Proceedings of the 2006 American Control Conference
Minneapolis, Minnesota, USA, June 14-16, 2006

A Bottom-Up Approach to Verification of Hybrid Model-Based Hierarchical Controllers with application to Underwater Vehicles

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

Abstract — **We present a systematic method of verification for a hierarchical hybrid system which is developed using a bottom-up approach. The bottom level of the hybrid system hierarchy is verified first, and each higher-level is subsequently verified with the assumption that all lower levels are correct. At each step in the verification process, lower and higher levels than the one currently being verified may be abstracted, thus reducing the complexity of verification. This method is algorithmically developed and integrated into the design of a hierarchical hybrid mission-level controller for an autonomous underwater vehicle.**

I. Introduction

Hybrid systems, those containing both continuous dynamics and discrete transitions, have become the focus of much research in the areas of control and computer science because of their wide range of practical use, which includes automated highway systems, high-level embedded controllers, manufacturing process control, robotics, air traffic management systems, and communication network synthesis. In each of these areas, much emphasis must be placed on safe, reliable and correct operation. Informally, safe, reliable and correct operation requires that the system, during all times of operation, will *never* perform any unsafe tasks and will eventually complete a desired task. For the case of an Autonomous Underwater Vehicle (AUV), a simple example of safe and correct operation requires that the vehicle never exceeds a certain depth and eventually completes the mission tasks. An AUV, like many autonomous systems, contains multiple levels of control that must each satisfy a set of requirements in order to guarantee correctness of the overall system. In this paper, we present a methodology for the verification of hierarchical hybrid systems that is integrated with the design process and, specifically, verification of a hierarchical AUV mission controller, where the verification specifications (at this level) are derived from high-level specifications.

Manuscript received March 23, 2006. This work was supported in part by the NSF under grants NSF-ECS-0218207, NSF-ECS-0244732, NSF-EPNES-0323379, and NSF-ECS-0424048, by ONR under a DoD-EPSCoR grant N000140110621, and by an Exploratory and Foundational Research grant from ARL.

M. O'Connor (phone: 814-863-8418; email: mjo151@psu.edu) and S. Tangirala (phone: 814-863-7594; email: cxt148@psu.edu) are with the Applied Research Laboratory at Penn State University.

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

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

M. Sznaier is with Penn State Univ., Dept. of Electrical Engineering.

High-level control of AUV's, such as mission control, is often more abstract and includes additional requirements such as re-configurability, learning, safety, failure tolerance, the ability to manage dynamically changing mission goals, and increased autonomy. In order to cope with such complexity, mission control is often hierarchically decomposed, and thus a hierarchical method of hybrid system design, in which each layer of the hierarchy is responsible for either executing or coordinating a set of tasks, may be used [1]. In order to deal with the complexity of verifying a hierarchical hybrid system, we present a *bottom-up* approach to verification, where subsystems on all levels, other than the level currently being verified, may be abstracted by removing all irrelevant details. Our approach to hierarchical modeling and verification is systematic and can easily be applied to a wide range of hybrid systems.

In Section II, we briefly revisit the hybrid mission controller architecture for AUVs presented in [1]. Section IV presents a rigorous hybrid system model which formalizes the mission controller. Section A discusses our approach to hybrid system verification, specifically describing a bottom-up approach and a high-level verification algorithm. An example of verification is also presented.

