2005 American Control Conference
June 8-10, 2005. Portland, OR, USA

Hybrid-Model based Hierarchical Mission Control Architecture for Autonomous Underwater Vehicles

S. Tangirala, R. Kumar, S. Bhattacharyya, M. O'Connor, and L. E. Holloway

Abstract— **We present a hybrid, hierarchical architecture for mission control of autonomous underwater vehicles (AUVs). The architecture is model based and is designed with semi-automatic verification of safety and performance specifications as a primary capability in addition to the usual requirements such as real-time constraints, scheduling, shared-data integrity, etc. The architecture is realized using a commercially available graphical hybrid systems design and code generation tool. While the tool facilitates rapid redesign and deployment, it is crucial to include safety and performance verification into each step of the (re)design process. A formal model of the interacting hybrid automata in the design tool is outlined, and a tool is presented to automatically convert hybrid automata descriptions from the design tool into a format required by two hybrid verification tools. The application of this mission control architecture to a survey AUV is described and the procedures for verification outlined.**

I. Introduction

We present an architecture for high-level control of complex systems such as AUVs that is hierarchical, hybrid, and model-based. This architecture is hierarchical in order to manage complexity; hybrid to more accurately capture the discrete and continuous nature of AUV control systems; and model-based so that it may be represented mathematically, and hence formally and analytically verified against a set of requirements. Our goal is to develop a high-level control architecture and an associated tool set to facilitate graphical design, rapid prototyping, faster-than-real-time simulation, code generation for easy deployment on target platforms, and to *incorporate formal verification* of a set of requirements into every stage of the design process. While the impetus for the development of this architecture has been the control of autonomous underwater vehicles (AUVs), it is general enough to be broadly applicable. The mission controller has been under development at the Applied Research Laboratory at the Pennsylvania State University and has benefited from discussions with collaborators from Iowa State.

Manuscript received September 15, 2004. This work was supported in part by the National Science Foundation under the grants NSF-ECS-0218207, NSF-ECS-0244732, NSF-EPNES-0323379, and NSF-ECS-0424048, a DoD-EPSCoR grant from the Office of Naval Research under the grant N000140110621, and a Exploratory and Foundational Research Grant from ARL.

S. Tangirala (phone: 814-863-7594; fax: 814-863-8783; email: cxt148@psu.edu) and M. O'Connor are with the Applied Research Laboratory at Penn State University.

R. Kumar is with the Iowa State University, Dept. of Electrical and Computer Engineering.

S. Bhattacharyya and L. E. Holloway are with University of Kentucky, Dept. of Electrical and Computer Engineering.

AUVs are complex, large-scale, highly nonlinear, time-varying, stochastic, and operate in an uncertain and unpredictable environment. Vehicle control systems for AUVs typically have several communicating subsystems/modules which need to interact amongst themselves and the environment via sensors to successfully execute a mission within satisfactory real time bounds. In order to manage the complexity and uncertain operating environment, control systems for AUVs are in general, hierarchical. Lower level control such as speed and attitude control is typically dynamic-model based and is designed using conventional or modern control methodologies. The higher control levels are usually more abstract and include additional requirements such as re-configurability, learning, safety, failure, and exception handling, the ability to manage dynamically changing mission goals, multi-system coordination, and increased autonomy. Additionally, concerns of real-time execution, shared-data integrity and high-level programming have to be addressed.

Traditionally, artificial intelligence methods are encountered in literature to deal with high-level programming. Several programming and control architectures have been developed for high-level control of mobile-robots, in general, and AUVs, in particular [1]-[3]. Some, such as planning-based systems, are not suitable for real-time operations in systems of reasonable complexity. Other approaches, such as behavior-based systems, were developed to answer real-time concerns and provide flexibility, but many of them lack a rigorous set of definitions and an associated systems analysis. While the focus of intelligent control architectures has been the use of technologies such as adaptation, learning, etc., to facilitate safe execution of missions in complex environments, our focus is additionally on real-time operations, automatic code-generation, and semi-automatic verification of safety and progress at every design stage. To this end, the architecture proposed here is model-based and hierarchical. The models we use are general hybrid dynamical systems, where the enabling conditions and output actions associated with state transitions can be general functions, and real-time constraints are explicitly taken into account. The mathematical modeling of the controller allows the application of formal verification methods to guarantee that some predetermined (safety and/or progress) specification is met.

