REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
28-02-07	Final Technical Report	01-04-01 to 31-03-05

4. TITLE AND SUBTITLE
DEPSCOR: Research on ARL's Intelligent Control Architecture

5a. CONTRACT NUMBER
N00014-01-1-0621

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Holloway, Lawrence E.
Kumar, Ratnesh

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Kentucky Research Foundation
DUNS: 939017877
Office of Grants and Contracts
Lexington, KY 40506

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research Regional Office Chicago
230 South Dearborn, Room 380
Chicago, IL 60604-1595

10. SPONSOR/MONITOR'S ACRONYM(S)
ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this research, our goal has been to develop hierarchical hybrid mission control architecture for autonomous systems illustrating its application to autonomous underwater vehicle (AUV), verify the logical correctness of the controller designed, look into the feasibility of simulating the operations executed by the AUV, and automate controller synthesis. The correct operation of a system we design is a requirement. The challenge to develop a hierarchical hybrid mission controller for underwater vehicle which facilitates modeling, verification, simulation and automated synthesis of coordinators has lead to research in this area. We have worked and are working on these issues with Applied Research Laboratory (ARL) at Pennsylvania State University (PSU) who have designed autonomous underwater vehicles for over 50 years primarily under the support of the U.S. Navy through the Office of Naval Research (ONR).

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Larry Holloway
U	U	U	UU		19b. TELEPHONE NUMBER *(Include area code)* 859-257-6262 ext203

DEPSCOR: RESEARCH ON ARL's INTELLIGENT CONTROL ARCHITECTURE:

HIERARCHICAL HYBRID-MODEL BASED DESIGN, VERIFICATION, SIMULATION, AND SYNTHESIS OF MISSION CONTROL FOR AUTONOMOUS UNDERWATER VEHICLES

Final Technical Report

ONR Project Award Number
N00014-01-1-0621

University of Kentucky

Principal Investigators:

Ratnesh Kumar (Iowa State University)
and
Lawrence E. Holloway (University of Kentucky)

TABLE OF CONTENTS

HIERARCHICAL HYBRID-MODEL BASED DESIGN, VERIFICATION, SIMULATION, AND SYNTHESIS OF MISSION CONTROL FOR AUTONOMOUS UNDERWATER VEHICLES

Final Technical Report

ONR Project Award Number
N00014-01-1-0621

University of Kentucky

Principal Investigators:
R. Kumar (Iowa State University)
and
L. Holloway

TABLE OF CONTENTS

HIERARCHICAL HYBRID-MODEL BASED DESIGN, VERIFICATION, SIMULATION, AND SYNTHESIS OF MISSION CONTROL FOR AUTONOMOUS UNDERWATER VEHICLES

Principle Investigators: Ratnesh Kumar and Lawrence E. Holloway

1. Students involved

The following students were involved in research related to above ONR funded project:

1. Siddhartha Bhattacharyya, Received PhD from Univ. of KY, Currently Assistant Professor at Kentucky State University.

2. Matt O'Connor, Received MS from Penn State Univ., Employed by ARL-PSU after graduation, presently working for a Controls Company in Pittsburgh. (Matt contributed to project but funded separately by ARL.)

3. Zhen Yu, Received MS from Iowa State Univ., Currently pursuing PhD at Iowa State Univ.

The research also benefited from collaboration with the Applied Research Laboratory (ARL) at the Pennsylvania State University. The work involved developing a hierarchical control architecture, and modeling, design, simulation/animation, verification, and implementation of a mission-planner and controller within this architecture. The mission controller has been implemented by ARL researchers (under the leadership of Drs. Sekhar Tangirala and John Dzielski) on two of its AUVs. The results of the research have been reported in a series of papers as indicated in section 8.

The work accomplished under the above grant is briefly discussed in the following various subsections.

2. Introduction

In this research, our goal has been to develop hierarchical hybrid mission control architecture for autonomous systems illustrating its application to autonomous underwater vehicle (AUV), verify the logical correctness of the controller designed, look into the feasibility of simulating the operations executed by the AUV, and automate controller synthesis. The correct operation of a system we design is a requirement. The challenge to develop a hierarchical hybrid mission controller for underwater vehicle which facilitates modeling, verification, simulation and automated synthesis of coordinators has lead to research in this area. We have worked and are working on these issues with Applied Research Laboratory (ARL) at Pennsylvania State University (PSU) who have designed autonomous underwater vehicles for over 50 years primarily under the support of the U.S. Navy through the Office of Naval Research (ONR).

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. (Discussion on hybrid systems is in the next section.) Design and verification of hybrid systems is highly challenging task, owing to the sophistication and complexity of design and verification. To simplify the complexity of design, researchers at ARL worked with us to formulate a hierarchical control architecture upon which the mission controller design is based. Similar hierarchical architectures built earlier didn't facilitate the development of a model which can be easily put to verification. Our hierarchical hybrid mission control architecture not only facilitates the design of a complex mission controller, it also facilitates verification (done hierarchically in a bottom-up fashion), simulation and also the automated synthesis of the highest level mission coordinators.

Our method can be used for other autonomous systems. 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-maneuver 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. We have illustrated our approach through a specific example of mission, namely, a search mission. The same philosophy can be utilized in designing mission controller for other types of missions, such as surveillance and attack. So although the behavior/operation/coordinator controllers that will be designed will vary from mission type to mission type, the approach of the whole of the mission-controller remains the same for all autonomous vehicles for all missions. The generalized approach to automate the synthesis of mission coordinators can be used to any kind of application for any kind of AUV.

Control of autonomous underwater vehicles (AUVs) present specific issues related to automatic control but also classic concerns of real time and high level programming. Several approaches to the design of controllers for AUVs have missed the classical concerns which are a necessity because of the environment in which the AUV needs to operate. Control systems for AUVs have several communicating subsystems/modules. These subsystems/modules need to interact among themselves to successfully execute a mission within satisfactory real time bounds. They constantly react with the environment so they must react in real time to sensory information, thus the need for considering classical issues. Our hierarchical hybrid mission control architecture takes into consideration the real time and high level programming concerns as well. Each of the subsystems developed is a hybrid system which takes care of real time issues by including continuous variables which implement real time or time bounded constraints. The subsystems have been implemented using a high level programming environment provided by Teja.

Verification of a mission control architecture developed for an AUV or for any autonomous system has been neglected because of the lack of such a simplified model. Our hierarchical hybrid mission control architecture supports verification with ease because of the simplified hybrid model on which it is based. For formal verification we use a graphic tool called Uppaal as opposed to Esterel. This is because Esterel does formal verification of control laws with a complicated method of verification and requires very careful coding in Esterel whereas Uppaal, a graphical tool, models a hybrid automaton easily and gives diagnostic traces efficiently. The verification method used abstracts the subsystem models. This method of using abstractions might miss upon some situations which can be caught in an environment in which all the coordinators and the controllers work together. Such a combined approach can be developed if we can simulate the working of the hierarchical hybrid mission control architecture as a whole. Thus the simulation built simulated the combined working of the subsystems to execute an operation which involves the execution of a sequence of actions.

Mission coordinators at the highest level of the hierarchical hybrid mission control architecture pass control to the lower level controllers for the execution of mission orders. A general design of the mission coordinators will help in using the same mission coordinators for different kind of applications for AUVs other than that illustrated. So we finally automated the design of the mission coordinators.

In our approach the initial mission controller modules have been developed using TEJA NP networking software platform by the ARL researchers. TEJA supports the design of interacting hybrid controller modules, and offers the autocode generation capability.
For verification purposes, these modules can be converted to modules of the UPPAAL verifier. UPPAAL is hybrid system verification software which can be used to verify the safety and correctness of mission controller. In order to proceed with the verification, the first task is to develop an extended hybrid state machine based model of the mission controller, which we have accomplished (see section 2). Then we formalized the notion of correctness that demonstrated that any mission can be correctly executed by the mission controller. Also, abstract models of the lower level vehicle controllers (and possibly the underwater sea vehicle) were developed. This we did in consultation with ARL researchers. Next, a bottom-up approach to verification of logical correctness was formulated and implemented. Verification can be done for something that is already designed, so we use our hierarchical mission controller architecture for search as the model to verify the correctness of the existing design. We simulated the operations executed by the hierarchical hybrid mission control architecture, and we finally accomplished the task of automating the synthesis of a general mission coordinator. As an example the coordinator synthesis works correctly to synthesize the current coordinators built at ARL, by the experience gained we suggest further modifications in the controller design approach within future work. In section 2 we discuss hierarchical hybrid mission control architecture and the hybrid system model, in section 3 we discuss the approach used for verification, then in section 4 we discuss the simulation of the missions and in section 5 we discuss the automated synthesis of coordinators. In section 6 we conclude with future work.

3. Hierarchical Architecture

The hybrid mission controller is organized hierarchically as shown in Fig 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.

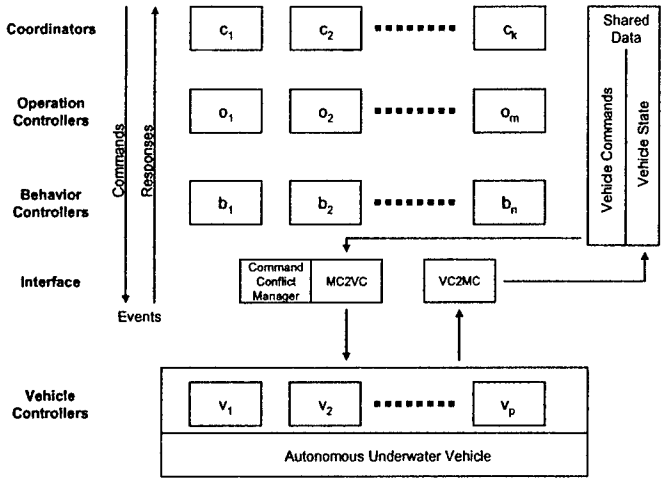

Fig.1. Hybrid Mission Control Architecture

As seen in Fig, 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 three types: Sequential, 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. If an interrupt-driven operation is due and does not require suspension of the currently executing sequential operation, then the two operations occur in parallel. 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. The relative priorities between the coordinators are implemented 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 B denote the set of behaviors, O denote the set of operations, and V denote the set of vehicle subsystem controllers. A mission, m is defined as $m \in M \subset (O+V)^*$, where $(O+V)^*$ is the set of all sequences containing elements of O and V, and M is the set of all possible missions. Similarly, each operation $o_j \in (B+V)^*$, and each behavior $b_k \in V^*$.

a. 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 **Error! Reference source not found.**. 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.

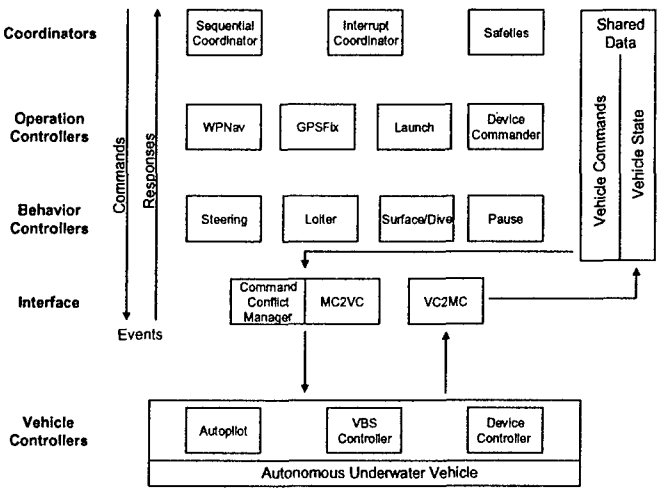

Fig.2. Survey AUV Mission Controller

The lowest level of this mission controller is comprised of four behavior controllers which issue commands to the vehicle controllers and monitor their responses, via the vehicle state vector, to achieve their control objectives. These are described next. The *Steering* behavior controller is responsible for steering the vehicle to a specified location in space and interacts with the Autopilot. This controller contains a model of the earth in 3D and functions that convert between Cartesian and geodetic coordinates. This controller uses navigation data from the vehicle and the destination coordinates to compute a heading command for the AUV which is updated periodically if necessary. Steering may operate in one of two modes: steer-to-point where the objective is to arrive at a point regardless of the path taken; and steer-to-line where the AUV is steered to follow a line between a source location and destination location. In steer-to-point mode, the range and course from the vehicle to the target location are computed periodically and a new heading command equal to the course is assigned to the Autopilot. This algorithm will not steer a line between the source and destination locations in the presence of a current or other external disturbance. Steer-to-line on the other hand computes the equation of a line connecting the source and destination locations and periodically steers

the AUV to ensure it is following this line, even in the presence of external disturbances.

The *Loiter* behavior controller controls the AUV to loiter at a specific location in space for a specified duration and interacts with the Autopilot and VBS Controller. Loiter may be commanded directly by a mission order is some circumstances but is usually used by an operation controller as a means to stay near a specified location while waiting for some event to occur such as external communications, or arrival of a support craft for rendezvous. Loiter may be commanded in one of two modes: *hover*, where the AUV is commanded to hover in the water column at zero forward speed using the variable buoyancy system; and *circle*, where the AUV is steered to a series of locations approximating a circle centered about the specified loiter location. The remaining loiter time is monitored in both modes, so that the AUV is steered in such a way as to arrive at the loiter point when the loiter duration expires. The Loiter behavior controller interacts with the Steering behavior controller, in circle mode, to steer to the points defining the circle; and at the end of both modes to arrive at the final loiter point. This is an example of horizontal communication within a level in the mission controller hierarchy.

The *Surface/Dive* behavior controller commands the vehicle to go to or come-off of the surface and interacts with the Autopilot and the VBS Controller. On a dive command, this controller sequences the variable buoyancy system and the autopilot to bring the AUV off of the surface and to a specified depth while running at a prescribed speed. It also ensures, if necessary, that the AUV is suitably trimmed given the ambient conditions. On a surface command, this controller brings the AUV to the surface and ensures that it is at zero speed and is positively buoyant to prevent it from sinking.

The final behavior controller, *Pause,* is used under certain situations to let the vehicle remain at its current state for a specified duration.

The behavior controllers are, in turn, sequenced and commanded by the operation controllers, which correspond directly to mission orders which are specified by the user. The operation controllers which make up the survey AUV are described next and will demonstrate several examples of horizontal as well as vertical communications between modules in the mission controller hierarchy. 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 Variable Buoyancy System, the Device Commander described later, and the Surface/Dive behavior controller. The Launch operation controller is responsible for determining if the AUV is in a safe region for launch (e.g. enough water depth) and also that it is in a safe state for launch (e.g. all masts retracted) before it commands the Surface/Dive behavior controller to bring the AUV off of the surface. The Launch operation controller uses the Device Commander to command all masts to their retracted positions and to ensure that all required sensors such as the inertial navigation system are in proper states for launch. If Surface/Dive is unable to bring the AUV off of the surface after some predetermined amount of time, the Launch controller issues a mission abort on the assumption that there is a hardware or environmental problem. The launch controller is also responsible for detecting and compensating for surface capture which affects large AUVs. Under certain conditions, it is possible for large AUVs to launch in shallow water but return to the surface due to a pitch up attitude and get captured on the surface. In this situation the variable buoyancy system is used to compensate for the effects of surface capture and allow the AUV to break the surface.

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. Time-outs are built into all states of the GPSFix operation controller to ensure that it is not deadlocked in any state due the inability to complete some action. The GPSFix controller also maintains a log of failed GPS fixes and will disable further GPS fixes if this number crosses some critical value on the assumption of a hardware failure.

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 waypoint order specification includes a list of sensors and devices and their power states. This controller ensures that required/specified devices are turned on using the Device Controller and then uses the Steering behavior controller to steer to the specified waypoint under the specified mode, line or point. This controller is also capable of sequencing behaviors to arrive at the specified location at a specified time. This arrival time is ensured by loitering if necessary and/or by adjusting AUV speed within acceptable limits if necessary. If loitering is required the specified loiter mode is used. This operation concludes when the AUV is within a specified distance of the destination.

The Device Commander is used to control sensors and devices on the AUV in response to mission orders and commands from other operation controllers; this controller interacts with the Device Controller. Besides power states (on/off), various sensor parameters may also be specified when required.

The *Sequential* and *Interrupt-driven* coordinators behave as described in the first portion of this section. The *Safety* coordinator is use to monitor several critical parameters which ensure safety. Violations of safety criteria lead either to modified commands to the vehicle subsystems or in some cases, a mission abort. The safety coordinator monitors the power source on the AUV for remaining energy. If some predetermined (system parameter) minimum energy level is reached, the mission is aborted. It is also possible to switch devices off to conserve energy. The safety coordinator also monitors the height of the water column in which the AUV is operating, if this height falls below a user-specified threshold, the mission will be aborted since the AUV is deemed to be operating in a dangerous environment. The safety coordinator also watches the altitude of the AUV from the bottom. Unless the AUV is commanded to sit on the bottom, the safety coordinator will command a shallower depth command if a user-specified minimum altitude is violated. If the violation continues for a specified amount of time, despite the change in depth commands, the safety coordinator deems the AUV unresponsive to depth commands and issues a mission abort.

The mission controller modules are realized using TEJA NP networking software tool**Error! Reference source not found.**. 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. 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 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. 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.

Error! Reference source not found. shows the hybrid automaton representation of the Launch operation controller modeled using Teja. 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. Continuous state variables, restricted to clocks in the Launcher, are specified and their flows are defined for each discrete state. 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.