II. Hierarchical hybrid mission controller

A mission control architecture for AUVs that has been designed to include verification of a set of requirements in every design phase is presented in [1]. A specific application of this architecture to a generic *survey* AUV is shown in . 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. The survey AUV control architecture of is organized hierarchically and is composed of various modules, where each module is a hybrid system, and the entire architecture is modeled as a set of interacting hybrid systems. At the lowest level of the hierarchy is the underwater vehicle (plant) and the vehicle controllers (VCs), which have a hybrid state space and, together, serve as the plant for the higher level mission controller (MC). The VC and MC communicate through the interface layer shown in . The mission controller consists of a collection of high-level hybrid automata that communicate via shared data and synchronization events. As shown in , the MC, formally $\{\mathcal{H}_i^j\}$, is comprised of a number of subsystems that operate at various levels, where i denotes the level and j denotes the subsystem on level i. Each subsystem consists of a hybrid automaton that is responsible for completing a predetermined set of tasks. The MC is hierarchically decomposed into *Behavior Controllers*, *Operation Controllers*, and at the highest level, the *Mission Coordinators*. A mission is defined as a coordinated sequence of operations, each of which is a sequence of behaviors, which are in turn, sequences of vehicle commands.

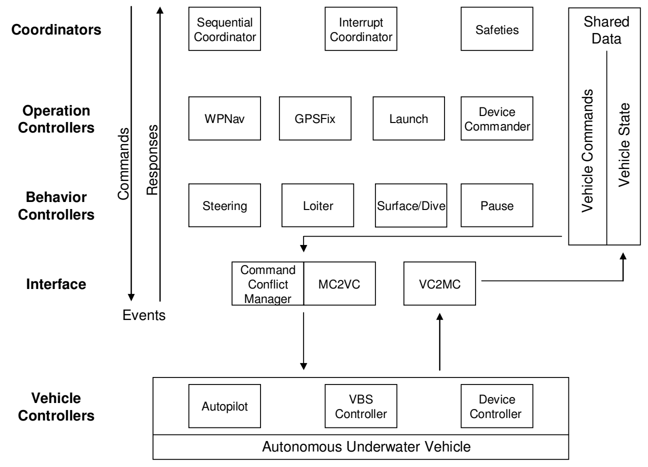

Figure 1: Hybrid Survey AUV Control Architecture

III. Hybrid System Model

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

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

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. $u:[0,\infty)\to U$ denotes 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, also denoted $F(l,u)=f_l(\cdot,u)$.

Output Functions: H is a set of output functions, one for each location $l\in L$. The notation $H(l)=h_l$ is used, where $h_l : X\times U\to Y$ 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 $I(l)=i_l\subseteq X$ may also be used. If no i_l is specified for some $l\in L$, then $i_l = X$.

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 $G_e = g_e\subseteq X$ may also be used. If no g_e is explicitly specified for some edge $e\in E$, then the default value is taken to be $g_e = X$.

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 taken to be 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 between discrete states and is also know as a discrete jump. A σ-step need not be taken even if $x\in g_e$, but some σ-step must be taken before $x\notin i_l$.

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 $\tau\in[0,t]$.

Definition - trajectory: A trajectory π of $\mathcal{H}$ is a finite or infinite sequence $\pi : q_0\overset{\theta_0}{\to}q_1\overset{\theta_1}{\to}\ldots q_{i-1}\overset{\theta_{i-1}}{\to}q_i\ldots$ where $q_i\in Q$ and $\theta_i\in\Sigma\cup\Re^+$. A trajectory is accepted by $\mathcal{H}$ if each $q_i\overset{\theta_i}{\to}q_{i+1}$ is a t -step or σ -step of $\mathcal{H}$, and we denote the space of all such trajectories by $\mathcal{H}$. A step of a trajectory refers to a t-step followed by a σ -step. Associated with the kth step of a trajectory is (a) the time interval of the step, $I^0=[0,t^1]$ or $I^k=[t^k,t^{k+1}]$ for $k\ge 1$, (b) its duration, $\tau^k = t^{k+1}-t^k$, (c) the associated edge, $e^k=(l^k,\sigma^k,l^{k+1})$, and (d) the state, $q^k=(l^k,x^k(t))$, where l^k is fixed over I^k and $x^k(t)$ satisfies $\dot x^k(t)=F_l^k(x^k(t),u(t))$. Thus, the step can be represented as $(l^k,x^k(t^k+))\overset{\tau^k}{\to}(l^k,x^k(t^{k+1}+))\overset{\sigma^k}{\to}(l^{k+1},x^{k+1}(t^{k+1}+))$ satisfying $x^k(t^{k+1})\in g_{e^k}$ and $x^k(t^{k+1}+)\in\tau_{e^k}(x^k(t^{k+1}))$.