In our approach the control tasks for an AUV can broadly be divided into lower level control, concerned with continuous dynamics, and high-level control, which is typically discrete and event/time-driven. In this paper, we refer to the lower level of control as the Vehicle Control (VC) and to the higher level of control as Mission Control (MC). The overall system is therefore a hybrid system containing both continuous and discrete states. The basic idea is to hierarchically decompose AUV missions into sequences of *operations*, operations into sequences of *behaviors*, and behaviors into sequences of vehicle *maneuvers*. A mission/operation can also contain commands for vehicle maneuvers. Then, each level of the hierarchy coordinates the level below it to accomplish specific tasks. The MC design and verification is then accomplished in a bottom-up fashion, starting with the behavior controllers, which coordinate vehicle controllers, moving up to operation controllers, which coordinate the behavior controllers; and finally, a coordinator for each type of mission specification (e.g., safety and progress), which coordinates the operation controllers.

The mission controller modules are designed using TEJA NP networking software tool [4]. 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, which are required to conform to a formal model, are converted into a format readable by UPPAAL [5], a hybrid system modeling, simulation, and verification tool. Abstractions are used to reduce the order of the verification process at any level, and verification proceeds in a bottom-up fashion until the entire controller is verified. Section II describes our hybrid mission control architecture, and Section III applies this architecture to a survey AUV. Section IV outlines the hybrid systems modeling framework that is used to formalize the mission controller modules and describes the tool and procedure for formal verification of safety and performance of a specific mission control system.

II. Hybrid Mission Controller Architecture

The hybrid mission controller is organized hierarchically as shown in Figure 1 below. Each of the modules that make up the mission controller hierarchy is a hybrid system, and the entire mission controller is modeled as a set of interacting hybrid systems. Modules at any level may command other modules at that and lower levels and send responses to that and higher levels. All levels in the mission controller hierarchy may assign vehicle commands directly by placing appropriate vehicle commands in the shared database. At the lowest level of the hierarchy is the underwater vehicle (plant) along with the vehicle controllers (VCs). The vehicle and 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.

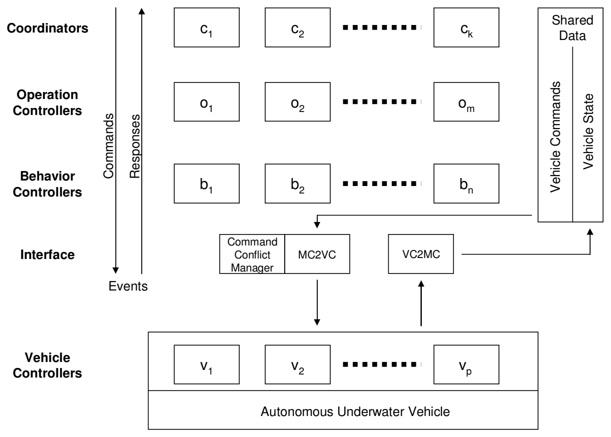


Figure 1: Hybrid Mission Control Architecture

As seen in Figure 1, the mission controller is organized in a three-tier hierarchy, and all communication between modules is restricted to event synchronization and shared data. 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 that may be static or dynamic. An event may also initialize parameters within modules in the hierarchy. Command events take the general form $do_m^n(command, params)$, 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 $done_n^m(response, results)$, 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. 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 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 is user/mission-centric. These correspond directly to user supplied mission orders and 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: *Progress*, further divided into two parts: *Sequential*, and *Interrupt-driven*; and *Safety*. The sequential coordinator is responsible for executing a mission consisting of a sequence of operations; the interrupt-driven coordinator is responsible for executing a time or state-based interrupt driven sequence of operations; and a safety coordinator ensures safe operation of the vehicle. When an interrupt-driven operation is due, the currently executing sequential operation is suspended, if necessary, until the interrupt-driven operation has been executed. Sequential operation is resumed until the next (if any) interrupt-driven order is due. Interrupts are classified and prioritized so that some may have priority over sequential operations, while others do not and may therefore not be able to interrupt certain classes of sequential operations. The safety coordinator has priority over all other coordinators. When an unsafe operating 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 and by event dependencies and synchronization.

A mission is therefore 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 sensors and periodically transferred by the VC2MC interface 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. Formally, let $\mathcal{B}$ denote the set of behaviors, $\mathcal{O}$ denote the set of operations, and $\mathcal{V}$ denote the set of vehicle subsystem controllers. A mission, m is defined as $m \in \mathcal{M} \subset (\mathcal{O}+\mathcal{V})^*$, where $(\mathcal{O}+\mathcal{V})^*$ is the set of all sequences containing elements of $\mathcal{O}$ and $\mathcal{V}$, and $\mathcal{M}$ is the set of all possible missions. Similarly, each operation $o_j \in (\mathcal{B}+\mathcal{V})^*$, and each behavior $b_k \in \mathcal{V}^*$.