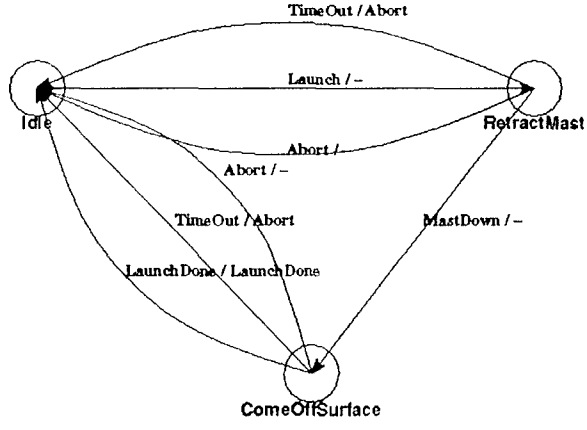

Fig.3.The *Launch* operation controller

On initialization, the sequential coordinator reads in the user specified mission order

file and populates two queues, the sequential order queue, and the interrupt-driven order queue. Mission order syntax is specified using an order specification which is first read in by the mission controller. This order specification contains the following attributes for each elements of each order:

- Name
- Conditionality on other elements and their values
- Critical or optional
- Data type
- Enumerated values if any and optionally a default value
- Units – enumerated list with an optional default unit

Orders in the mission order file are compared against their specification and only valid orders are passed through. Orders deemed syntactically invalid are flagged as such, and descriptive error messages are provided to aid the user in fixing them. The mission is not started if there are any invalid orders.

On generation of a mission abort event by any module in the hierarchy, all controllers are sent events which ensure that they gracefully terminate their activity and an end-of-mission sequence is initiated. This sequence can vary from vehicle to vehicle but typically takes the form of releasing a drop weight if one exists, or blowing ballast if the AUV is equipped with ballast tanks. The AUV is then either driven to the surface or allowed to float up since it is positively buoyant after releasing ballast.

Besides the modules shown in Fig. 2, there are additional modules in the Teja implementation of the MC. These comprise of data structure definitions and modules which communicate with the vehicle, i.e., those which make up the interface layer. Additionally, a faster-than-real-time simulation capability is built into the MC to allow mission to be rapidly simulated and viewed graphically. This capability is included in the MC as a mission planning aid. It is facilitated by simplified, linearized models of important vehicle behavior such as its hydrodynamics maneuvering characteristics, the behavior of a variable ballast system, etc. Many sensor subsystems are not modeled since the purpose of this simulation capability is to test the mission validity primarily from a navigation standpoint. The mission controller in the simulation is identical to the one which runs on the actual vehicle. Teja allows the generated code to be run as fast as possible on any given platform in simulation mode; all relative timing is maintained but absolute mission execution time is a fraction of the real mission time. This simulation capability has proven valuable during field trials of several AUVs.

4. Verification of Mission Controller Architecture

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.

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. 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.

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. 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. 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 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 userdefined 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 bottomup 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.

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

Following the above bottom-up approach, the process of checking progress for a hierarchical interacting hybrid system, $\boldsymbol{H}=\boldsymbol{H}^{1}_{1}\|\ldots\boldsymbol{H}^{j}_{i}\ldots\|\boldsymbol{H}^{m}_{n}$, where $\boldsymbol{H}^{j}_{i}=\{Q_i^j,\Sigma_i^j,U_i^j,Y_i^j,F_i^j,H_i^j,I_i^j,E_i^j,G_i^j,R_i^j\}$, is the hybrid automaton model of the jth module ($j=1\ldots m_i$) on the ith level ($i=1\ldots n$), the following algorithm describes the verification approach.

- For $i = 1$ *to* n
- For $j = 1$ *to* mi
  - Select subsystem $\boldsymbol{H}\,ji$ for verification
  - Find all subsystems $\boldsymbol{H}\,lk$, $k=1$ *to* n, $l=1$ *to* mk, that interact with subsystem $\boldsymbol{H}\,ji$
  - Abstract all subsystems $\boldsymbol{H}\,lk$, 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
  - Compose the system as $\boldsymbol{H'}=\boldsymbol{H}\,ji\,\|\,\boldsymbol{H}\,lk$ for k,l as found above
  - Formulate queries using temporal logic formulas based on the safety/progress requirements of the system, check queries on composed system $\boldsymbol{H'}$, and decide on progress
  - correct any safety/progress problems found for module $\boldsymbol{H}\,ji$
- 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.

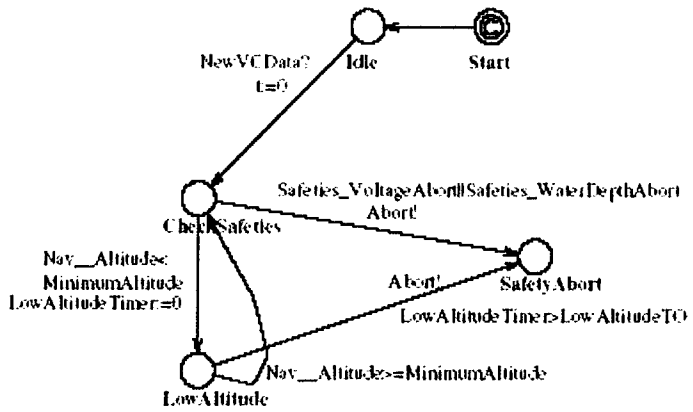

Figure 4 - Safeties subsystem

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 4, 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 5(a). The safeties driver must be included in the verification of every subsystem in the mission controller.

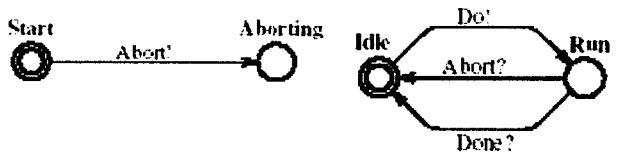

Figure 5: (a) Safeties driver (b) Generic driver susbsystem

All other drivers generally take the form shown in Figure 5(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 2, 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 requirementsthat 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 2 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 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 7b.

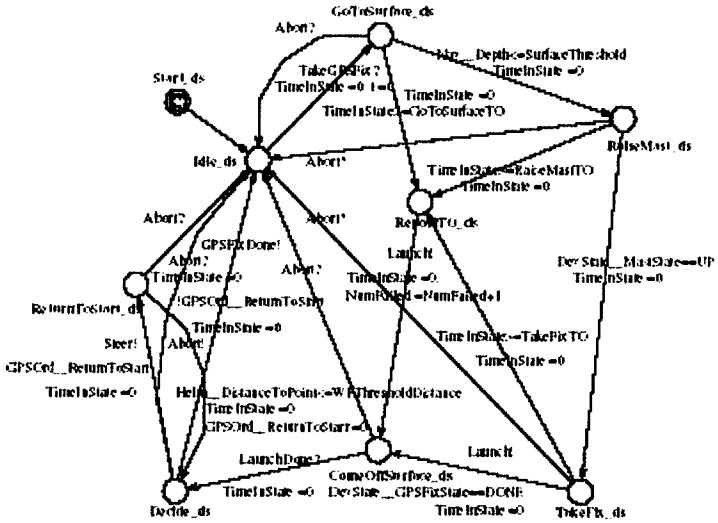

Figure 6: GPSFix subsystem

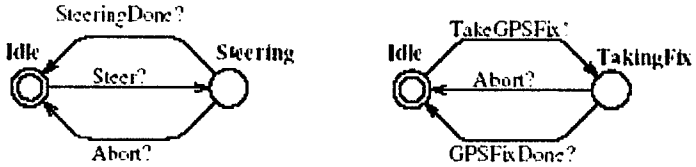

Figure 7: (a) Steering stub subsystem (b) GPSFix driver subsystem

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 7(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.

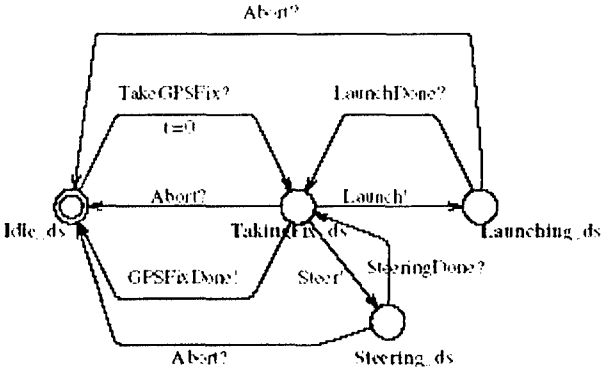

Figure 8: GPSFix stub subsystem

5. Animation of missions

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 complex system such as an autonomous underwater vehicle can undergo many sequences of events, which are impossible to anticipate humanly but can be seen visually in simulation. 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 which might miss situations that might occur. No such abstraction is needed for

simulation. Animation/Simulation involves both the continuous and discrete dynamics combined together to successfully execute a mission. So a simulation can capture some interactions which might be missed while carrying out verification. 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. 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 and establishes 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, 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 [1] into a format readable by Uppaal, a hybrid system modeling, simulation, and verification tool. For animation, the mission controller modules in Uppaal are further converted to animation modules of OpenGL.

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

OpenGLis 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** (Open**GL** **U**tility **T**oolkit) 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.

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 10), AUV raising mast after reaching the water surface to update the navigation system (Figure 10) and results during GPSFix operation execution (Figure 9).

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 $\boldsymbol{H}$ at the $Level_i$ to the converter
    The converter extract events σ, guard g_e and variables *Vars*
    Generate OpenGL code to model the events, guards using the variables extracted
  Next k
Next i

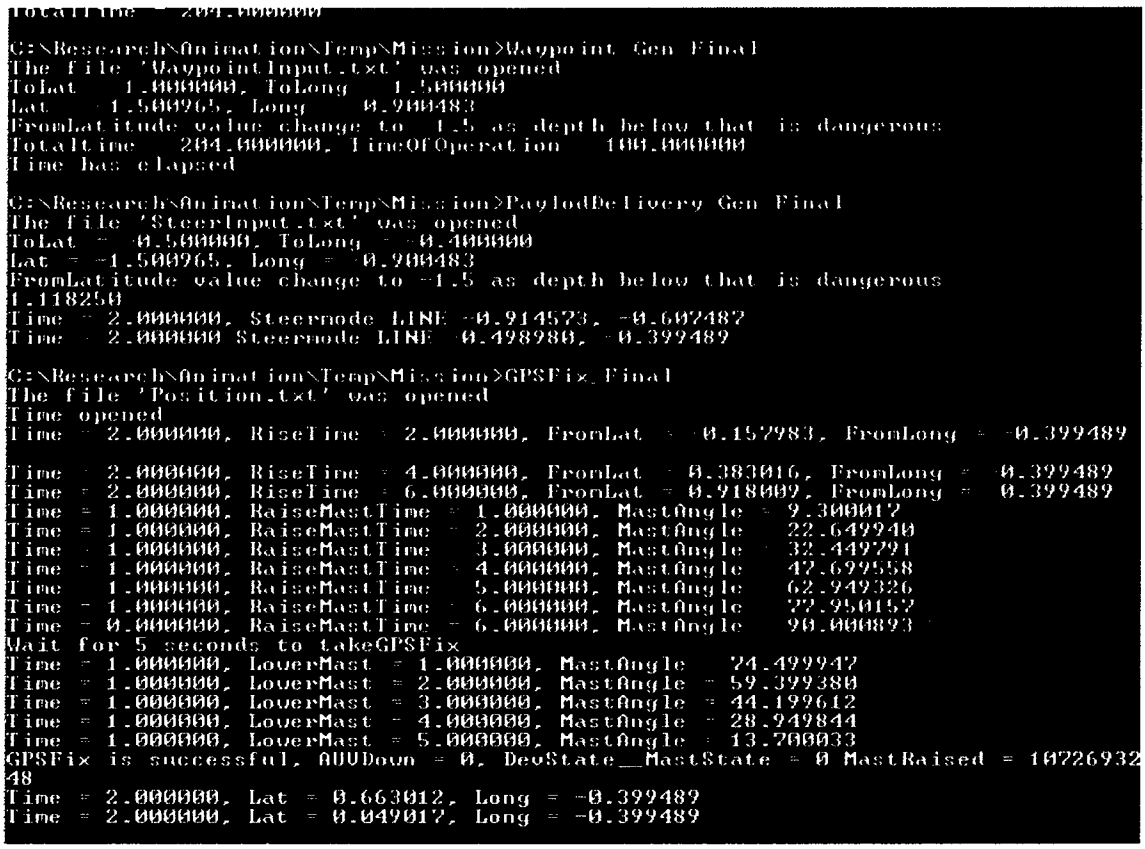

Figure 9: Result showing mission execution

c. Abstract of 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 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.

As an example, the abstracted code for the steering module is as given next.

#Initialization *(initializes the variables used in converter)*

 # Generate the initialization code

{ print OUTFILE "\n#include<stdio.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 =~ /int\s\S+:=\d/)

```
{       $stringofdecl = $&; # Store the pattern in a string
            @arrayofdecl = split(' ', $stringofdecl); # Get rid of int
        $stringofvar = $arrayofdecl[1]; # Get string of vars.
        @actualvar = split(',',$stringofvar); # pattern match
              for ($count = 0; $count<=$#actualvar; $count++) #
Iterate for all the variables
          {$stringid = $actualvar[$count]; # Store each variable in a string
              %sepvalue = split(':',$stringid); # Split variables based on colon
              print OUTFILE "static GLfloat "; print OUTFILE %sepvalue; 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) {...}
# Generate the graphics of the AUV and undersea env.
      print OUTFILE "\nvoid reshape (int w, int h)";      {...}
# Associating mouse and keyboard related actions
      print OUTFILE "\nvoid keyboard (unsigned char key, int x, int y)" {...}
```

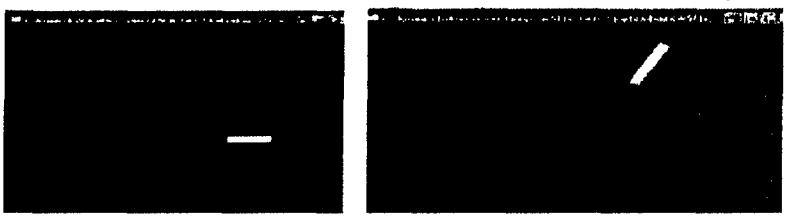

Figure 10: AUV (Green) moving up and raising Mast (Yellow) to execute GPSFix

6. Synthesis of coordinators

The research involved the automation of the synthesis of coordinators (i.e. controllers at the topmost layer) for hierarchy based intelligent control architecture for AUVs. The interactions within the modules in hierarchical intelligent control architecture are complex. Synthesis of a coordinator for such a system is a challenging task as it requires careful monitoring of the inputs received and the outputs send. The controller is a hybrid system with discrete states and continuous dynamics. The continuous dynamics are implemented as functions. The coordinators are a special case of hybrid system which only involves timing constraints known as timed automata.

Automated synthesis of the coordinators promises reduction in time to develop and implement coordinators for underwater and aerial vehicles. It also improves modification and debugging capability. Our goal in the automation of coordinators had been to

translate the higher level specifications and user inputs into sequence of actions to successfully execute the mission. The coordinators we synthesize are timed automata with timing constraints.

We consider the requirement of three coordinators at the topmost level. The three coordinators are a sequential coordinator (implementing sequential control to execute a sequence of actions for a mission), a timed coordinator (implementing time critical missions) and a safety coordinator (implements safe execution of mission). These coordinators are synthesized based on user input and high level specification. The coordinators consist of a basic structure and a synthesized part. The basic structure implements control common to any kind of mission coordinator built. For example each and every coordinator needs to establish connection with the vehicle before requesting a mission. The synthesized part is developed based on the specific mission to be executed. For example a mission can be find the present location using a GPS or fire a missile.

Sequential coordinator is used to coordinate the execution of sequence of actions involved in the successful execution of an untimed mission. The sequential coordinator is synthesized based on the inputs received and its response. Inputs received by the sequential coordinator can be requests made by the user i.e. the mission order or other coordinators at the same level or responses received from lower level or same level coordinators. The algorithm consists of two parts the first part implements the basic structure and the second part implements the augmentation of new edges, guards, reset values and locations to the sequential coordinator. The simplest structure of the sequential coordinator is shown in figure 1. The basic structure is the same for all the sequential coordinators which involves two different phases: Initialization phase, and Communication establishment phase. The mission specific structure contains mission phase, and response phase. The two other coordinators have the same basic structure.