Note that we do not exclude the possibility that $\tau^k=0$, in which case there is only a σ-step.

Definition - run: A run of a hybrid automaton $\mathcal{H}$ is the projection to the discrete part of a trajectory in $\mathcal{H}$; namely, a finite or infinite sequence $l^0, l^1, l^2, ...$ of admissible locations. We also refer to $x(t) = \sum_{k=0}^{\infty} x^k(t) \mathrm{II}_{I^k}(t)$ where $\mathrm{II}_{I^k}(t)$ is the indicator function of the interval I^k, as the continuous part of the trajectory. Note that it is not in general true that $x(t^k+) = x^k(t^k+)$. For instance, if $\tau^k = 0$ and $\tau^{k+1} > 0$ then $x(t^k+) = x^{k+1}(t^k+)$ which need not be $= x^k(t^k+)$.

A. 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, $\{\mathcal{H}^j\}$. 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 $j \in In(\sigma)$, $(l_1^j, x_1^j) \xrightarrow{\sigma} (l_2^j, x_2^j)$ 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) $x_2^j \in r^j{}_{e^j}(x_1^j) \cap i^j{}_{l_2^j}$.

IV. Hybrid system verification

A. Verification Techniques

Since hybrid systems are prevalent in a variety of real-world applications, verification techniques for such systems have been extensively researched and developed. In general, three methods of hybrid system verification are available: simulation, model checking, and theorem proving. No single method is perfect, as simulation can never exhaustively test every possible path in the system, model checking may not be decidable for certain classes of hybrid systems, and theorem proving is often too complex for reasonably sized systems [4]. When verifying real-time systems, simulation and model checking are used more prominently, as both methods are made available through computational tools. In this paper, we present a methodology for the verification of hierarchical hybrid systems that is tightly coupled with the design process and uses the automated model checking tool Uppaal [5]. Uppaal was chosen due to its compatibility with the modeling formalism, GUI, ease of use, and portability.

B. Development Tool vs. Verification Tool

The survey AUV hybrid mission controller has been designed and implemented in Teja NP. Teja NP [3] is a graphical hybrid system design tool that contains built-in support for automatic code generation. Following a hybrid system description, Teja facilitates communication between hybrid subsystems via shared data and event synchronization. Each Teja system must contain a user-defined event dependency table that specifies which subsystems may receive events that are sent from another subsystem. When a Teja subsystem initiates an event, it is passed to all subsystems listed within the event dependency table, causing synchronization.

Teja, however, does not contain functionality for formal verification; thus an external tool such as Uppaal must be used for verification. In order to facilitate rapid (re)design and verification, a converter was created that converts a hybrid (timed) autonomous system description in Teja to an Uppaal system description. The details of this converter are omitted here due to space restrictions.

Although Teja and Uppaal both support timed autonomous systems, there are several differences in the tools that must not be overlooked. As previously mentioned, event synchronization in Teja occurs according to an event dependency table; thus, events can only be sent to subsystems listed in the event dependency table, and any number of subsystems, if enabled, can synchronize on any given event. Uppaal, however, does not contain an event dependency table. Two, and only two, Uppaal subsystems may synchronize on two enabled edges over a normal channel if one edge is *commanding* and one edge is *accepting*. *Any one* Uppaal subsystem with an enabled edge may synchronize with the commanding subsystem, and if no synchronizing edge is available, no transition will take place; whereas in Teja, the transition will take place in the commanding subsystem regardless of how many systems, including zero, are synchronizing on the event.