Teja, the design tool used to implement this mission control architecture, 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 that 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 Windows, 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.

III. Hybrid Mission Controller for a Survey AUV

The details of a specific application of the general AUV mission control architecture to a generic *survey AUV* are seen in Figure 2. 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 or altitude. In this example, there are three vehicle controllers (VCs), the *Autopilot*, which accepts commands to control the attitude, 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 onboard devices.

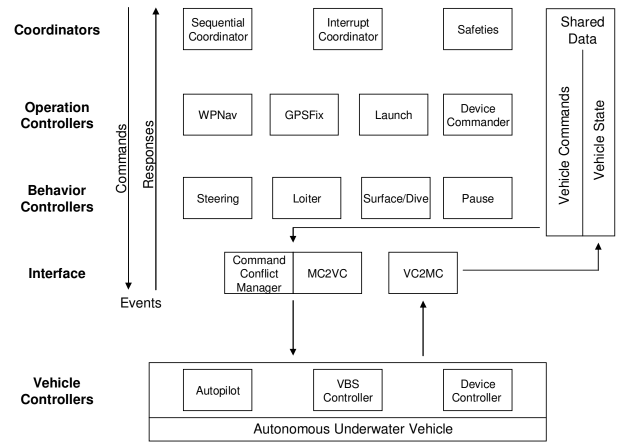


Figure 2: Survey AUV Mission Controller

The lowest level of this mission controller is comprised of four 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; *Surface/Dive*, which commands the vehicle to go to or come-off of the surface and interacts with the Autopilot and the VBS Controller; and *Pause*, which is used under certain situations to let the vehicle remain at it's current state for a specified duration. These behavior controllers issue appropriate commands to the 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 *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, the Device Commander, and the Surface/Dive behavior controller. The *GPSFix* operation 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, the Surface/Dive behavior controller, the Device Commander, the Device Controller, 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 Controller.

IV. Formal Modeling and verification

The complexity of AUV systems, control systems, and missions, and the difficulty in reproducing exhaustive operational scenarios, including sensor data in simulations, dictates the need for analytical verification of some measure of correctness of the mission controller/vehicle controller combination and the mission itself to eliminate many common errors and oversights. This is an area of ongoing research and some results and tools are available such as HyTech [8], and UPPAAL [1], [5], both of which perform automatic symbolic verification for a specific class of linear hybrid systems.

Our mission control architecture is designed with semi-automatic verification as an integral capability. To this end, and mindful of the state-of-the-art in verification tools, certain restrictions are imposed on the hybrid modules that make up the mission control architecture. The logic within individual automata is restricted to use clocks and variables, whose dynamics are represented by differential inclusions, as the only continuous variables; all other continuous dynamics are encapsulated in functions. Interactions between modules are restricted to event synchronizations and shared data. Hybrid automata have been used as mathematical models for many important applications, such as automated highway systems, air-traffic management systems, embedded automotive controllers, manufacturing systems, chemical processes, robotics, real-time communication networks, and real-time circuits. Their wide applicability has inspired a great deal of research from both control theory and theoretical computer science. The formalism presented in this section is used to model our mission control architecture -- individual modules are modeled as controlled hybrid automata and the mission controller, as a whole, is modeled as a set of interacting controller hybrid automata.

A. Controlled hybrid automaton

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

State space: $Q = L\times X$ is the state space of $\mathcal{H}$, L is a finite set of locations and $X = \Re^n$ is the continuous state space.

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

Continuous Controls and Parameters: $U = \Re^m$ is the continuous control space consisting of control and exogenous continuous-time parameters. $u:[0,\infty)\to U$ is a vector comprised of these controls and 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. The notation $F(l,u) = f_l(\cdot,u)$ is also used.

Output Functions: P is a set of output functions, one for each location $l \in L$. $P(l) = p_l : X \times U \to Y$ is the output function associated with location $l \in L$.

Invariant conditions: $I \subset 2^X$ is a set of invariant conditions on the continuous states, one for each location $l \in L$. The notation $I(l) = i_l \subseteq X$ is also used. At each $l \in L$, the default value is $i_l = X$.

Edges: $E \subset L \times \Sigma \times L$ 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 $\sigma \in \Sigma$.

Guard conditions: $G \subset 2^X$ is the set of guard conditions on the continuous states, one for each edge $e \in E$. The notation $G_e = g_e \subseteq X$ is also used. If no g_e is explicitly specified for some edge $e \in E$, then the default value is $g_e = X$.