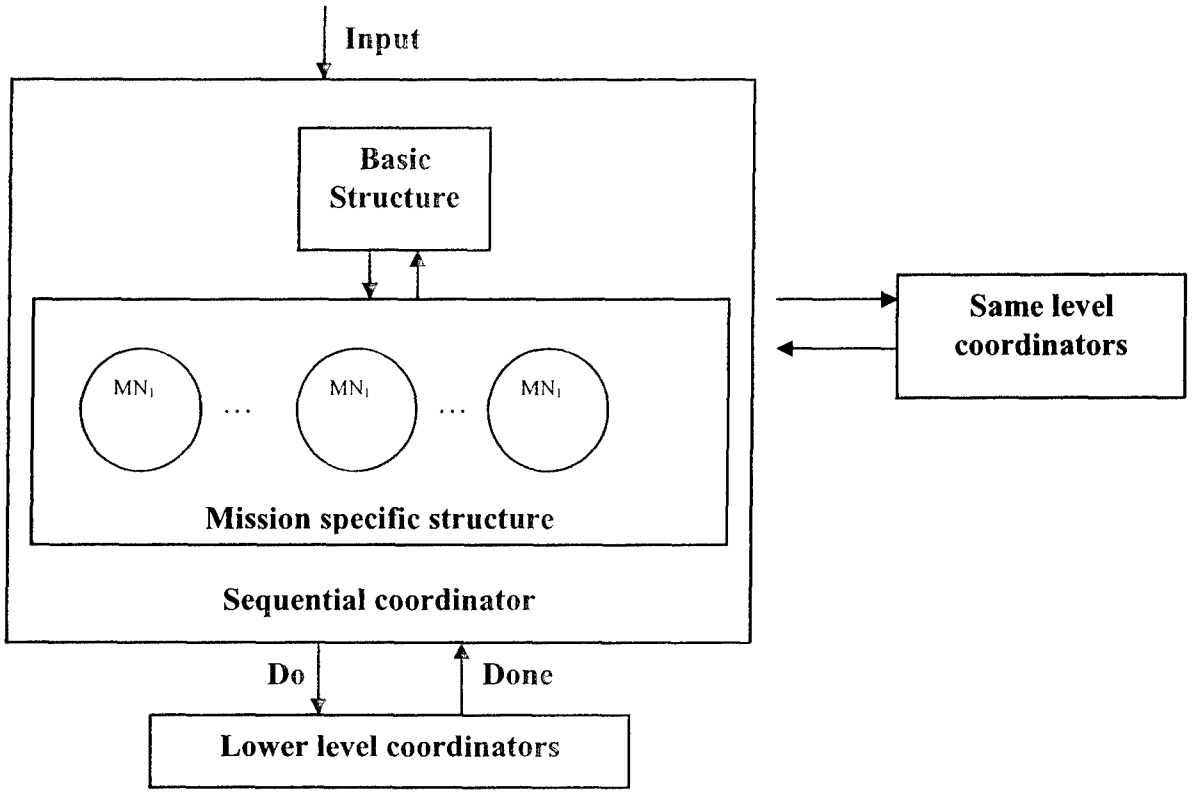

Figure 11: Basic structure of Sequential Coordinator

a. Sequential coordinator synthesis

Algorithm:

Create five locations $l \in L$ and name them as ***Idle, WaitforVCComms, Run, Suspend*** and ***Endmission***. (control for any AUV needs all these states)

- Create an edge e_0 from ***Idle*** to ***WaitforVCComms*** (indicating transition to a state to wait to establish communication with the Vehicle)
  - Set event $\sigma_{in} = Init$
  - Set guard condition $G^i(e_0) = t>=T$ where T is a constant time to initialize the system (for our case $T = 1$)
  - Set reset condition $R(e_0) = \{t=0\}$
- Create an edge e_1 to ***Run*** state from ***WaitforVCComms*** if a connection with vehicle is established
  - Set event $\sigma_{in} = NewVCData$
  - Set guard condition $G^i(e_1) = t>=10$
  - Set reset condition $R(e_1) = \{t = 0, MissionTime = 0, Suspendable = 0\}$
- Create an edge e_2 from ***Run*** state to ***EndMission*** state
  - Set event $\sigma_i = Endmission$
  - For each $Controller_i \in Level_j$ where i= 1…n, j = 1 only
    - Set the guard condition on the edge $G^i(e_2) = (\cup Controller_k\text{->Idle})$ where k = 1,2…n, k != i checking the status of other controllers (0 meaning idle)

- Set reset condition $R(e_2)$ = {t = 0, Suspendable = 0, Idle = 0} to indicate that all the coordinators are idle

- At ***EndMission*** state
  - Draw a self loop edge e_3
    - Set event σ_{in} = *OnSurface*
    - Set guard condition $G^i(e_3)$ = (Var_k < = *SurfaceThreshold*) where Var_k is k^{th} variable mapped to set of real numbers (indicating sensor value of depth) *SurfaceThreshold* indicates a constant value

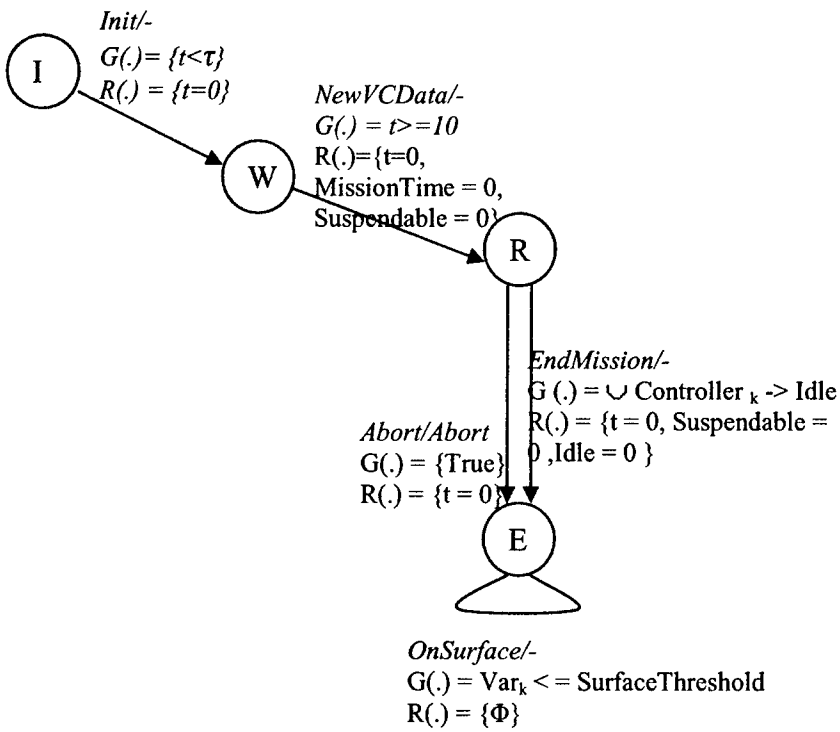

Figure 12: Basic structure for Sequential Coordinator

- **Start:** Get Mission order name **Or_n(<MissionName>, Prm).**
- If mission name is obtained for the first time
  - Create a location $l \in$ L and name it **<MissionName>**
  - Draw an edge e_i from the ***Run*** state to **<MissionName>** state where i = j+1… n, where j is the number for the last edge that was drawn
    - Set the events as σ_{in}/σ_o = **<MissionName> /Do<MissionName>** command sent to the lower order controllers
  - Set the guard condition $G^j(e_i)$ = {Var_k = **"<MissionName>"**}
  - Set the reset condition R(e_i) ={Suspendable = (0 or 1), Idle = 0, t = 0}
  - If **Suspendable** = 1
    - Create an edge e_i from **<Mission Name>** state to ***Suspend*** state
      - Set event σ_{in} = *Suspend*
      - Set guard condition $G(e_i)$ = *{True}*

- Set reset condition $R(e_i) = \{t = 0, Suspendable = 0\}$
- If connecting to the ***Suspend*** state for the first time
  - Create a self loop e_i at the ***Suspend*** state
    - Set the event $\sigma_{in} / \sigma_o =$ *Abort / Abort*
    - Set guard condition $G(e_i) = \{Var_k = !Suspended\}$
    - Set reset condition $R(e_i) = \{Suspended = 1\}$
  - Create an edge e_i from ***Suspend*** state to ***Run*** state
    - Set event $\sigma_{in} =$ *Resume*
    - Set guard condition $G(e_i) = \{True\}$
    - Set reset condition $R(e_i) = \{Suspended = 0, Suspendable = 0, t = 0\}$
- Draw an edge e_i from the **<Mission Name>** state to ***EndMission*** State
  - Set the $\sigma_{in}/\sigma_o =$ *Abort / Abort*
  - Set guard condition $G(e_i) = \{True\}$
  - Set reset condition $R(e_i) = \{t = 0\}$
- Create an edge e_i from **<Mission Name>** state to ***Run*** State
  - Set event $\sigma_{in} =$ **<MissionName>Done**
  - Set guard condition $G(e_i) = \{True\}$
  - Set reset condition $R(e_i) = \{t = 0, Suspendable = 0\}$

- Else if mission name is already there
  - Go to **Start** to get the name of the next mission
- End

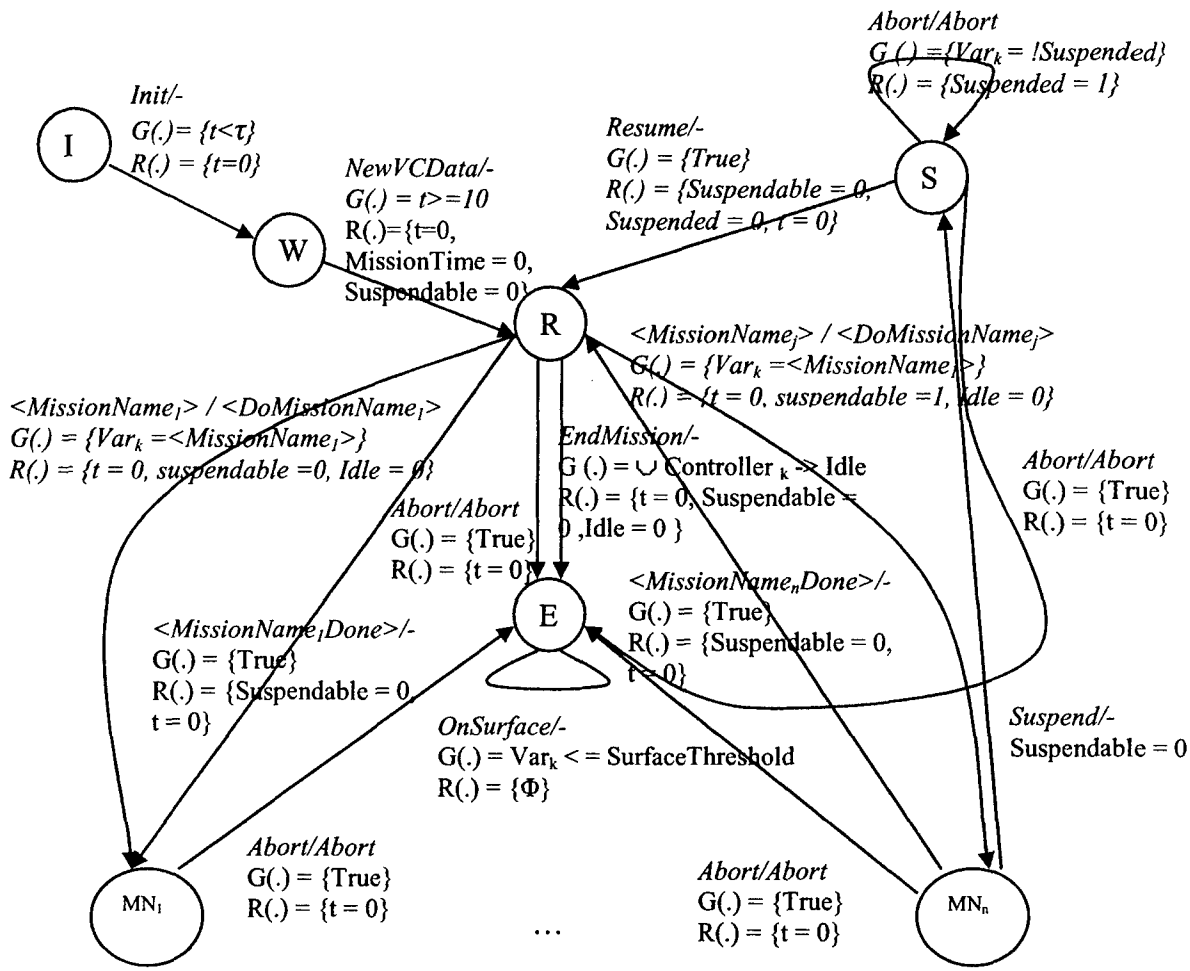

Figure 13: Sequential coordinator

b. Timed coordinator synthesis

Timed coordinator is used to coordinate the execution of time critical mission. Timed critical mission involves execution of sequence of actions with timing constraints. The timed coordinator is synthesized based on the inputs received and its response. Inputs received by the sequential coordinator can be requests made by the user i.e. the mission order or other coordinators at the same level or responses received from lower level or same level coordinators. The algorithm till the label Start implements the basic structure and the remaining part implements the augmentation of new edges, guards, reset values and locations to the sequential coordinator.

Algorithm:

- Create seven locations $l \in L$ and name them as ***Idle, WaitForFirstTO, CheckOrders, Wait4Suspend, Check4Resume, Decide*** and ***End.***
- Draw an edge e_0 from ***Idle*** state to ***WaitForFirstTO*** state
  - Set event σ_{in} = *Init/-*

- Set guard condition $G(e_0) = \{t >= \tau\}$ where τ is a constant
- Set reset condition $R(e_0) = \{ \Phi \}$

- Draw an edge e_1 from ***WaitForFirstTO*** to ***CheckOrders***
  - Set event σ_{in} = *NewVCData/-*
  - Set guard condition $G(e_1) = \{True\}$
  - Set reset condition $R(e_1) = \{MissionTime = 0, t = 0, Done = 1\}$
- Draw an edge e_2 from ***CheckOrders*** state to ***End*** state
  - Set event σ_{in} = *EndMission*
  - Set guard condition $G(e_2) = \{True\}$
  - Set reset condition $R(e_2) = \{Idle = 0\}$
- Draw an edge e_3 from ***CheckOrders*** state to ***Decide*** state
  - Set event σ_{in} = *NewOrder*
  - Set guard condition $G^i(e)$ = strcmp(this->CurrTimedOrd->Name,"None") && TimedActions_get_MissionTime() >= this->CurrTimedOrd->Time && (!TimedActions_CheckSuspend(this)||this->SeqController->Idle||this->SeqController->Suspended)
  - Set reset condition $R(e_3) = \{Idle = 0\}$
- Draw an edge e_4 from ***CheckOrders*** to ***Wait4Suspend*** (indicating that the mission requires suspension of the other coordinators)
  - Set σ_{in} = *Suspend/ Suspend*
  - Set guard condition $G(e_4)$ = *{ strcmp(this->CurrTimedOrd->Name,"None") && TimedActions_get_MissionTime() >= this->CurrTimedOrd->Time && (TimedActions_CheckSuspend(this)&&this->SeqController->Suspendable && !this->SeqController->Idle) &&!this->SeqController->Suspended}*
  - Set reset condition $R(e_4) = \{t = 0, Idle = 0, Time2Suspend = 0\}$
- Create a loop e_5 at ***Wait4Suspend*** state
  - Set σ_{in} = *Suspend/ Suspend* (suspend the Sequential Coordinator)
  - Set guard condition $G^i(e_5)$ = !this->SeqController->Suspended
  - Set reset condition $R(e_5) = \{t = 0\}$
- Draw an edge e_6 from ***Wait4Suspend*** to ***Decide*** state
  - Set σ_{in} = *NewOrder*
  - Set $G^i(e_6)$ = this->SeqController->Suspended
  - Set reset condition $R(e_6) = \{\Phi\}$
- Draw an edge e_7 from ***Check4Resume*** to ***CheckOrders*** without *Resume* event
  - Set σ_{in} = *OrderComplete/-*
  - Set the $G^i(e_7)$ = !this->SeqController->Suspended || (TimedActions_get_MissionTime() >= this->CurrTimedOrd->Time && strcmp(this->CurrTimedOrd->Name,"None"))
  - Set the reset condition $R(e_7) = \{\Phi\}$
- Draw an edge e_8 from ***Check4Resume*** to ***CheckOrders***
  - Set σ_{in} = *OrderComplete /Resume*
  - Set $G^i(e_8)$ = this->SeqController->Suspended && (TimedActions_get_MissionTime() < this->CurrTimedOrd->Time || !strcmp(this->CurrTimedOrd->Name,"None"))
  - Set the reset condition $R(e_8) = \{\Phi\}$

- Draw an edge e_9 from each of the states (excepting ***Idle*** and ***WaitForFirstTO***) to ***End*** state
  - Set event $\sigma_{in} / \sigma_o =$ *Abort/Abort*
  - Set guard condition $G^i(e_9) = \{True\}$
  - Set reset condition $R(e_9) = \{\Phi\}$
- **Start:** Get Mission order name $\mathbf{Or_n}$**(<MissionName>, Prm).**
- If mission name is obtained for the first time
  - Create a location $l \in L$ and name it **<MissionName>**
  - Draw an edge e_i from the ***Decide*** state to **<MissionName>** state where i = j+1…n where j is the number for the last edge drawn
    - Set $\sigma_{in} / \sigma_o =$ **<MissionName>/ Do<MissionName>** sent to lower level controllers
    - Set guard condition $G(e_i) = \{CurrentOrder = <MissionName>\}$
    - Set reset condition $R(e_i) = \{\Phi\}$
  - Draw an edge e_i from the **<MissionName>** state to ***End*** State
    - Set $\sigma_{in} / \sigma_o =$ *Abort*
    - Set guard condition $G(e_i) = \{True\}$
    - Set reset condition $R(e_i) = \{\Phi\}$
  - Create an edge e_i from **<MissionName>** state to ***Check4Resume*** State
    - Set $\sigma_{in} =$ **<MissionName>Done** signal
    - Set guard condition $G(e_i) = \{True\}$
    - Set reset condition $R(e_i) = \{\Phi\}$
- Else if mission name is already there
  - Go to **Start** to look for the next order
- End

c. *Safety Coordinator synthesis*

Safety by definition is the freedom from danger, damage or risk. Thus the goal of a safety coordinator is to prevent the vehicle from taking actions which might damage the vehicle. The safety coordinator monitors the different parameters involved in the missions ordered by mission coordinators, the proper functioning of the components of the vehicle and the environment surrounding the vehicle. So a safety coordinator basically is an observer which acts only when the operations lead to unsafe state. When the safety coordinator finds that a mission prompts execution of an unsafe action it tries to correct the action and make it safe. If the safety coordinator is not able to make the mission safe it aborts the mission. For example if a mission commands the vehicle to go to a depth of 500ft and the present safe depth is only 200ft the safety coordinator changes the depth to 200ft. If the safety coordinator is able to correct it the mission is carried out or else it aborts the mission. We here list a set of safety issues a safety coordinator should satisfy.