To overcome this problem, all channels in Uppaal must be declared as broadcast channels. Zero, one, or multiple Uppaal subsystems may synchronize on a single event over a broadcast channel. We are however restricted in that a certain subsystem, not listed to receive an event in the Teja event dependency table, may still synchronize on that event in Uppaal. This restriction must be overcome by examining the Teja event dependency table during Uppaal verification.

C. Bottom-up Approach

In order to deal with the complexity of verifying multiple levels in a hierarchical hybrid system, we propose a bottom-up method of hybrid verification, in which the bottom-most subsystems are verified first, the subsequent higher level is verified next, assuming the bottom level has been correctly verified, and this process is continued until all levels have been properly verified. Using this approach, the verification process is simplified in the following ways: (1) subsystems on lower levels, once verified, may be abstracted by removing all irrelevant details; (2) subsystems on higher levels, before being verified, may be abstracted by removing all intrinsic details, as well as, all states not relevant to the subsystem currently being verified; (3) changes to subsystems arising from *their* verification do not necessitate re-verification of other subsystems.

In a hierarchical hybrid system, subsystems may synchronize with other subsystems on either higher, lower, or the same level (lateral subsystems). During verification of a particular subsystem, a conservatively 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, a conservatively 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 serve the purpose of simplifying 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.

D. Hierarchical Verification Algorithm

The first step of the verification process involves determining a set of requirements that the system must satisfy. This step can often prove to be very difficult and time-consuming and is currently the subject of further research. Examples of high-level requirements for a specific hierarchical hybrid system are provided in Section I.A.

Once the high-level requirements have been identified, the verification process for a hierarchical interacting hybrid system

$$\mathcal{H}^j_i = \left\{Q_i^j, \Sigma_i^j, U_i^j, Y_i^j, F_i^j, H_i^j, I_i^j, E_i^j, G_i^j, R_i^j\right\} \qquad (2)$$

which synchronize as $\mathcal{H} = \mathcal{H}_1^1 \parallel \mathcal{H}_1^2 \ldots \mathcal{H}_1^{m_1} \ldots \mathcal{H}_n^{m_n}$, where $i=1\ldots n$ (number of levels) and $j=1\ldots m_i$ (number of interacting hybrid automata on level *i*), may be performed using the following algorithm, which has been derived from the bottom-up approach. Initially, model each subsystem as an interacting hybrid system as given by eq. (2), where $i=1$ corresponds to the lowest level and $j=1$ corresponds to the first subsystem on each level. Order the subsystems on each level according to their *lateral* dependence: $\mathcal{H}_i^{j_2}$ commands $\mathcal{H}_i^{j_1} \Rightarrow j_2 > j_1$.

```
For $i = 1$ to $n$
    For $j = 1$ to $m_i$
        - Select subsystem $\mathcal{H}_i^j$ for verification
        - Find all subsystems $\mathcal{H}_k^l$, $k=1$ to $n$, $l=1$ to $m_k$,
          $(k,l) \neq (i,j)$ that interact with subsystem $\mathcal{H}_i^j$
        - Abstract all subsystems $\mathcal{H}_k^l$, for $k,l$ as found above,
          whose internal states are not relevant to verification,
          as drivers or stubs, and replace the original
          subsystems with the abstracted subsystems. Denote
          the abstracted subsystems by $\mathcal{H}'^l_k$
        - Compose the system as $\mathcal{H} = \mathcal{H}_i^j \parallel \mathcal{H}'^l_k$ for $k,l$ as found
          above
        - Formulate queries using temporal logic formulas
          based on the requirements of the system and check
          queries on the composed system $\mathcal{H}'$
        - Correct any problems with subsystem $\mathcal{H}_i^j$
    Next $j$
Next $i$
```

E. Hybrid System Abstractions

A hybrid system may be abstracted in two ways [2]: the discrete behavior of the system may be abstracted or the continuous behavior of the system may be abstracted. In the context of logical verification, the survey AUV mission controller does not depend on the continuous dynamics of the underwater vehicle. At this level of verification, the continuous dynamics are ignored, except in the case of real-valued clocks. A similar approach is taken in [6].