Reset conditions: R is the set of reset conditions, one for each edge $e \in E$. The notation $R(e) = r_e$ is used, 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 the identity function.

Definition - σ-step: For $\sigma \in \Sigma$, a σ-step is a binary relation $\overset{\sigma}{\to} \subset Q \times Q$ and it is true that $(l, x) \overset{\sigma}{\to} (l', x')$ if and only if (a) $e = (l, \sigma, l') \in E$, (b) $x \in g_e \cap i_l$ and (c) $x' \in r_e(x) \cap i_{l'}$. A σ-step is a transition or jump between discrete states. A σ-step need not be taken even if $x \in g_e$, but some σ-step must be taken before it holds that $x \notin i_l$.

Definition - t-step: Let $\varphi_t^l(x, u)$ be a trajectory of $f_l(\cdot, u)$ with initial state *x* and evolving for time t. For $t \in \Re^+$, a t-step is a binary relation $\overset{t}{\to} \subset Q \times Q$ and it is true that $(l, x) \overset{t}{\to} (l', x')$ if and only if (a) $l = l'$, (b) $x' = x$ for $t = 0$ and (c) $x' = \varphi_t^l(x, u)$ for $t > 0$ where for $\tau \in [0, t]$, $\dot{\varphi}_\tau^l(x, u) \in f_l(\varphi_\tau^l(x, u))$ and (d) for all $\tau \in [0, t]$, $x(\tau) \in i_l$. Accordingly, a t-step is a time trajectory of the system that is valid for $\tau \in [0, t]$.

B. *Interacting Controlled Hybrid Automata*

In order to cope with the 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, $\{\mathcal{H}^j\}$. Each hybrid automaton in the set is a tuple as before:

$$\mathcal{H}^j = \left\{ Q^j, \Sigma^j, U^j, Y^j, F^j, H^j, I^j, E^j, G^j, R^j \right\} \qquad 1$$

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 $l \in L^j, I^j(l) \subseteq X^j \times \prod_k Y^k$, where *k=1…j-1, j+1…n.*

Guard Conditions: For each $e \in E^j$, $G^j(e) = g_{e^j} \subseteq X^j \times \prod_k Y^k$, where *k=1…j-1, j+1…n.*

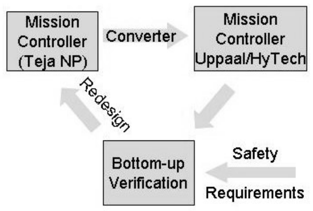
All other components of the tuple are analogous to those of the single hybrid automaton defined above.

Event Synchronization: For an event $\sigma \in \Sigma = \bigcup_j \Sigma^j$, let $In(\sigma) = \left\{ j \mid \sigma \in \Sigma^j \right\}$ be the set of indices of the event sets that contain the event σ. Then each σ-step must be taken synchronously by each of the hybrid automata $\mathcal{H}^j$ if $j \in In(\sigma)$, the corresponding guard condition g_{e^j} is satisfied, and the invariant condition $I^j(l)$ of the accepting state is satisfied. In other words, for each $j \in In(\sigma)$, $(l_1^j, x_1^j) \overset{\sigma}{\to} (l_2^j, x_2^j)$ if and only if (a) $e^j = (l_1^j, \sigma, l_2^j) \in E^j$ (b) $x_1^j \in g_{e^j} \cap i_{l^j}$ and (c) $x_2^j \in r^j{}_{e^j}(x_1^j) \cap i^j{}_{l_2^j}$.

C. *Formal Verification*

While mission control architectures for AUVs have been deployed successfully [1]-[3], none of them were designed for semi-automatic verification in all phases of controller development using modern verification tools and techniques without considerable overhead. Formal verification of a set of requirements is achieved through reachability analysis, forward or backward depending on the specification. For a given region *W* in the hybrid state space, the region forward reachable from *W* is defined as the set of all states reachable from *W* after a finite number of steps with a similar definition for backward reachability. While reachability is difficult to prove for general hybrid automata, tools have been developed for automatic model checking for linear hybrid automata whose continuous dynamics are governed by rectangular constraints on the variables and their derivatives [8],[1]. Properties that can be checked by these tools include safety, liveness, time-boundedness, and duration requirements. Following [6], we see that in practice, safety and other analyses can be posed as reachability problems. Often, this involves the creation of specialized monitor processes that are composed with the system to be analyzed and "watch" the system. These monitor processes enter a violation state if the main system violates a specified safety requirement. It turns out that all timed safety requirements, including bounded-time response requirements, can be verified in this manner. While there is no guarantee of termination of the reachability analysis, it has been found that, in practice, most analyses do terminate [7]. A major strength of HyTech is its ability to perform parametric analyses where the ranges of values of specific parameters necessary for achievement of a safety or other specification may be determined analytically.