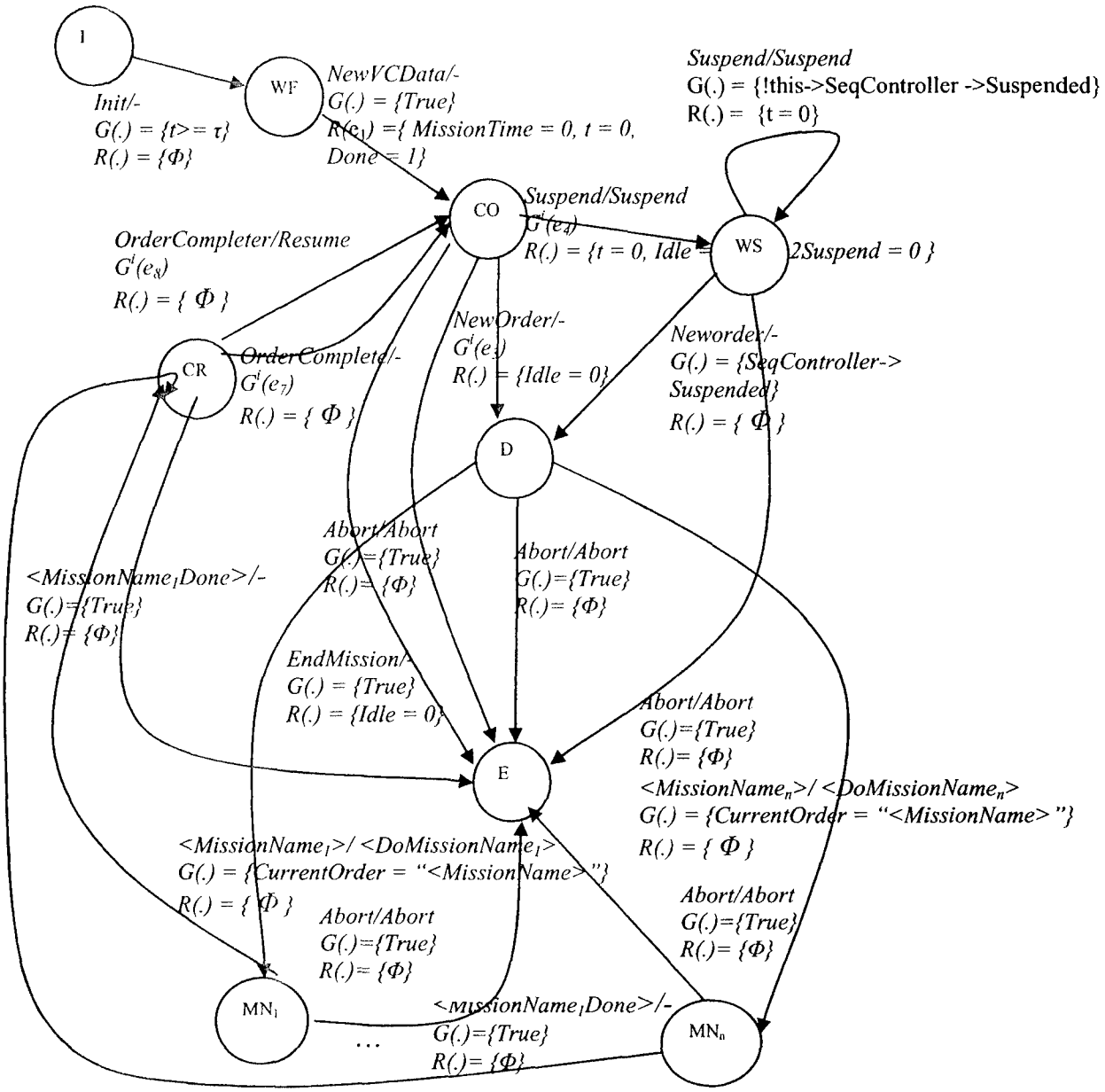

Figure 14 : Timed coordinator

The safety issues which a safety coordinator for an AUV should take care of are as listed below.

1. *Water depth safety* monitoring should check the altitude of the vehicle from the bottom of the sea and thus prevent the vehicle from hitting the bottom of the sea.
2. *Obstacle avoidance safety* should monitor the presence of obstacles which might be other vehicles, or mountains under sea and prevent collision of the AUV with the obstacle.
3. *Device functioning safety* should monitor the functioning of the different critical components which constitute an AUV. Critical components are those components malfunctioning of which might lead to damage of vehicle or undesirable situation like AUV stuck at the bottom of the sea due to battery failure.

All these safety issues can be modeled as constraints within a hybrid system as has been done for the survey AUV built at ARL. The constraints are the guard conditions which prompt the transition from one state to other depending upon the situation.
The coordinators synthesized here work together to successfully execute a mission. It is shown next. Given a mission the coordinators communicate among each other to successfully complete a mission.

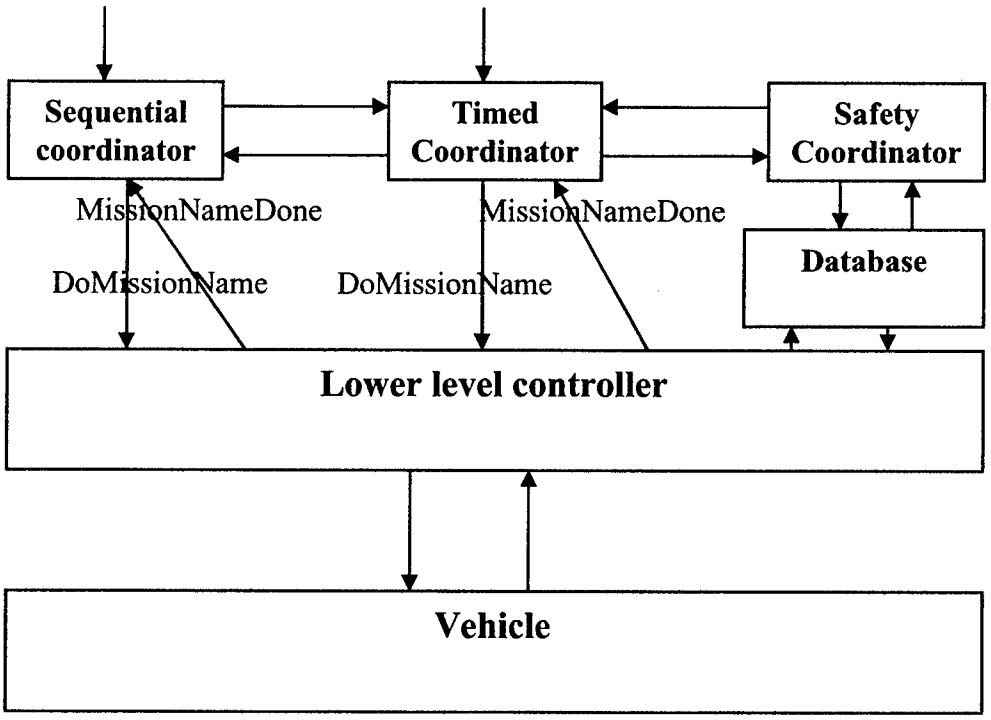

Figure 15: The complete structure

Here we are concerned with the successful execution of the mission as ordered by the highest level controllers which we have automatically synthesized.
The analysis is provided based on the interactions between the modules shown in Figure and the detailed modules of the sequential (Figure) and timed coordinator (Figure).
Both the coordinators are initialized first. During initialization the sequential coordinator establishes contact with the vehicle and the terminal from which mission orders are received.
When new order is received both the coordinators transition to the state at which they become ready to execute a mission. If it's an untimed mission the sequential coordinator accepts the input and sends **<DoMissionName>** (Figure) to the lower level controllers. Once the mission is successfully executed the sequential coordinator receives **<MissionNameDone>** (Figure) from the lower level controllers. Then the SC considers the next order in queue and passes control to the concerned lower level controller. If due to some malfunctioning the mission needs to be terminated an *abort* signal is received by the sequential coordinator from the lower level controllers involved in the mission. The sequential coordinator then broadcasts the *abort* signal (Figure 13) and terminates the

execution of all other missions. If there are no more orders in the queue the SC checks for the status of the TC. If the TC is idle SC sends *EndMission* and transitions to the ***EndMission*** state (Figure).
If a timed mission is received then the timed coordinator checks whether the execution of the present mission needs the suspension of the sequential coordinator or not (these constraints are guard conditions on edges). If TC needs to suspend SC, TC sends the *suspend* signal to SC (Figure). If the mission which SC is executing is suspendable then SC synchronizes with the event *suspend* and transitions to the ***Suspend*** state (Figure). When SC is suspended TC sends the order as **<DoMissionName>** to the lower level controllers (Figure). The lower level controllers respond back with the **<MissionNameDone>** event to the TC when the mission is completed (Figure). TC then finds the next order in queue and either resumes the SC or keeps it suspended or keeps it unsuspended.

7. Conclusion and future work

We have proposed hierarchical hybrid mission control architecture for AUVs. The architecture has been successfully implemented at ARL. The modular approach to execute a mission breaks down complex missions into simple modules which aids in easy design and implementation of the architecture. The present architecture is also not strictly priority driven. A priority driven architecture with timed and untimed missions strictly separated would lead to lesser missions being expired and give efficient performance. A priority driven system with higher priority for timed missions would look into all the timed and untimed missions in a queue and execute timed missions before the untimed ones if there is a possibility of missing the successful completion of timed missions.
A real-life, complex, and hierarchically structured hybrid control system has been verified using our bottom up approach. The advantage of bottom-up approach is reduction of complexity, and also if an error is detected in a certain module, the lower level modules do not need to be revised. The current approaches typically consider reachability properties for verification, but through our modular bottom up approach we are able to analyze the correctness of the entire controller. As far as logical correctness is concerned, we verified 12 different modules against a total of 148 queries; the table shows the number of queries and the name of the module. The verification confirms the correctness of designed modules for progress.

S.No.	Name of the subsystem	No. Of Queries
1	Steering	3
2	Loiter	22
3	Rendezvous	8
4	Payload	4
5	Pause	2
6	Launcher	4
7	GPSFixer	12
8	DeviceCommander	4

9	WaypointNavigator	26
10	Sequential Coordinator	34
11	Timed Coordinator	25
12	Safety Coordinator	4

The problem of complexity still exists if a modular design is not performed, and properties to be verified are not dependent on behaviors of small sub-collection of modules, rather the entire set of modules. In this present work we verified logical correctness of the missions executed. The correctness of function-calls is another issue not addressed here. Function-call verification will include the verification of the whole hybrid system. The present architecture can be extended to multiple underwater-vehicles.
We have simulated a hierarchically organized mission controller architecture using a bottom up approach to conversion from coordinator modules to OpenGL code. The simulation involved all the coordinators involved in a specific operation and thus strengthened the correctness of the model. The simulation proved the feasibility of building a simulation tool for a mission driven AUV. The simulation tool can be further developed and generalized to be used for any kind of mission (other than just survey) for an AUV. The simulation tool can also be enhanced to get real time sensor information as feedback and then take actions accordingly. The simulation tool can be further advanced so as to implement the complex mathematical models involved and give a much more accurate and attractive result.
Finally we have designed automated synthesis of coordinators. These synthesis algorithms need to be implemented. The coordinators synthesized (with modification of safety coordinator) can be used in future for hierarchical hybrid mission control architecture for aerial vehicles as well.

8. Publications

1. S. Tangirala, R. Kumar, S. Bhattacharyya, M. O'Connor, and L. E. Holloway, ``Hybrid-Model based Hierarchical Mission Control Architecture for Autonomous Underwater Vehicles", IEEE Transactions on Automation Science and Eng., Submitted.

2. M. O'Connor, S. Tangirala, R. Kumar, S. Bhattacharyya, M. Sznaire, and L. E. Holloway, ``A Bottom-up Approach to Verification of Hybrid Model-Based Hierarchical Controllers with application to Underwater Vehicles", 2006 American Control Conference, Minneapolis, June 2006.

3. S. Bhattacharyya, R. Kumar, S. Tangirala, M. O'Connor, and L. E. Holloway, ``Animation/Simulation of Missions for Autonomous Underwater Vehicles with Hybrid-Model based Hierarchical Mission Control Architecture", 2006 American Control Conference, Minneapolis, June 2006.

4. Matt O'Connor, Sekhar Tangirala, Ratnesh Kumar, John Dzielski, and Mario Sznaier, "A Bottom-Up Approach to Verification of Hybrid Model-Based Hierarchical Controllers with Application to Underwater Vehicles," Proceedings of the 14th

International Symposium on Unmanned Untethered Submersible Technology (UUST05), Durham, NH, 2005.

5. S. Tangirala, R. Kumar, S. Bhattacharyya, M. O'Connor, and L. E. Holloway, ``Hybrid-Model based Hierarchical Mission Control Architecture for Autonomous Underwater Vehicles", 2005 American Control Conference, Portland, OR, June 2005.

Appendix A: Commands for the underwater vehicle for search

Abort: This command is given to terminate an operation or procedure before completion, if some other higher priority operation needs to be taken care of or the present job doesn't need to be done.
DeviceDone: This response is sent by the Device Commander when a device required for an operation has been set.
GoToEndMission: This command is sent to indicate that a mission has been accomplished so all the operations are terminated.
GoToRendezvous: This command is sent to go to the desired meeting point.
GPSFixDone: This is response sent by the GPSFixer once the global positioning system finds the position.
Init: This command is sent by all the modules as an initializing command after which it transitions to the Idle state from the Start state and becomes ready for operation.
Launch: This command is sent to activate the Launcher module.
LaunchDone: This is response to the command launch sent once the function of the launcher is done with.
MastDown: This command is sent to the vehicle controller mast to lower the mast.
MastUp: This command is sent to rise the mast.
NewVCData: This is the data obtained by the Vehicle controller sensors.
OnSurface: This command is sent to indicate that the underwater sea vehicle has reached the water surface.
PayLoadDone: This is response given by the PayLoad module to indicate that payload has been delivered.
ProcessPayLoad: This command is sent to start the processing of the payload.
ProcessWP: This command is sent to process the direction of the vehicle based on the way point (it gives the coordinates of a point) it needs to go to.
RendezvousDone: This is response given to indicate that the rendezvous with other underwater vehicles has been done.
Resume: This command is given to resume operation after suspension.
SetDevice: This command is given to the Device commander to set a concerned device for a certain operation.
SuspendBehavior: This command is given to suspend a behavior.
TakeGPSFix: This command is given to find the global position of the vehicle.
Update: This command updates the present location of the vehicle.
Wait: This command is given to wait for sometime before staring on or resuming some operation.
WaitDone: This response is sent to indicate that waiting period is over now operation can be resumed.
WPDone: This response indicates that way point or location has been found.
AltitudeOK: This respone says that the depth to which the vehicle needs to go to is safe.
AltitudeSafety: This command checks whether the altitude level to go to is safe or not.
DeliverPayLoad: This command tells to deliver the payload.
GoToLoiter: This command tells the vehicle to go to start loitering.

LightOff: This command checks whether light is switched off
Loiter: This command is given to the vehicle to loiter in its surrounding area.
LoiterDone: This response indicates that loitering is done.
Steer: This command is sent to steer the vehicle to the desired location.
SurfaceCaptured: This command tells that the surface of interest has been captured.
TimeInState: This command indicates the time duration spent in a state.
TimeOut: This command indicates the time within which an event has to be conducted otherwise it is not executed.
Trim: This command is issued to increase speed of the vehicle.

Appendix B : Hybrid models in Teja

Sequential coordinator

Superclass: TejaComponent

Variables: NumWP, SurfaceThreshold, WPNum, Suspended, StartUTCTime, Suspendable, Idle.

Links: DevCmd (DeviceCmd), Actreq (ActionRequest), VehCmd (VehicleCmd), AutCmd (AutopilotCmd), Nav (NavState), Logs (Files), WPNav (WaypointNavigator), NonSeqController (TimedActions), Payload (PayloadDelivery), Components (ComponentList), Missionqueues (Queues), CurrOder (SeqOrder), DevCmdr (DeviceCommander), GPSFix (GPSFixer), Devstate (DeviceState), Waiter (Pause).

Functions:
ReadParams( ) is a function to read the parameters needed by the steer module to get executed successfully.
EndMission() is a function used to end a mission.
getMissionTime() is a function which returns the duration for which a mission is being executed till the current time.

Input: No input

Constructors:
DevCmd=p_devicecmd (initialzed to point to DeviceCmd)
ActReq=p_actionrequest (initialzed to point to ActionRequest)
VehCmd=p_vehiclecmd (initialzed to point to VehicleCmd)
AutCmd=p_autopilotcmd (initialzed to point to AutopilotCmd)
Nav=p_navstate (initialzed to point to NavState)
Logs=p_files (initialzed to point to Files)
WPNav=p_waypointnavigator (initialzed to point to Waypoint Navigator)
NonSeqController=p_timedactions (initialzed to point to TimedActions)
Payload=p_payloaddelivery (initialzed to point to PayloadDelivery)
Components=p_componentlist (initialzed to point to ComponentList)
MissionQueues=p_queues (initialzed to point to Queues)
DevCmdr=p_devicecommander (initialzed to point to DeviceCommander)
GPSFix=p_gpsfixer (initialzed to point to GPSFixer)
DevState=p_devicestate (initialzed to point to DeviceState)
Waiter=p_pause (initialzed to point to Pause)
WPNum=0 (initialzed to zero)

