code is as given below, which is followed by a few screen shots for the GPSFix mission (explained in IV A): The AUV with the mast moving up to the surface of water (Figure 6), AUV raising mast after reaching the water surface to update the navigation system (Figure 6) and results during GPSFix operation execution (Figure 7).

For i = n to 1 (where n is the lowest level)
  For k = 1 to m (where m is the number of modules in a level)
    Input the hybrid automaton \(\mathcal{H}\) at the Level, to the converter
    The converter extract events \(e\), guard \(g\) and variables \(\text{Vars}\)
    Generate OpenGL code to model the events, guards using the variables extracted
    Next k
  Next i

VI. ABSTRACT CONVERTER CODE FOR A MODULE

This section presents the abstracted code that is used for the conversion of the steering controller module (a part of GPSFix execution as explained in section IV A) from the Uppaal format to the OpenGL format. All the other conversions are
performed in a similar manner, only differing in the event names and their guards. There exists an initialization phase in which all the variables of a module 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 abstracted code for the steering module is given next.

#Initialization  (initializes the variables used in converter)
# Generate the initialization code
{ print OUTFILE "#ifndef include<stdio.h>"
;

# Input the steering module check for the number of lines of declaration of variables
while($input = <STEERFILE>)
  {
    if ($input =~ /int /)
      {
        $numbertimesint++; # Keeps track of iterations
        print'Integers = $numbertimesint
";
      }

#Generate the initialization modules and initialization variables
print OUTFILE "void init(void)";

# Extracting event abort and expanding its sequence
if($in =~ /&/& AbortNumber == 0) {

# Generate the graphics of the AUV and underwater env.
print OUTFILE "void reshape (int w, int h)";

# Associating mouse and keyboard related actions
print OUTFILE "void keyboard (unsigned char key, int x, int y)
";

VII. CONCLUSION

Animation/Simulation of missions executed by AUVs whose mission controller is modeled in a hierarchical hybrid, model-based architecture is presented. About 4500 lines of converter code has been written in Perl for the animation/simulation of an existing mission controller for performing the task of surveying. The converter code takes as input Uppaal specifications of the controller modules and extracts OpenGL executable code in an automated fashion. This is done in a bottom up fashion and the algorithm for doing the same is presented. Development of a more advanced animation tool incorporating the sensor feedback is a part of the future research work.

REFERENCES
[9] OpenGL programming guide – Redbook version 1.4