We partition the verification problem into *safety verification* – the verification that the logic of the mission controller will not allow anything “bad”, such as deadlocks or livelocks, to happen; and *progress verification* where the progress of the mission controller towards achieving its mission are checked both from a logical standpoint as well as an algorithmic standpoint. A bottom-up approach is employed where the controllers at the lowest level of the hierarchy are verified first, and assuming their correctness, the verification of the controllers in the next higher layer of the hierarchy is performed. During verification of a particular subsystem, an abstracted subsystem, called a *driver* subsystem, may be created to emulate only the relevant commands issued by either a higher level or lateral subsystem. Similarly, an abstracted subsystem, called a *stub* subsystem, may be created to emulate relevant responses issued by either lower level or lateral subsystems. *Driver* and *stub* subsystems reduce the complexity of verification by reducing the number of discrete states and clocks in a composed system. Subsystems whose internal states, guard conditions, or update laws affect the subsystem being verified should not be abstracted.


Mission Controller (Teja NP)
Converter
Mission Controller Uppaal/HyTech
Redesign
Bottom-up Verification
Safety Requirements
Figure 3: Semi-automatic verification

The tool used to implement the architecture, Teja NP [4], is a general purpose hybrid systems design and deployment tool. The structure of the model described in this section is imposed on all modules that make up the mission controller and that are designed using Teja NP. In addition to formalizing the interacting hybrid automata models developed using Teja, this has the further advantage of ensuring that a specific mission controller implementation may be formally verified against a set of requirements using tools that conform to this modeling formalism. A hybrid state machine is represented in Teja NP using an UML-like language. A converter has been developed to convert the hybrid state machine representation used by Teja NP into the specification that UPPAAL and HyTech use to represent linear hybrid systems. This allows the Teja models to be semi-automatically imported into UPPAAL and HyTech, where subsets may be collected and used for formal verification of safety and performance. This process is summarized in Figure 3. While there are several pending research issues in the application of analytical verification, including the question of how to determine and represent the safety specifications to be checked, this method shows some immediate promise and has already been used to answer some basic safety questions. The details of the hybrid state machine converter, the conversion process, the procedures for verification, and the results of verification studies are omitted here due to space considerations, but are the subject of a follow-on paper.

V. Conclusion

A hybrid, model-based architecture for mission control of AUVs is presented. The unique distinguishing feature of this architecture is that it supports automated code-generation, real-time operations, and formal verification. Most past architectures lack these essential features. A generic survey AUV mission controller conforming to this architecture, and developed using a commercial hybrid systems modeling and implementation tool, is presented. A modeling formalism, which allows formal verification methods and tools to be applied, is presented for the interacting hybrid automata that make up the mission controller. A tool to convert hybrid automata representations from the design platform to two verification tools is introduced, and the procedure to go from design to verification to re-design is outlined.

References

[1] Kumar, R., and Stover, J.A., “A Behavior-Based Intelligent Control Architecture with Application to Coordination of Multiple Underwater Vehicles,” *IEEE transactions on systems, man, and cybernetics—part a: systems and humans*, vol. 30, no. 6, November 2000.

[2] McPhail S.D., Pebody M., 1998, Autosub-1: Mission programming and control of an Autonomous Underwater Vehicle, *Proc. Third European Marine Science and technology Conference*, May 1998, Lisbon, Portugal.

[3] Bellingham, J.G., Consi, T.R., Beaton, R., and Hall, W., Keeping Layered Control Simple, In *Proceedings AUV ’90*, 1990.

[4] www.teja.com

[5] www.uppaal.com

[6] Alur, R., Courcoubetis, C., and Dill, D., “Model Checking for Real-Time Systems”, *Proceedings of Logic in Computer Science*, pp. 414-425, 1990.

[7] Alur, R., Henzinger, T.A., Lafferriere, G., and Pappas, G.J., *Discrete Abstractions of Hybrid Systems*. Proceedings of the IEEE, 88, July 2000.

[8] Henzinger, T.A., Ho, P-H., and Wong-Toi, H., “A user guide to HyTech,” *Proceedings of the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '95), Lecture Notes in Computer Science 1019, Springer-Verlag, 41-71, 1995.

[9] Larsen. K.G., and Pettersson, P., “Timed and Hybrid Systems in UPPAAL2k”, *MOVEP'2k : Modeling and Verification of Parallel Processes*, Nantes, France, 2000.