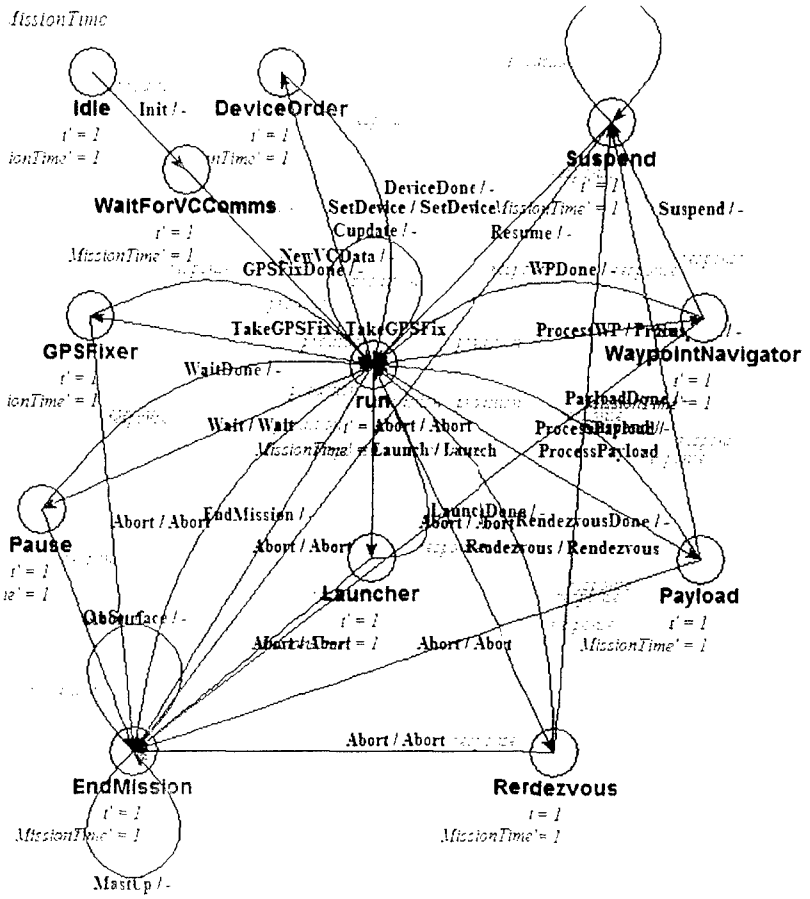

Figure 1: FSM for Sequential coordinator

Destructors: No destructors
Continuous states: t, MissionTime
Discrete states:
State1{ ***Idle***,t',MissionTime'}
State2{ ***DeviceOrder***, t',MissionTime'}
State3{ ***WaitForVCComms***, t',MissionTime'}
State4{ ***Suspend***, t',MissionTime'}
State5{ ***Pause***, t',MissionTime'}
State6{ ***GPSFixer***, t',MissionTime'}
State7{ ***run***, t',MissionTime'}
State8{ ***WaypointNavigator***, t',MissionTime'}
State9{ ***Launcher***, t',MissionTime'}
State10{ ***EndMission***, t',MissionTime'}
State11{ ***Rendezvous***, t',MissionTime'}
State12{***Payload***, t',MissionTime'}
Transitions:
Transition 1 {***Idle, WaitForVCComms,*** Init, t>=1, None, (CreateLogs(), ReadParams(), ReadOrderSpecs(), ReadMissionOrders() (and performs the actions as per need))}
Transition 2 {***WaitForVCComms, run, NewVCData***, True, (t=0, MissionTime=0, Suspendable=0, Idle=1), starts mission}

Transition 3 {***run, DeviceOrder, SetDevice***, True, (t=0, Suspendable=0, Idle=0), signal sent to device for execution}
Transition 4 {***DeviceOrder, run, DeviceDone***, True, (t=0, Suspendable=0, Idle=1), device command executed returns}
Transition 5 {***run, run, update***, t>=1, t=0, looks for new data or order availability}
Transition 6 {***Suspend, run, Resume***, True, (t=0, Suspended=0, Suspendable=0, Idle=1), resuming sequential execution}
Transition 7 {***Suspend, Suspend, Abort***, Suspended=True, Suspended=1}
Transition 8 {***WaypointNavigator, Suspend, Suspend***, True, (t=0, Suspendable=0, Idle=1), suspending waypoint}
Transition 9 {***Payload, Suspend, Suspend***, True, (t=0, Suspendable=0, Idle=1), suspending payload}
Transition 10 {***Rendezvous, Suspend, Suspend***, True, (t=0, Suspendable=0, Idle=1), suspending rendezvous}
Transition 11 {***Suspend, EndMission, Abort***, True, aborting from Suspend to endmission}
Transition 12 {***run, WaypointNavigator, ProcessWP***, True, (t=0, Suspendable=1, Idle=0), sending the datas to evaluate waypoint and perform actions accordingly}
Transition 13 {***WaypointNavigator, run, WPDone***, True, (t=0, Suspendable=0, Idle=1), mission to evaluate waypoint is executed}
Transition 14 {***run, Payload, ProcessPayload***, (CurrorderName=Payload), (t=0, Suspendable=1, Idle=0), evaluates waypoint to process payload}
Transition 15 {***Payload, run, PayloadDone***, True, (t=0, Suspendable=0, Idle=1), it states payload is done}
Transition 16 {***run, Launcher, Launch***, (CurrOrder=Launch), (Suspendable=0, Idle=0), launching vehicle }
Transition 17 {***Launcher, run, LaunchDone***, True, (t=0, Suspendable=0, Idle=1), launch has been done }
Transition 18 {***run, Pause, Wait***, (CurrOrder=Wait), (t=0, Suspendable=0, Idle=0), turning SSS and waiting }
Transition 19 {***Pause,run, WaitDone***, True, (t=0, Suspendable=0, Idle=1), sequential orders paused to wait to complete timed orders}
Transition 20 {***run, EndMission, EndMission***, (CurrOrder=EndMission and NonSeqController->Idle), end mission stopping prop and surfacing}
Transition 21 { ***EndMission, EndMission, OnSurface***, (Depth<=SurfaceThreshold and MastCmd!=Up), None, Comes on surface and mast is raised up}
Transition 22 { ***EndMission, EndMission, MastUp***, (MastState=Up), None, mast up and exiting mission}
Transition 23 {***Suspend, EndMission, Abort***, True, t=0, aborting from suspend }
Transition 24 {***WaypointNavigator, EndMission, Abort***, True, t=0, aborting from WaypointNavigator }
Transition 25 {***Payload, EndMission, Abort***, True, t=0, aborting from Payload}
Transition 26 {***Rendezvous, EndMission, Abort***, True, t=0, aborting from Rendezvous}
Transition 27 {***run, EndMission, Abort***, True, t=0, aborting from run}
Transition 28 {***Pause, EndMission, Abort***, True, t=0, aborting from Pause}
Transition 29 {***GPSFixer, EndMission, Abort***, True, t=0, aborting from GPSFixer}

Transition 30 {***Launcher, EndMission, Abort***, True, t=0, aborting from Launcher}

Timed Action (Timed Coordinator)

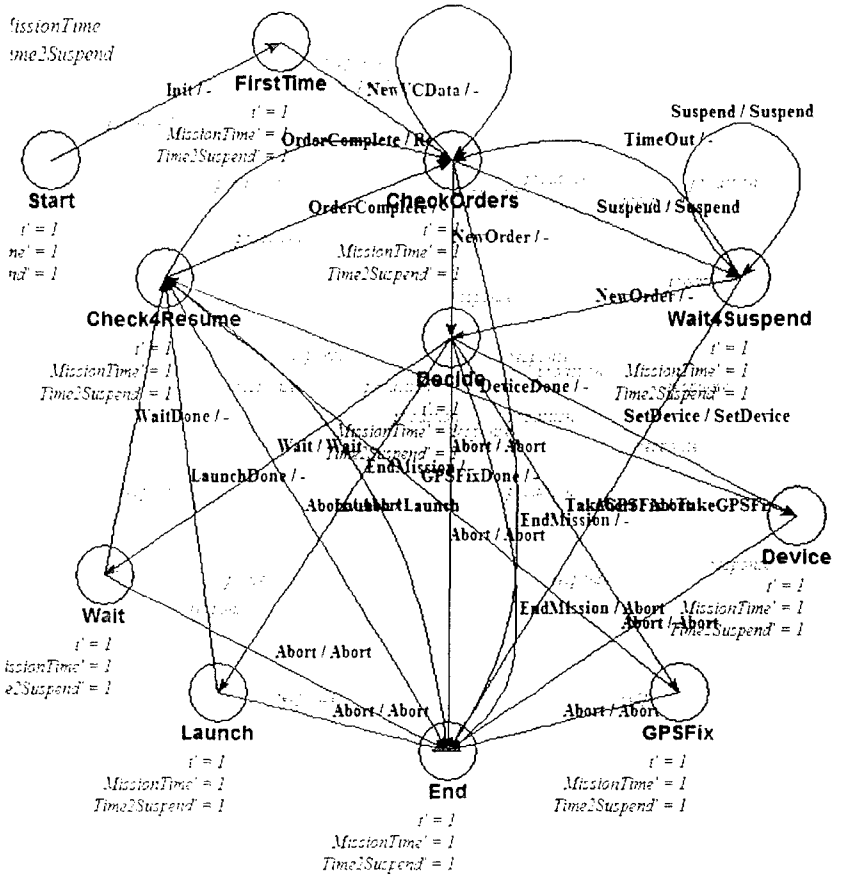

Figure 2: FSM of Timed coordinator

Superclass: TejaComponent

Variables: Token, TimedOrderTo, Suspend, Idle

Links: CurrTimeOrd (TimedOrder), Nav (NavState), NavMemory (NavState), SequenceController (Controller), Logs (Files), Components (ComponentList), MissionQueues (Queues), SeqOrdMemory (SeqOrder), CurrOrder (GPSOrder), GPSFix (GPSFixer), DevCmdr (DeviceCommander), Diver (Launcher), Waiter (Pause).

Functions:
ReadParams( ) is a function to read the parameters needed by the steer module to get executed successfully.
GetEarliestOrder() is used to retrieve the timed order with the earliest time requirement
CheckOrderTimes() is used to convert UTC times to mission times and check if > 0

Constructors:
MissionQueues=p_queues (initialzed to point to order Queue)
Nav=p_navstate (initialzed to point to NavState)
SeqController=p_controller (initialzed to point to Controller)
Logs=p_files (initialzed to point to Files)
Components=p_componentlist (initialzed to point to ComponentList)
GPSFix=p_gpsfixer (initialzed to point to GPSFixer)
DevCmdr=p_devicecommander (initialzed to point to DeviceCommander)
Diver=p_launcher (initialzed to point to Launcher)
Waiter=p_pause (initialzed to point to Pause)
NavMemory=NavState_new(teja_default_memory_space_id) (initialzed memory for NavState)

Desctructors: None

Continuous state: t, MissionTime

Discrete State:
State1{***Start,***t',MissionTime'}
State2{**FirstTime,**t',MissionTime'}
State3{***Decide,***t',MissionTime'}
State4{***CheckOrders,***t',MissionTime'}
State5{***GPSFix,***t',MissionTime'}
State6{***Device,***t',MissionTime'}
State7{***Wait,***t',MissionTime'}
State8{***Launch,***t',MissionTime'}
State9{**Wait4Suspend,** t', MissionTime'}
State10{**Check4Resume**, t', MissionTime'}
State11{**End**, t', MissionTime'}

Transitions:
Transition 1 {***Start,* **FirstTime**, Init, t>=1, None, Checks for availability of data}
Transition 2 { **FirstTime,** ***CheckOrders,*** NewVCData, MissionQueue>0, (MissionTime=0, Idle=0, t=0, Token=1), CheckOrderTimes and Loading Timed Order}
Transition 3 {***CheckOrders, End,*** Abort, True, None, Aborting}
Transition 4 {***Decide, End,*** Abort, True, None, Aborting Decide}
Transition 5 {***Check4Resume, CheckOrders,*** OrderComplete, (Token and TimedOrderQueue=0),(Idle=1, t=0, Token=1), No more Timed Orders so going to Idle state waiting for more tied orders}
Transition 6 {***Launch, End,*** Abort, True, None, Aborting Launch state}
Transition 7 {***Wait, End,*** Abort, True, None, Aborting Wait state}
Transition 8 {***Device, End,*** Abort, True, None, Aborting Device state}
Transition 9 {***GPSFix, End,*** Abort, True, None, Aborting GPSFix state}

Transition 10 {***Launch, Check4Resume,*** LaunchDone, ! (suspend and MissionQueue)=0, (Idle=1, t=0, Token=1), (teja_get_time(), TimedActions_get_MissionTime())}
Transition 11 {***Wait, Check4Resume,*** WaitDone, ! (suspend and MissionQueue)=0, (Idle=1, t=0, Token=1), (teja_get_time(),TimedActions_get_MissionTime())}
Transition 12 {***Device, Check4Resume,*** DeviceDone, ! (suspend and MissionQueue)=0, (Idle=0, t=0, Token=1), (teja_get_time(),TimedActions_get_MissionTime())}
Transition 13 {***GPSFix, Check4Resume,*** GPSFixDone, ! (suspend and MissionQueue)=0, (Idle=1, t=0, Token=0), (teja_get_time(),TimedActions_get_MissionTime())}

Transition 14 {***Wait4Suspend, Decide, NewOrder***, (TimedOrderQueue)>0, (t=0), (teja_get_time(),TimedActions_get_MissionTime(),CurrTimedOrd())}
Transition 15 {***CheckOrders, Wait4Suspend,*** Suspend, (TimedActions_get_MissionTime() >= CurrTimedOrd && Suspend), (Token=0,t=0), Store Current Nav State and Store Current sequential order }

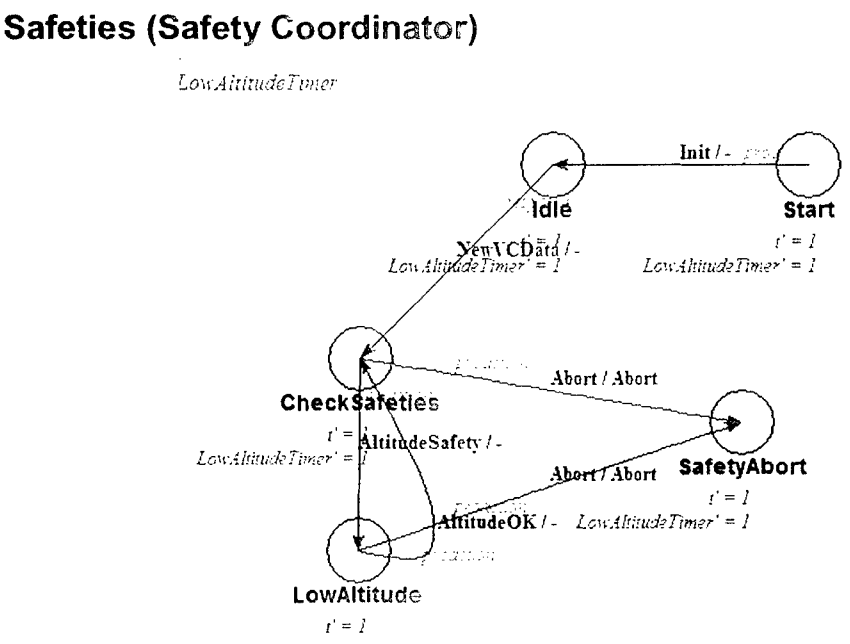

Figure 3: FSM of Safety coordinator

Superclass: TejaComponent

Variables: WaterDepthSafety, LowBatteryVoltage, MinimumAltitude, LowAltitudeTo.

Links: Bat(BatteryState), Nav(NavState), Logs(Files), Components(ComponentList).

Functions:

VoltageAbort() is a function which checks whether the average voltage is less than a threshold and accordingly returns a value.
WaterDepthAbort() is a function which checks whether the depth to which the vehicle can go to is safe or not and returns a value accordingly.
ReadParams() is a function to read the parameters needed by the safety module to get executed successfully.

Constructors:
Bat=p_batterystate; (initialzed to point to BatteryState)
Nav=p_navstate; (initialzed to point to NavState)
Logs=p_files; (initialzed to point to Files)
Components=p_componentlist; (initialzed to point to ComponentList)

Destructor: No destructors.

Continuous state: t, LowAltitudeTimer

Discrete States:
State 1 {***Start***, t'=1, LowAltitudeTimer'=1}
State 2 {***Idle***, t'=1, LowAltitudeTimer'=1}
State 3 {***CheckSafeties***, t'=1, LowAltitudeTimer'=1}
State 4 {***SafetyAbort***, t'=1, LowAltitudeTimer'=1}
State 5 {***LowAltitude***, t'=1, LowAltitudeTimer'=1}
State 6 {***Error***, t'=1, LowAltitudeTimer'=1}
State 7 {***Stop***, none}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, ReadParams}
Transition 2 {***Idle, CheckSafeties***, NewVCData, True, t=0, (prints teja_get_time(), Safeties_get_t() in execlog and flushes execlog)}
Transition 3 {***CheckSafeties, LowAltitude, AltitudeSafety***, (Altitude< MinimumAltitude) , LowAltitudeTimer=0, None}
Transition 4 {***CheckSafeties, SafetyAbort, Abort***, (VoltageAbort or WaterDepthAbort), None, (VoltageAbort or WaterDepthAbort (print teja_get_time(),Safeties_get_t() into errorlog, flushes errorrlog))}
Transition 5 {***LowAltitude, SafetyAbort, Abort***, (Safeties_VoltageAbort() or Safeties_WaterDepthAbort() or Safeties_get_LowAltitudeTimer() >LowAltitudeTO), (Safeties_VoltageAbort or Safeties_WaterDepthAbort or Safeties_get_LowAltitudeTimer() > LowAltitudeTO) (print teja_get_time(), Safeties_get_t() to errorlog , flush errorlog finally) }
Transition 6 {***LowAltitude, Checksafeties, AltitudeOk***, Altitude>MinimumAltitude, None, None}
Transition 7 {***Error, Stop***, Error, True, None, None}

ReplayMission

The ReplayMission module is used to write a human readable commands file. It takes in the input commands and writes out them in formatted output file.

Superclass: TejaComponent

Variables: None

Links: Bat, Dev, Nav, Veh

Functions:
Quicklook() is a function which writes the commands in human readable form.

Constructors:
Bat=BatteryState_new(teja_default_memory_space_id,NUMBEROFBATTERYSWITCHES); (Initializes space to Bat of type BatteryState)
Dev=DeviceState_new(teja_default_memory_space_id); (Initializes space to Dev of type DeviceState)
Nav=NavState_new(teja_default_memory_space_id); (Initializes space to Nav of type NavState)
Veh=VehicleState_new(teja_default_memory_space_id); (Initializes space to Veh of type VehicleState)

Destructor: No destructors.

Continuous state: t

Discrete States:
State 1 {***Idle***, tâ€™=1}
State 2 {***Error***, tâ€™=1}
State 3 {***Stop***, None}

Transitions:
Transition 1 {***Idle, Idle***, Init, t>=1, None, (ReplayMission_Quicklook() print "Quicklook files created, ending Replay") }
Transition 2 {***Error, Stop***, Error, True, None, None}

GPSFixer

Superclass: TejaComponent

Variables: GoToSurfaceTo, RaiseMastTo, TakeMastTo, SurfaceThreshold, NumFailed, WPThresholdDistance.

Links: Nav (NavState), DevState (DeviceState), AutCmd(AutopilotCmd), DevCmd (DeviceCmd), VehCmd (VehicleCmd), ActReq (ActionRequest), Logs (Files),

Components (ComponentList), NavMemory (NavState), Helm (Steering), GPSOrd (GPSOrder).