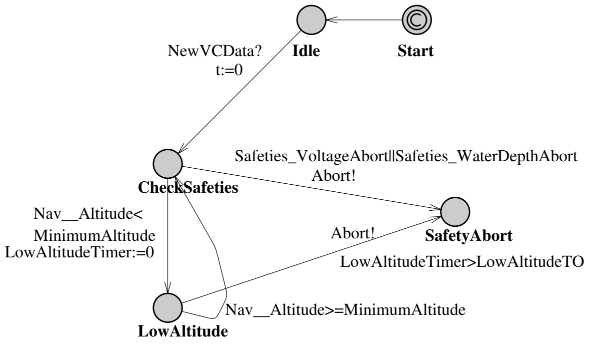

Figure 2 - Safeties subsystem

A hybrid system may also be discretely abstracted. An example of discretely abstracting a hybrid automaton is now presented. The *Safeties* mission coordinator, shown in Figure 2, resides at the top level of the AUV mission control architecture and maintains safe operation of the vehicle at all times. If an unsafe condition is detected, Safeties may abort the mission by aborting operation of all subsystems. Thus, if a safety abort occurs, all subsystems must properly respond and abort operation. An abstracted version of the safeties subsystem, called the *safeties driver*, was created for verification and is shown in Figure 3(a). The safeties driver must be included in the verification of every subsystem in the mission controller.

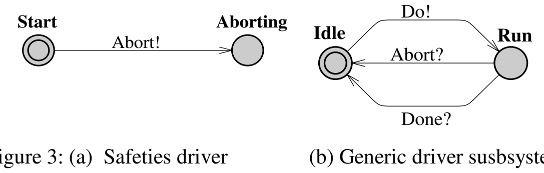

Figure 3: (a) Safeties driver (b) Generic driver susbsystem

All other drivers generally take the form shown in Figure 3(b), where the ! denotes a commanding transition and the ? denotes an accepting transition.

F. *Mission Controller Subsystem Requirements*

As previously mentioned, the first step in the verification process involves identifying a set of system requirements. For the case of the AUV mission controller presented in Section I.A, a set of requirements based on high-level specifications has been derived for each level within the hierarchy. The behavior and operation controllers, which are responsible for executing tasks, share the same requirements that are listed below.

(1-b) The composed system must *never* be deadlocked
(2-b) The subsystem, when in any state, must properly respond to an abort command
(3-b) The subsystem must not improperly abort commanding subsystems
(4-b) The subsystem must properly respond to a command from a higher level or lateral subsystem
(5-b) The subsystem must properly issue commands to other subsystems, if necessary
(6-b) States in which outgoing transitions rely on the navigational or functional state of the vehicle must contain timeout conditions

An example of verifying requirement (2-b), which may be used for every behavior or operation subsystem, is illustrated using the Uppaal query shown below.

E<> SafetyDriver.Aborting and not Subsystem.Idle

a. Does a path exist where the safety driver has issued an abort command but the subsystem has not correctly responded by transitioning to the Idle state?
b. If the query is satisfied, the subsystem contains a path that does not correctly respond to an abort command; otherwise, the subsystem correctly responds to an abort command throughout all paths in the system.

As with requirement (2-b), generic Uppaal queries have been formulated to check requirements (1-b), (3-b), and (4-b) on every behavior and operation subsystem; however, requirements (5-b) and (6-b) necessitate a more rigorous inspection of each individual subsystem. Several cases are examined in Example 1 below.

The mission coordinators are responsible for coordinating tasks (rather than executing tasks), and thus share a different set of requirements, which are listed below.

(1-c) The composed system must *never* be deadlocked
(2-c) Each coordinator must always properly respond to an abort command.
(3-c) Each coordinator must properly respond to a done event from a lower-level subsystem.
(4-c) Each coordinator must properly issue commands to lower-level subsystems.
(5-c) Interaction among coordinators must *always* occur correctly. (e.g.) The Interrupt coordinator must properly suspend the Sequential coordinator when necessary.
(6-c) Coordinators only issue commands when appropriate. (e.g.) The Interrupt coordinator must not start a timed order before the order is scheduled to occur.