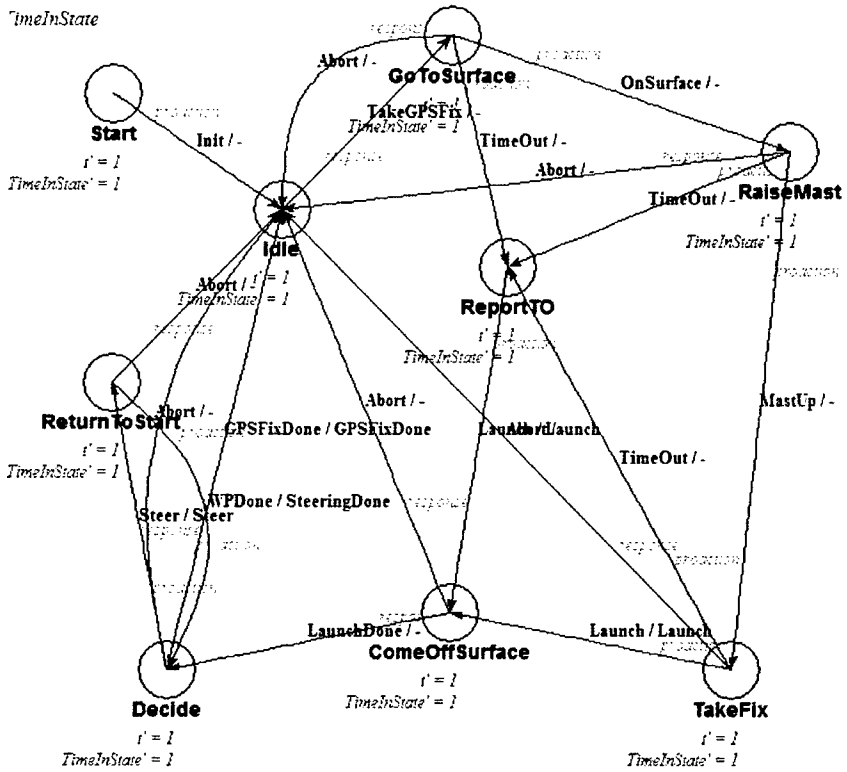

Figure 4: FSM for GPSFixer module

Functions:

ReadParams( ) is a function to read the parameters needed by the GPSFixer module to get executed successfully.

Constructor:

Nav=p_navstate; (initialzed to point to NavState)
DevState=p_devicestate; (initialzed to point to DeviceState)
AutCmd=p_autopilotcmd; (initialzed to point to AutopilotCmd)
DevCmd=p_devicecmd; (initialzed to point to DeviceCmd)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
ActReq=p_actionrequest; (initialzed to point to ActionRequest)
Logs=p_files; (initialzed to point to Files)
Components=p_componentlist; (initialzed to point to ComponentList)
Helm=p_steering; (initialzed to point to Steering)
NavMemory=NavState_new(teja_default_memory_space_id); (initialzed to point to NavState)
NumFailed=0; (initialzed to 0)

Destructor: No destructors.

Continuous state: t, TimeInState

Discrete States:
State 1 {***Start***, t'=1, TimeInState'=1}
State 2 {***Idle***, t'=1, TimeInState'=1}
State 3 {***GoToSurface***, t'=1, TimeInState'=1}
State 4 {***RaiseMast***, t'=1, TimeInState'=1}
State 5 {***ReportTo***, t'=1, TimeInState'=1}
State 6 {***TakeFix***, t'=1, TimeInState'=1}
State 7 {***ComeOffSurface***, t'=1, TimeInState'=1}
State 8 {***ReturnToStart***, t'=1, TimeInState'=1}
State 9 {***Decide***, t'=1, TimeInState'=1}
State 10 {***Error***, t'=1, TimeInState'=1}
State 11 {***Stop***, t'=1, TimeInState'=1}

Tranisitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, ReadParams}
Transition 2 {***Idle, GoToSurface***, TakeGPSFix, True, (t=0, TimeInState=0), (Turning off prop and blowing tanks and performing operation according to VehCmd, DevCmd, AutCmd, ActReq)}
Transition 3 {***GoToSurface, Idle***, Abort, True, none, (Aborting and going to surface and print to errorlog (teja_get_time(),GPSFixer_get_t()), finally flush errorlog) }
Transition 4 {***RaiseMast, Idle***, Abort, True, none, (Aborting raising mast and print to errorlog (teja_get_time(),GPSFixer_get_t()), finally flush errorlog) }
Transition 5 {***TakeFix, Idle***, Abort, True, none, (Aborting TakeFix and print to errorlog (teja_get_time(),GPSFixer_get_t()), finally flush errorlog) }
Transition 6 {***ComeOffSurface, Idle***, Abort, True, none, (Aborting ComeOffSurface and print to errorlog (teja_get_time(),GPSFixer_get_t()), finally flush errorlog) }
Transition 7 {***Decide, Idle***, Abort, True, TimeInState=0, (Aborting Decide and print to errorlog (teja_get_time(),GPSFixer_get_t()), finally flush errorlog) }
Transition 8 {***ReturnToStart, Idle***, Abort, True, none, (Aborting ReturnToStart and print to errorlog (teja_get_time(),GPSFixer_get_t()), finally flush errorlog) }
Transition 9 {***Decide, Idle***, GPSFixDone, GPSOrd=!ReturnToStart,TimeInState=0, (GPSFixDone and print to execlog (teja_get_time(),GPSFixer_get_t()), finally flush execlog) }
Transition 10 {***GoToSurface, ReportTo***, Timeout, TimeInState> GoToSurface, TimeInState=0, Print to file errorlog GoToSurface Timed Out teja_get_time(), GPSFixer_get_t()}
Transition 11 {***GoToSurface, RaiseMast,*** OnSurface, Depth <= SurfaceThreshold, TimeInState=0, Print to execlog teja_get_time(),GPSFixer_get_t() and finally fflush Execlog and excute VehCmd, DevCmd, ActReq)}
Transition 12 {***RaiseMast, ReportTo,*** TimeOut, TimeInState>= RaiseMastTo, TimeInState=0, Print to file errorlog RaiseMast Timed out teja_get_time(), GPSFixer_get_t()) and finally fflush errorlog}
Transition 13 {***RaiseMast, TakeFix***, MastUp, MastState=Up, TimeInState=0, execute VehCmd, ActReq and print to the file execlog GPSFixer - Mast up, waiting for GPS Fix,

teja_get_time(),GPSFixer_get_t() and fflush execlog)}
Transition 14 {***TakeFix, ReportTo,*** TimeOut, (TimeInState>= TakeFixTo), TimeInState=0, Print TakeFix Timed Out, teja_get_time(),GPSFixer_get_t() in file errorlog finally fflush errorlog }
Transition 15 {***TakeFix, ComeOffsurface***, Launch, DevState->GPSFixState == DONE, TimeInState=0, Print Got GPS Fix, teja_get_time(),GPSFixer_get_t() into file execlog and finally fflush execlog) }
Transition 16 {***ReportTo, ComeOfsurface,*** Launch, True, None, (Print Time out, GPS Fix Done, teja_get_time(),GPSFixer_get_t() in file errorlog and finally fflush errorlog)}
Transition 17 {C***omeOfSurface, Decide***, LaunchDone, True, TimeInState=0, None}
Transition 18 {***Decide, ReturnToStart***, Steer, GPSOrd->ReturnToStart, TimeInState=0, Print ReturningToStartPt, teja_get_time(),GPSFixer_get_t() into file execlog and finally fflush execlog and executes VehCmd, AutCmd,DevCmd, ActReq commands) }
Transition 19 {***ReturnToStart, Decide***, WPDone, DistanceToPoint<= WPThresholdDistance, TimeInState=0, (Print at Start Point, teja_get_time(), GPSFixer_get_t() in file execlog, finally fflush execlog and GPSOrd->ReturnToStart = FALSE)}
Transition 20 {***Error, Stop***, Error, True, None, None}

Launcher

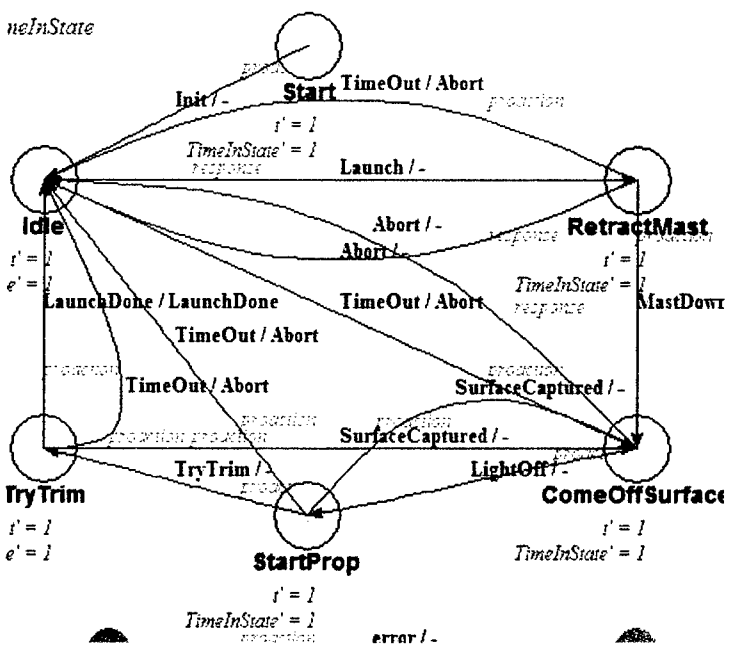

Figure 5: FSM of Launcher module

Superclass: TejaComponent

Variables: RetractMastTo, ComeOffSurfaceTo, LightOfDepth, FwdLaunchMast, AftlaunchMast.

Links: DevState (DeviceState), VehState(VehicleState), DevCmd(DeviceCmd), VehCmd(VehicleCmd), AutCmd(AutopilotCmd), ActReq(Actionreq), Logs(Files), Components(ComponentList), LaunchOrd(LaunchOrder), Nav(NavState).

Functions:
ReadParams( ) is a function to read the parameters needed by the Launcher module to get executed successfully.

Constructors: The data structures are initialized to point to their respective data structures.
DevState=p_devicestate; (initialzed to point to DeviceState)
VehState=p_vehiclestate; (initialzed to point to VehicleState)
DevCmd=p_devicecmd; (initialzed to point to DeviceCmd)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
AutCmd=p_autopilotcmd; (initialzed to point to AutopilotCmd)
ActReq=p_actionrequest; (initialzed to point to ActionReq)
Logs=p_files; (initialzed to point to Files)
Components=p_componentlist; (initialzed to point to ComponentList)
Nav=p_navstate; (initialzed to point to NavState)

Destructor: No destructors.

Continuous state: t, TimeInState.

Discrete States:
State 1 {***Start***, t'=1, TimeInState'=1}
State 2 {***Idle***, t'=1, TimeInState'=1}
State 3 {***RetractMast***, t'=1, TimeInState'=1}
State 4 {***ComOffSurface***, t'=1, TimeInState'=1}
State 5 {***StartProp***, t'=1, TimeInState'=1}
State 6 {***TryTrim***, t'=1, TimeInState'=1}
State 7 {***Error***, t'=1, TimeInState'=1}
State 8 {***Stop***, t'=1, TimeInState'=1}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, ReadParams}
Transition 2 {***RetractMast, Idle***, TimeOut, TimeInState >=RetractMastTo, none, (Print in file errorlog Launch - Retract Mast Timed Out - Aborting Mission, teja_get_time(), Launcher_get_t() and finally fflush errorlog)}
Transition 3 {***Idle, RetractMast,*** Launch, True, (t=0, TimeInstate=0), (Print in file execlog Launch - Retracting Mast, teja_get_time(), Launcher_get_t() and finally fflush execlog and execute commands by VehCmd, DevCmd, DevState, ActReq)}
Transition 4 {***ComeOffSurface, Idle,*** Abort, True, (Print in file errorlog Launch - Aborting ComeOffSurface on signal, teja_get_time(), Launcher_get_t() and finally fflush errorlog)}

Transition 5 {***RetractMast, Idle,*** Abort, True, (Print in file errorlog Launch - Aborting RetractMast on signal, teja_get_time(), Launcher_get_t() and finally fflush errorlog)}
Transition 6 {***ComeOffSurface, Idle,*** TimeOut, TimeInState>=ComeOffSurfaceTo, (Print in file errorlog Launch - ComeOffSurface Timed Out - Aborting Mission, teja_get_time(), Launcher_get_t() fflush errorlog)}
Transition 7 {***TryTrim, Idle,*** LaunchDone, (Speed > 1.5 && Launcher_get_TimeInState() > 60.0), None, (Print in file execlog Launch - LaunchDone, teja_get_time(),Launcher_get_t() and finally fflush execlog)}
Transition 8 {**RetractMast, ComeOffSurface**, MastDown, True, TimeInState=0, (Print to file execlog Launch - Mast Down, coming off surface, teja_get_time(), Launcher_get_t() and finally fflush execlog and (Altitude >20 LightOffdepth= 10 or Altitude >10 LightOffDepth=5 or LightOffDepth=5) and executes VehCmd, DevCmd and ActReq)}
Transition 9 {***ComeOffSurface, StartProp,*** LightOff, Depth >= LightOffDepth, (Print in file execlog Launch - Lighting-off prop, teja_get_time(), Launcher_get_t() and finally fflush execlog and execute commands from VehCmd, AutCmd, ActReq) }
Transition 10 {***StartProp, ComeOffSurface,*** SurfaceCaptured, Depth<1.5, TimeInState=0, (Print to file execlog Launch - Surface Captured - trying to come off surface, teja_get_time(), Launcher_get_t() and finally fflush execlog and (Altitude >20 LightOffdepth= 10 or Altitude >10 LightOffDepth=5 or LightOffDepth=5) and executes VehCmd, DevCmd and ActReq)}
Transition 11 {***TryTrim, ComeOffSurface***, SurfaceCaptured, Depth<1.5, TimeInState=0, (Print to file execlog Launch - Surface Captured - trying to come off surface, teja_get_time(), Launcher_get_t() and finally fflush execlog and (Altitude >20 LightOffdepth= 10 or Altitude >10 LightOffDepth=5 or LightOffDepth=5) and executes VehCmd, DevCmd and ActReq)}
Transition 12 {***StartProp, TryTrim,*** TryTrim, Speed>1, TimeInState=0, (Print in file execlog Launch - Trying VBS Trim to get fin authority, teja_get_time(), Launcher_get_t() and finally fflush execlog and execute commands from VehCmd, DevCmd, ActReq)}
Transition 13 {***Error, Stop***, Error, True, None, None}

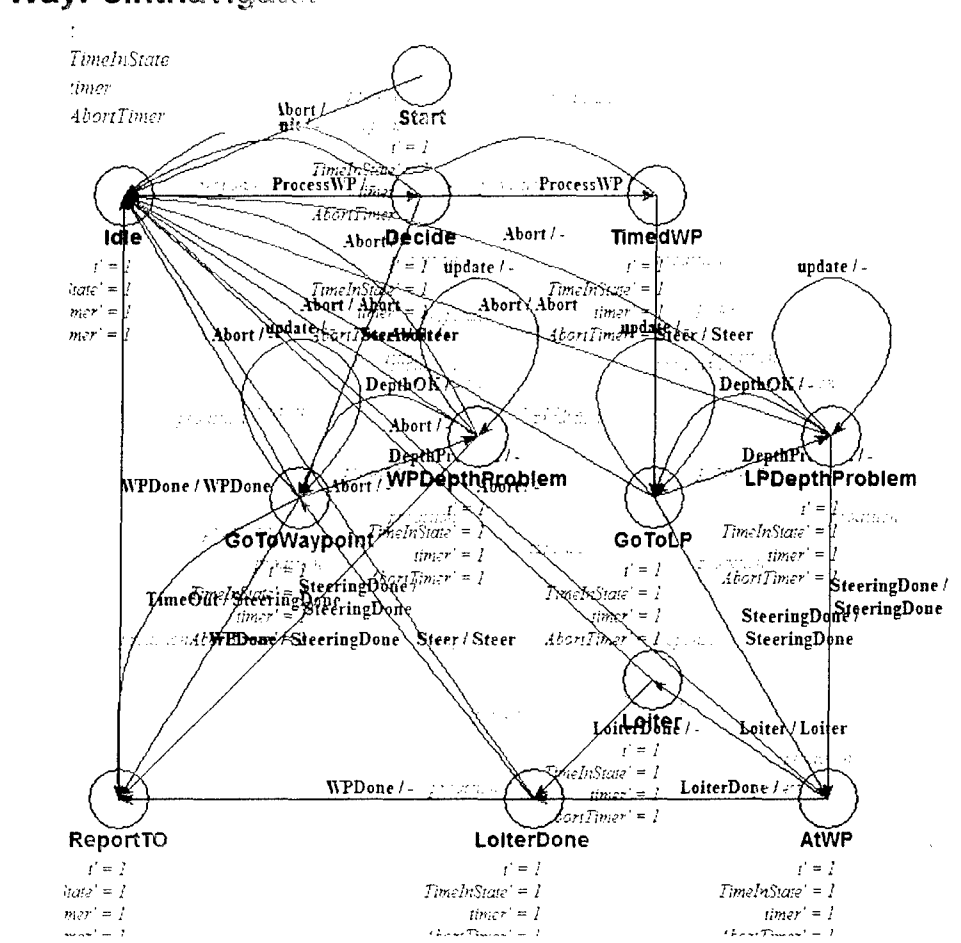

Figure 6: FSM of Waypointnavigator

Superclass: TejaComponent

Variables: WayPointPctOverrun, WayPointTo, ThresholdDistance, MinSpeed, MaxSpeed, TimeToWayPoint, DistanceToWayPoint, LoiterDistance, LoiterAwayDistance, Requestor.

Links: Nav (NavState), ToWP (WayPoints), AutCmd (AutopilotCmd), ActReq (ActionReq), DevCmd (DeviceCmd), VehCmd (VehicleCmd), FromWP (Wayoints), Logs (Files), Helm (Steering), MC (Controller), Vagabond (Loiter), Components (ComponentList)

Functions:
ReadParams( ) is a function to read the parameters needed by the WayPointNavigator module to get executed successfully.

Constructors: The data structures are initialized to point to their respective data structures.

Nav=p_navstate; (initialzed to point to NavState)
AutCmd=p_autopilotcmd; (initialzed to point to AutopilotCmd)
ActReq=p_actionrequest; (initialzed to point to ActionReq)
DevCmd=p_devicecmd; (initialzed to point to DevCmd)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
Helm=p_steering; (initialzed to point to Steering)
MC=p_controller; (initialzed to point to Controller)
Logs=p_files; (initialzed to point to Files)
Vagabond=p_loiter; (initialzed to point to Loiter)
Components=p_componentlist; (initialzed to point to ComponetList)
ToWP=Waypoints_new(teja_default_memory_space_id); (initialzed to point to WayPoint)
FromWP=Waypoints_new(teja_default_memory_space_id); (initialzed to point to WayPoint)

Destructor: No destructors.

Continuous state: t, TimeInState, timer

Discrete States:
State 1 {***Start***, t'=1, TimeInState'=1, timer'=1}
State 2 {***Idle***, t'=1, TimeInState'=1, timer'=1}
State 3 {***Decide***, t'=1, TimeInState'=1, timer'=1}
State 4 {***TimedWP***, t'=1, TimeInState'=1, timer'=1}
State 5 {***GoToWayPoint***, t'=1, TimeInState'=1, timer'=1}
State 6 {***GoToWP***, t'=1, TimeInState'=1, timer'=1}
State 7 {***Loiter***, t'=1, TimeInState'=1, timer'=1}
State 8 {***AtWP***, t'=1, TimeInState'=1, timer'=1}
State 9 {***LoiterDone***, t'=1, TimeInState'=1, timer'=1}
State 10 { ***ReportTo***, t'=1, TimeInState'=1, timer'=1}
State 11 {***Error***, t'=1, TimeInState'=1, timer'=1}
State 12 {***Stop***, none}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, (ReadParams, LoiterAwayDistance < 500.0) { LoiterAwayDistance=500.0, Print in file errorlog WaypointNavigator - LoiterAwayDistance too small, resetting to 500 m, teja_get_time(), WaypointNavigator_get_t() and finally fflush errorlog)}
Transition 2 {***Idle, Decide***, ProcessWP, True, t=0, Calculate the distance to cover}
Transition 3 {***ReportTo, Idle,*** WPDone, True, TimeInState=0, None (Output)}
Transition 4 {***Decide, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}

Transition 5 {***TimedWP, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}
Transition 6 {***GoToWayPoint, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}
Transition 7 {***GoToWP, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}
Transition 8 {***AtWP, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}
Transition 9 {***Loiter, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}
Transition 10 {***LoiterDone, Idle***, Abort, True, None, (Print in file errorlog WaypointNavigator - Aborting on signal, teja_get_time(), WaypointNavigator_get_t() finally fflush errorlog)}
Transition 11 {***Decide, TimedWP***, ProcessWP, (ToWP->Timed && TimeToWaypoint > DistanceToWaypoint/MinSpeed), None, Loiters to desired point with desired Loiter type}
Transition 12 {***Decide, GoToWayPoint***, Steer, (!ToWP->Timed || TimeToWaypoint <= DistanceToWaypoint/MinSpeed), (TimeInState=0, timer=0), Goes to desired latitude and longitude with desired steer mode}
Transition 13 {***TimedWP, GoToWP,*** True, timer=0, (Goes to desired latitude and longitude with desired SteerMode, HeadingMode, DepthMode)}
Transition 14 {***GoToWP, GoToWP,*** Update, timer>=1, timer=0, Calculate the distance to desired waypoint}
Transition 15 {***GoToWP, AtWP,*** Abort, (Helm->DistanceToPoint <= LoiterDistance) || (TimeToWaypoint-WaypointNavigator_get_t() <= DistanceToWaypoint/MinSpeed), None, Calculates by how much the vehicle loiters away from the desired waypoint }
Transition 16 {***AtWP, Loiter***, Loiter, ToWP->LoiterDuration>0, None, (Print in file execlog WaypointNavigator - Going to loiter, teja_get_time(), WaypointNavigator_get_t() fflush execlog) }
Transition 17 {***AtWP, LoiterDone,*** LoiterDone, ToWP->LoiterDuration<=0, None, (Print in file errorlog WaypointNavigator - No time to loiter , teja_get_time(), WaypointNavigator_get_t() fflush errorlog)}
Transition 18 {***Loiter, LoiterDone,*** LoiterDone, True, None, None}
Transition 19 {***LoiterDone,GoToWayPoint***, Steer, (!ToWP->LoiterAtWP && TimeToWaypoint >= WaypointNavigator_get_t()), (TimeInState=0, timer=0), Calculate the distance to desired latitude and longitude with desired speedmode}
Transition 20 {***LoiterDone, ReportTo,*** WPDone, (ToWP->LoiterAtWP || (TimeToWaypoint <= WaypointNavigator_get_t())), None, None}
Transition 21 {***GoToWayPoint, GoToWayPoint,*** Update, timer>=2, timer=0, Calculates the time to go to the desired location with the desired speedmode}
Transition 22 { ***GoToWayPoint, ReportTo***, WPDone, ((!ToWP->Timed && Helm->DistanceToPoint <= ThresholdDistance) || (ToWP->Timed && Helm-

>DistanceToPoint <= 5.0)), TimeInState=0, (Print in file execlog WaypointNavigator - WP Done, teja_get_time(), WaypointNavigator_get_t() and finally fflush execlog)}
Transition 23 { ***GoToWayPoint, ReportTo***, TimeOut, TimeInState>=WayPointTo, (Print in file errorlog WaypointNavigator - Waypoint Timed Out, teja_get_time(), WaypointNavigator_get_t() and finally fflush errorlog)}
Transition 24 {***ReportTo, Idle,*** WPDone, True, TimeInstate=0, None}
Transition 25 {***Error, Stop***, Error, True, None, None}

Rendezvous

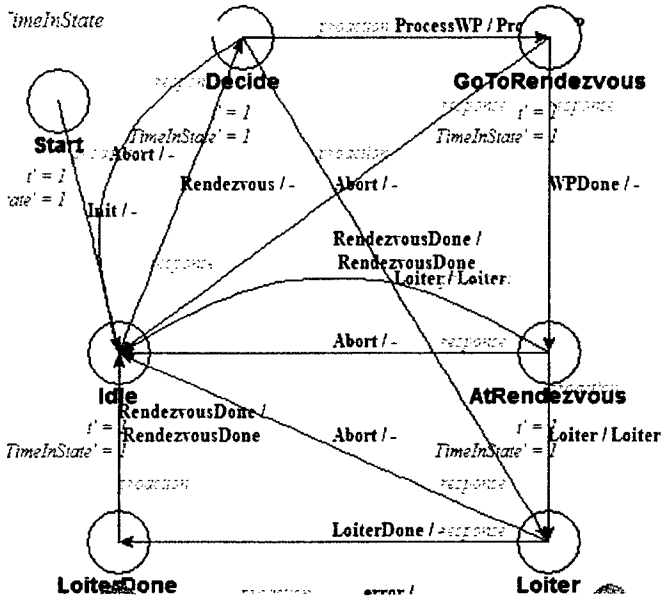

Figure 7: FSM of Rendezvous module

Superclass: TejaComponent

Variables: WPThreshDistMemory

Links: ActReq, Logs, Vagabond, WPNav, DevCmd, VehCmd, Components

Functions: None

Constructors:
ActReq=p_actionrequest; (initialzed to point to Actionreq)
Logs=p_files; (initialzed to point to Files)
Vagabond=p_loiter; (initialzed to point to Loiter)
WPNav=p_waypointnavigator; (initialzed to point to WayPointNavigator)
DevCmd=p_devicecmd; (initialzed to point to DeviceCmd)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
Components=p_componentlist; (initialzed to point to ComponentList)

Destructor: No destructors.

Continuous state: t, TimeInState
Discrete States:
State 1 {***Start***, t'=1, TimeInState'=1}
State 2 {***Idle***, t'=1, TimeInState'=1}
State 3 {***LoiterDone***, t'=1, TimeInState'=1}
State 4 {***Decide***, t'=1, TimeInState'=1}
State 5 {***GoToRendezvous***, t'=1, TimeInState'=1}
State 6 {***AtRendouzvous***, t'=1, TimeInState'=1}
State 7 {***Loiter***, t'=1, TimeInState'=1}
State 8 {***Error***, t'=1, TimeInState'=1}
State 9 {***Stop***, none}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, None, None}
Transition 2 {***Decide, Idle***, Abort, True, None, (Print in file errorlog Rendezvous - Aborting, teja_get_time(),Rendezvous_get_t() and finally fflush errorlog)}
Transition 3 {***Idle, Decide,*** Rendezvous, True, (t=0, TimeInState=0), Calculates the Way point threshold distance depending upon the loiter type used for movement}
Transition 4 {***GoToRendezvous, Idle,*** Abort, True, None, (WPNav->ThresholdDistance=WPThreshDistMemory, Print in file errorlog,Rendezvous - Aborting, teja_get_time(), Rendezvous_get_t(), and finally fflush errorlog)}
Transition 5 {***AtRendezvous, Idle***, RendezvousDone, (WPNav->ToWP->LoiterType==NONE || WPNav->ToWP->LoiterDuration==0.0), None, (WPNav->ThresholdDistance= WPThreshDistMemory, Print in file execlog Rendezvous - No loiter specified, leaving rendezvous, teja_get_time(), Rendezvous_get_t() and finally fflush execlog)}
Transition 6 {***AtRendezvous, Idle,*** Abort, True, None, (WPNav->ThresholdDistance= WPThreshDistMemory, Print to file errorlog: Rendezvous â€“ Aborting, teja_get_time(), Rendezvous_get_t() and finally fflush errorlog)}
Transition 7 {***Loiter, Idle,*** Abort, True, None, (WPNav->ThresholdDistance= WPThreshDistMemory, Print to file errorlog: Rendezvous â€“ Aborting, teja_get_time(), Rendezvous_get_t() and finally fflush errorlog)}
Transition 8 {***LoiterDone, Idle,*** RendezvousDone, True, TimeInState=0, (WPNav->ThresholdDistance= WPThreshDistMemory, Print to file execlog: Rendezvous - Leaving rendezvous, teja_get_time(), Rendezvous_get_t() and finally fflush execlog)}
Transition 9 {***Decide, GoToRendezvous***, ProcessWP, True, None, executes commands given by VehCmd, DevCmd, ActReq}
Transition 10 {***Decide, Loiter***, Loiter, (WPNav->ToWP->Latitude==0 && WPNav->ToWP->Longitude==0 && WPNav->ToWP->Depth==0 && WPNav->ToWP->Speed==0), TimeInState=0, Updates various loiter variables and goes of to loiter when specified rendezvous point is not given}
Transition 11 {***GoToRendezvous, AtRendezvous***, WPDone, True, None, None}

Transition 12 {***AtRendezvous, Loiter***, Loiter, True, TimeInState=0, (Print in file excelog Rendezvous - Going to Loiter, teja_get_time(),Rendezvous_get_t(), and finally fflush execlog and loiter to different locations)}
Transition 13 {***Loiter, LoiterDone***, LoiterDone, True, None, None}
Transition 14 {***Error, Stop***, Error, True, None, None}

DeviceCommander

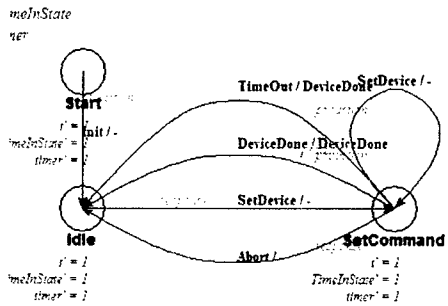

Figure 8: FSM for DeviceCommander

Superclass: TejaComponent
Variables: CmdSet, GoSurfaceTo, RaiseMastTo, SurfaceThreshold, ComeOffSurfaceTo, RetractMastTo, SetSwitchTo

Links: Components (ComponentList), Logs (Files), DevOrd (DeviceOrd),DevCmd (DeviceCmd), DevState (DeviceState), ActReq (ActionReq), VehCmd (VehicleCmd), Nav (NavState), InitDevState (DeviceState), AutCmd (AutopilotCmd), InitNav (NavState)

Functions:
SetDeviceCmd( ) is a function to set the device command specified in the current device command order.
DeviceCmdDone( ) is a function to check if the current device command is completed
CheckDeviceCmd( ) is a function which checks if it is safe to apply current device command
ReadParams( ) is a function to read the parameters needed by the DeviceCommander module to get executed successfully.
TimeOut( ) is a function to check if a time out for the current device has occurred

Constructors:
Components=p_componentlist; (initialzed to point to ComponentList)
Logs=p_files; (initialzed to point to Files)
DevCmd=p_devicecmd; (initialzed to point to DeviceCmd)
DevState=p_devicestate; (initialzed to point to DeviceState)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
ActReq=p_actionrequest; (initialzed to point to ActionReq)
Nav=p_navstate; (initialzed to point to NavState)

AutCmd=p_autopilotcmd; (initialzed to point to AutopilotCmd)
SetSwitchTO=5.0;
InitDevState=DeviceState_new(teja_default_memory_space_id); (initializing memory space for DeviceState)
InitNav=NavState_new(teja_default_memory_space_id); (initializing memory space for NavState)

Destructor: No destructors.

Continuous state: t, TimeInState, timer

Discrete States:
State 1 {***Start***, t'=1, TimeInState'=1, timer'=1}
State 2 {***Idle***, t'=1, TimeInState'=1, timer'=1}
State 3 {***SetCommand***, t'=1, TimeInState'=1, timer'=1}
State 4 {***Error***, t'=1, timer'=1}
State 5 {***Stop***, none}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, ReadParams}
Transition 2 {***Idle, SetCommand***, SetDevice, True, t=0, TimeInState=0, timer=0, (Saves initial device states,saves initial nav state, (CmdSet=True, Device Commander - Setting Device Commands) (CmdSet= False, Device Commander - Waiting to set Device Commands))}
Transition 3 {***SetCommand, Idle***, DeviceDone, (DeviceCommander_DeviceCmdDone() && CmdSet), TimeInState=0, (Print Commander - Device command done, teja_get_time(),DeviceCommander_get_t())}
Transition 4 {***SetCommand, Idle***, TimeOut, DeviceCommander_TimeOut(this), None, None)
Transition 5 {***SetCommand, Idle***, Abort, True, None,(Print to file errorlog DeviceCommander - Aborting set device command, teja_get_time(), DeviceCommander_get_t() and finally fflush errorlog)
Transition 6 {***SetCommand, SetCommand,*** SetDevice, (DeviceCommander_get_timer() >=1 && !CmdSet), timer=0, (DeviceCommander_CheckDeviceCmd
DeviceCommander_SetDeviceCmd() CmdSet=TRUE, Print to file execlog Device Commander - Setting Device Commands, teja_get_time(), DeviceCommander_get_t())}
Transition 7 {***Error, Stop***, Error, True, None, None}

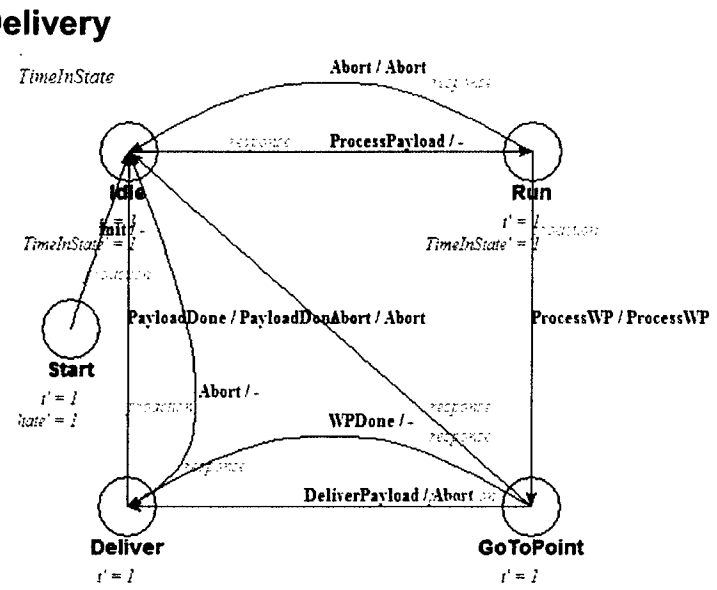

Figure 9: FSM of Payload module

Superclass: TejaComponent

Variables: DeliveryDelay, PayloadDescription, WPDistThreshMemory

Links: Logs, WPNav, VehCmd, ActReq, DevCmd, AutCmd, Components

Functions:
ReadParams( ) is a function to read the parameters needed by the PayloadDelivery module to get executed successfully.

Constructors:
AutCmd=p_autopilotcmd; (initialzed to point to AutopilotCmd)
WPNav=p_waypointnavigator; (initialzed to point to WayPointNavigator)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
ActReq=p_actionrequest; (initialzed to point to ActionReq)
DevCmd=p_devicecommand; (initialzed to point to DeviceCommand)
Logs=p_files; (initialzed to point to Files)
Components=p_componentlist; (initialzed to point to ComoponentList)

Destructor: No destructors.