The requirements listed above must be examined with all coordinators in the composed system. An example of checking requirement (2-c) is illustrated using the Uppaal query shown below.

E<> Coordinator1.End and not Coordinator2.End

a. Does a path exist where Coordinator1 is in the End state but Coordinator2 is not?
b. If the query is satisfied, Coordinator2 may not properly end execution when Coordinator1 ends execution; otherwise, Coordinator2 properly transitions to the End state when Coordinator1 transitions to the End state.
c. This query must also be verified in the reverse case.

G. *Example: Operation level verification*

Applying the algorithm listed in Section IV.D to the survey AUV, verification begins at the behavior level, proceeds to the operation level, then finally to the coordinator level. In the following verification example, the GPSFix subsystem, shown in Figure 4 and denoted by $\mathcal{H}_2^2$, is verified using Uppaal.

Following the algorithm for verifying a hierarchical hybrid system, the behavior controllers, which, in this case, have already been verified, are replaced by *stub* subsystems, as is illustrated for the Steering subsystem in Figure 5a. Likewise, a *driver* subsystem has been created to imitate the synchronization that normally occurs between the GPSFix subsystem and a coordinator level subsystem, as shown in Figure 5b.

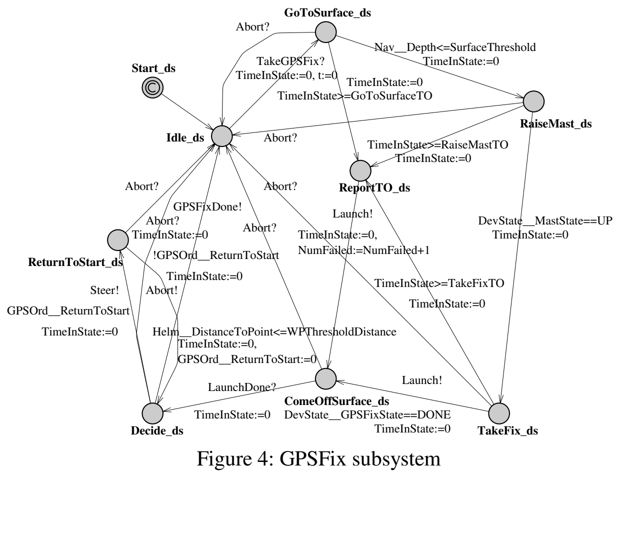

Figure 4: GPSFix subsystem

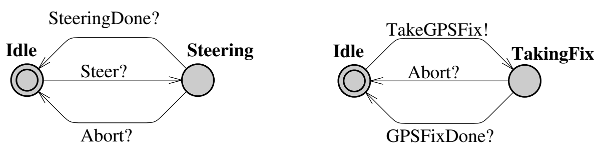

Figure 5: (a) Steering stub subsystem   (b) GPSFix driver subsystem

The Launcher subsystem (not shown) is an operation controller that is commanded by the GPSFixer. Since the GPSFixer (laterally) depends on the Launcher, the Launcher was verified first and replaced by an abstracted stub subsystem and included in the verification of the GPSFix subsystem. Also included in the integrated system is the abstracted Safeties driver shown in Figure 3(a).The GPSFix subsystem, GPSFix driver subsystem, Steering stub subsystem, Launcher stub subsystem, and Safeties driver subsystem, which interact according to (2), are synchronously composed in Uppaal, and a set of temporal logic queries are formulated based on requirements (1-b) through (6-b), a few of which are listed below. Note that the subsystem requirements are transformed into temporal logic queries that specify an erroneous set of states, and Uppaal is used to check whether this set is reachable. If the set is reachable, Uppaal generates a diagnostic trace that is used to identify and correct the problem.

A[] not deadlock

- Requirement (1-b)
- For all paths, is the system not deadlocked?
- The query is satisfied signifying no immediate deadlocks.

E<> SafetyDriver.Aborting and not GPSFixer.Idle

- Requirement (2-b)
- Does a path exist where a safety abort has occurred but the GPSFix subsystem does not properly abort?
- The query is satisfied suggesting that the GPSFixer may not properly abort under certain conditions.
- **The ReportTO state is missing an abort response transition to the Idle state, which must be added.**

E<> GPSdriver.Idle and not GPSFixer.Idle

- Requirement (3-b)
- Does a path exist where the GPSFix driver is in the Idle state but GPSFix is not?
- The query is satisfied indicating that the GPSFixer may improperly abort the GPSFix driver.
- **The abort output event on the transition from ReturnToStart to Decide should be changed to a SteeringDone output event.**

E<> GPSdriver.TakingFix and GPSFixer.Idle

- Requirement (4-b)
- Does a path exist where the GPSFixer does not properly respond to a TakeGPSFix command?
- The query is not satisfied, signifying proper GPSFix subsystem response.

Once verification of the GPSFix subsystem is complete, it can be replaced with a stub system, as shown in Figure 6, when subsequently verifying the top level of the mission controller hierarchy. Notice that, in Figure 6, the GPSFix stub subsystem contains more than two states. Since the GPSFix subsystem commands other subsystems (in this case the *Launcher* subsystem and *Steering* subsystem), *do*/*done* transitions that command/respond to the other subsystems must be included in the abstracted subsystem.

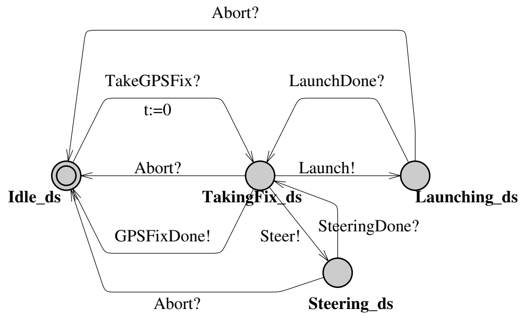

Figure 6: GPSFix stub subsystem

V. Conclusion

In a complicated hierarchically structured hybrid system, the verification process may be greatly simplified by using a bottom-up approach and employing the algorithm presented here. This approach is tractable and efficient, as it significantly reduces the complexity of verification for two distinct reasons. First, the verification process is divided according to the number of levels in the system, verification occurs at each level, and the overall system is guaranteed to be correct when all levels have been verified. Second, when verifying a particular level, all subsystems on other levels may be abstracted thus reducing the number of states in the system and, consequently, the complexity and computational requirements of verification. We also present a high-level verification algorithm for a hierarchal hybrid system. This algorithm, along with the state machine description converter which allows Teja NP hybrid systems to be semi-automatically imported into Uppaal, results in a tightly coupled design/verification process resulting in controllers which are guaranteed to satisfy a set of requirements. Illustrative examples of verification are provided.

References

[1] Tangirala, S., Kumar, R., Bhattacharyya, S., O'Connor, M., and Holloway, L.E., "Hybrid-Model based Hierarchical Mission Control Architecture for Autonomous Underwater Vehicles", *Proceedings from the American Control Conference*, 2005.
[2] Alur, R., Henzinger, T., Lafferriere, G., and Pappas, G., "Discrete Abstractions of Hybrid Systems", *Proceeding of the IEEE*, v 88, n 7, July 2000.
[3] www.teja.com
[4] Greenstreet, M., "Pragmatic Verification for Hybrid and Real-time Designs", *Proceedings of the American Control Conference*, Chicago, Illinois, June 2000.
[5] www.uppaal.com
[6] Puri, A. and Varaiya, P., "Modeling and Verification of Hybrid Systems", *Proceedings from the American Control Conference*, Seattle, Washington, June 1996.