Continuous state: t, TimeInState

Discrete states:
State 1 {***Start***, t'=1, TimeInState'=1}
State 2 {***Idle***, t'=1, TimeInState'=1}
State 3 {***Run***, t'=1, TimeInState'=1}
State 4 {***Deliver***, t'=1, TimeInState'=1}

State 5 {***GoToPoint***, t'=1, TimeInState'=1}
State 6 {***Error***, t'=1, timer'=1}
State 7 {***Stop***, none}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, ReadParams}
Transition 2 {***Deliver, Idle,*** PayloadDone, TimeInState>=DeliveryDelay, TimeInState=0, (WPNav->ThresholdDistance=WPDistThreshMemory and executes commands DevCmd, Acteq and prints to file execlog Payload - Payload Delivery Done, teja_get_time(), PayloadDelivery_get_t() and finally fflush execlog)}
Transition 3 {***Idle, Run***, ProcessPayload, True, (t=0, TimeInState=0), (WPDistThreshMemory=WPNav->ThresholdDistance)}
Transition 4 {***Run, Idle,*** True, None, (Print to file errorlog PayloadDelivery - Aborting, teja_get_time(), PayloadDelivery_get_t(), and finally fflush errorlog)}
Transition 5 {***GoToPoint, Idle***, Abort, True, None, (print to errorlog PayloadDelivery - Aborting, teja_get_time(), PayloadDelivery_get_t() and finally fflush errorlog)}
Transition 6 {***Deliver, Idle***, Abort, True, (print to errorlog PayloadDelivery - Aborting, teja_get_time(), PayloadDelivery_get_t() and finally fflush errorlog)}
Transition 7 {***Run, GoToPoint***, ProcessWP, True, None, (WPNav->ThresholdDistance=5.0 print to file execlog Payload - Proceeding to Payload Delivery Point, teja_get_time(), PayloadDelivery_get_t() and finally fflush execlog and commands in VehCmd, DevCmd, ActReq are operated)}
Transition 8 {***GoToPoint, Deliver***, DeliverPayload, (WPNav->TimeToWaypoint <= DeliveryDelay), TimeInState=0, (execute commands VehCmd, DevCmd, ActReq, PayloadDescription==PORT print to file execlog Payload - Deliver port payload, teja_get_time(),PayloadDelivery_get_t() or PayloadDescription==STBD print to file execlog Payload - Deliver port payload, teja_get_time(),PayloadDelivery_get_t() or PayloadDescription==BOTH print to file execlog Payload - Deliver port payload, teja_get_time(),PayloadDelivery_get_t() and finally fflush execlog)}
Transition 9 {***GoToPoint, Deliver***, WPDone, True, TimeInState=0, (execute commands VehCmd, DevCmd, ActReq, PayloadDescription==PORT print to file execlog Payload - Deliver port payload, teja_get_time(),PayloadDelivery_get_t() or PayloadDescription==STBD print to file execlog Payload - Deliver port payload, teja_get_time(),PayloadDelivery_get_t() or PayloadDescription==BOTH print to file execlog Payload - Deliver port payload, teja_get_time(),PayloadDelivery_get_t() and finally fflush execlog)}
Transition 10 {***Error, Stop***, Error, True, None, None}

Loiter

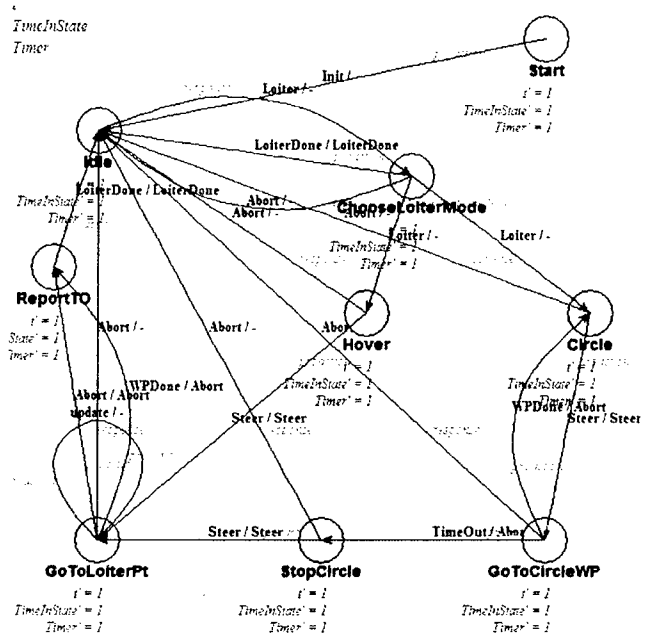

Figure 10: FSM of Loiter module

Superclass: TejaComponent

Variables: LoiterTo, Radius, LoiterLat, LoiterLon, LoiterDepth, LoiterSpeed, CircleLats, CircleLons, NumPoints, CurrentWP, LoiterSpeedMode, MinSpeed, MaxSpeed, Requestor

Links: Logs, Nav, WPNav, ActReq, VehCmd, AutCmd, DevCmd, Helm, Components

Functions:
ReadParams( ) is a function to read the parameters needed by the Loiter module to get executed successfully.
PlotCircle( ) is the function which finds the latitudes and longitudes of points that define a circle about the Loiter Point.
TimeToLoiterPoint( ) is a function that calculates time to loiter point.

Constructors:
Logs=p_files; (initialzed to point to Files)
Nav=p_navstate; (initialzed to point to NavState)
WPNav=p_waypointnavigator; (initialized to point to WayPointNavigator)
ActReq=p_actionrequest; (initialzed to point to ActionRequest)
VehCmd=p_vehiclecmd; (initialzed to point to VehicleCmd)
AutCmd=p_autopilotcmd; (initialzed to point to AutopilotCmd)
DevCmd=p_devicecmd; (initialzed to point to DeviceCmd)
Helm=p_steering; (initialzed to point to Steering)
Components=p_componentlist; (initialzed to point to ComponentList)

LoiterSpeedMode=OPENLOOP;
CurrentWP=0;

Destructor: No destructors.

Continuous state: t, TimeInState, timer

Discrete states:
State 1 {***Start***, t'=1, TimeInState'=1, timer'=1}
State 2 {***Idle***, t'=1, TimeInState'=1, timer'=1}
State 3 {***ReportTo***, t'=1, TimeInState'=1, timer'=1}
State 4 {***CooseLoiterMode***, t'=1, TimeInState'=1, timer'=1}
State 5 {***Hover***, t'=1, TimeInState'=1, timer'=1}
State 6 {***Circle***, t'=1, TimeInState'=1, timer'=1}
State 7 {***GoToCircleWP***, t'=1, TimeInState'=1, timer'=1}
State 8 {***StopCircle***, t'=1, TimeInState'=1, timer'=1}
State 9 {***GoToLoiterPt***, t'=1, TimeInState'=1, timer'=1}
State 10 {***Error***, t'=1, TimeInState'=1, timer'=1}
State 11 {***Stop***, none}

Transitions:
Transition 1 {***Start, Idle***, Init, t>=1, none, ReadParams}
Transition 2 {***Idle,ChooseLoiterMode,*** Loiter, True, t=0, None}
Transition 3 {***ChooseLoiteMode, Idle***, LoiterMode, (NumPoints<2 && LoiterTO==0) || WPNav->ToWP->LoiterType==NONE), None, (Print to file execlog Loiter - No loiter required....Ending Loiter, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 4 {***ChooseLoiteMode, Idle***, Abort, True, None, (Print to file execlog Loiter - Aborting from Loiter on abort signal, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 5 {***Hover, Idle***, Abort, True, (t=0, TimeInState=0), (Print to file execlog Loiter - Aborting from Loiter on abort signal, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 6 {***Circle, Idle***, Abort, True, (t=0, TimeInState=0), (Print to file execlog Loiter - Aborting from Loiter on abort signal, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 7 {***GoToCircleWP, Idle***, Abort, True, None, (Print to file execlog Loiter - Aborting from Loiter on abort signal, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 8 {***StopCircle, Idle***, Abort, True, (t=0, TimeInState=0), (Print to file execlog Loiter - Aborting from Loiter on abort signal, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 9 {***GoToLoiterPt, Idle***, Abort, True, (t=0, TimeInState=0), (Print to file execlog Loiter - Aborting from Loiter on abort signal, teja_get_time(), Loiter_get_t() and finally fflush execlog)}
Transition 10 {***ReportTo, Idle***, LoiterDone, True, None, None}

Transition 11 {***ChooseLoiterMode, Hover***, Loiter, (NumPoints<2 && LoiterTO>0) || WPNav->ToWP->LoiterType==HOVER), TimeInState=0, (Print to file execlog Loiter - Setting VBS in Hover mode, teja_get_time(),Loiter_get_t() and finally fflush execlog and execute commands in VehCmd, AutCmd, DevCmd, ActReq)}
Transition 12 {***ChooseLoiterMode, Circle***, Loiter, (NumPoints>=2 && LoiterTO>0 && WPNav->ToWP->LoiterType==CIRCLE), TimeInState=0, (Print to file execlog Loiter - Computing Loiter Circle and going to first point, teja_get_time(),Loiter_get_t() and finally fflush execlog and execute commands in VehCmd, AutCmd, ActReq and Loiter_PlotCircle)
Transition 13 {***Circle, GoToCircleWP***, Steer, NumPoints>=2, (WPNav->ToWP->UseSSS) (Print to execlog Loiter - Turning On SSS, teja_get_time(),Loiter_get_t() fflush execlog, DevCmd->SSSCmd=ON) or (DevCmd->SSSCmd=OFF DevCmd->VBSCmd=TRIM, DevCmd->VBSDepthCmd=WPNav->ToWP->Depth)}
Transition 14 {***GoToCircleWP, Circle***, WPDone, Helm->DistanceToPoint<=20.0, None, Loiter - Processing loiter waypoint}
Transition 15 {***Hover, GoToLoiterpt***, (TimeInState>= LoiterTO-Loiter_TimeToLoiterPt() â€“ 20), Timer=0, (Loiter - Turning On SSS or Loiter â€“ returningToLoiterPt)}
Transition 16 {***GoToCircleWP, StopCircle***, TimeOut, (TimeInState >= LoiterTO-Loiter_TimeToLoiterPt(), None, None}
Transition 17 {***StopCircle, GoToLoiterpt***, Steer, True, None, Loiter â€“ Returning

Appendix C: Illustration of Verification of logical correctness of the controller modules

We start with the **lowest level** (*i=1*), and pick the **Steering** module first as it receives orders from the others, and only responds to those orders. We develop abstract models of the *commanding* and *commanded* environment, called drivers and stubs. The steering module is shown in Figure C.111, whereas the abstraction of commanding environment the driver module is shown in Figure . There is no module that the steering module commands.

C.1 Verification of Steering module

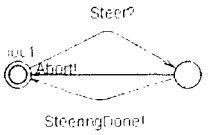

Figure C.111: Steering module in UPPAAL

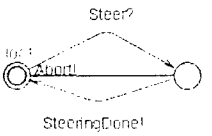

Figure C.2: Driver for steering module in UPPAAL

Queries

The following queries were formulated as temporal logic formulae in order to perform verification of logical correcteness of the steering module.

E[] Steering_P.Idle_ds

/*Eventually in future there always is a path, which goes to the final state (here Idle_ds)*/

A[] Steering_P.SteerToPoint_ds imply Steering_P.timer<=2

/*The steering module always updates the present location of the AUV every 2 seconds when at state SteerToPoint_ds */

A<> Steering_P.SteerToPoint_ds imply Steering_P.Idle_ds

/* Always eventually the vehicle is steered to the desired point */

C.2 Verification of Loiter module

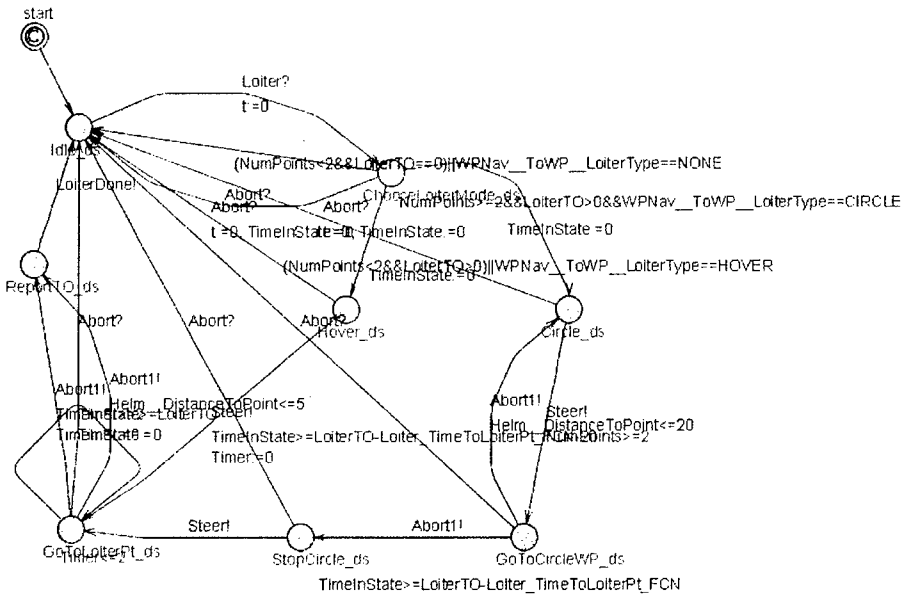

Figure 12: Loiter module module in UPPAAL

The next module selected is the Loiter module at level 1, then its environment is abstracted. The Loiter module is shown in Figure 12, the abstracted commanding environment is shown in Figure 13 and the abstracted commanded environment is shown in Figure 14.

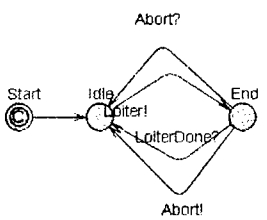

Figure 13: Driver for loiter module module in UPPAAL

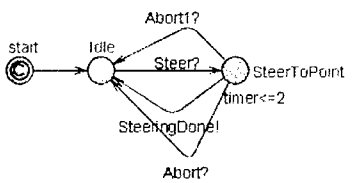

Figure 14: Stub for loiter module module in UPPAAL

Queries formulated as temporal formulas are as shown below.

A<> Loiter_P.Idle_ds

/* All paths eventually lead to the final or end state (here Idle_ds), indicating progress i.e. order is completed as final state is reached or it might be that the mission is aborted but that too signifies progress indicating the detection of failure and no deadlock. */

A<> Loiter_P.ChooseLoiterMode_ds imply (Loiter_P.Hover_ds||Loiter_P.Circle_ds)

/* All paths from the state ChooseLoiterMode eventually either leads to hovering or circling (indicating progress as it should select a mode to loiter). */

A[] Loiter_P.Circle_ds imply (Loiter_P.LoiterTO>0 && Loiter_P.NumPoints>2)&& WPNav__ToWP__LoiterType==Loiter_P.CIRCLE

/* For all paths if "Circling" is selected the guard conditions are LoiterTO>0 and NumPoints>2 (this indicates that the guard conditions required for circling like number of points are taken care of) Changes: numpoints>2 and LoiterTO>0 in global declaration */

A[] Loiter_P.Hover_ds imply (Loiter_P.LoiterTO>0 && Loiter_P.NumPoints>2)|| WPNav__ToWP__LoiterType==Loiter_P.HOVER

/* For all paths always if "Hovering" is selected then the required guard conditions are satisfied. Changes: NumPoints<2 and LoiterTO>0 */

A[] Loiter_P.Circle_ds imply Loiter_P.NumPoints>2

/* For all paths always circling means number of points >2 Changes: NumPoints>=2 */

E<> not Loiter_P.NumPoints>2 and Loiter_P.Hover_ds

/* This statement proves that there doesnt exist a path where eventually NumPoints>2 hold after Hover mode */

E<> (Loiter_P.NumPoints<2 && WPNav__ToWP__LoiterType==Loiter_P.HOVER) imply Loiter_P.Hover_ds

/* Is it possible to reach Hover mode with all parameters for HOVERING mode */

E<> (Loiter_P.NumPoints>2 && WPNav__ToWP__LoiterType==Loiter_P.CIRCLE) and Loiter_P.Hover_ds

/* Is it possible to reach Hover mode with all parameters for Circling mode */

E<> Loiter_P.NumPoints>2 imply Loiter_P.GoToCircleWP_ds

/* Does there exist a path where NumPoints>2 leads to GoToCircleWP_ds (indicating progress of Circle type order being executed correctly) */

E<> (Loiter_P.NumPoints>2 && WPNav__ToWP__LoiterType==Loiter_P.CIRCLE) imply Loiter_P.Circle_ds

/* Is it possible to reach CIRCLE mode with circling mode type input (indicating that logic of guards for progress of executing orders correctly)*/

E<> (Loiter_P.NumPoints<2 && WPNav__ToWP__LoiterType==Loiter_P.HOVER) and Loiter_P.Circle_ds

/* Is it possible to reach CIRCLE mode with hovering mode type input (indicating that guards for progress are given correctly)*/

E<> Loiter_P.TimeInState<=Loiter_P.LoiterTO-Loiter_P.Loiter_TimeToLoiterPt_FCN imply not Loiter_P.StopCircle_ds

/*Is it possible to reach StopCircle_ds when the guard condition leading to that state is satisfied */

E<> (Loiter_P.GoToCircleWP_ds and Helm__DistanceToPoint<=20) imply Loiter_P.Circle_ds

/* Check correct execution: Is it possible to reach to Circle_ds when at GoToCircleWP_ds and guard Helm__DistanceToPoint<=20 is satisfied */

E<> (Loiter_P.Hover_ds || Loiter_P.Circle_ds) imply Loiter_P.GoToLoiterPt_ds

/*Is it possible to reach the GoToLoiterPt_ds state after choosing the mode of loitering */

A<> Loiter_P.ChooseLoiterMode_ds imply Loiter_P.Idle_ds

/* All paths eventually lead to final state from the present state indicating either successful completion or termination of a mission*/

A<> Loiter_P.Hover_ds imply Loiter_P.Idle_ds

/* All paths eventually lead to final state from the present state */

A<> Loiter_P.Circle_ds imply Loiter_P.Idle_ds

/* All paths eventually lead to final state from the present state */

A<> Loiter_P.GoToCircleWP_ds imply Loiter_P.Idle_ds

/*All paths eventually lead to final state from the present state*/

A<> Loiter_P.StopCircle_ds imply Loiter_P.Idle_ds

/*All paths eventually lead to final state from the present state*/

A<> Loiter_P.GoToLoiterPt_ds imply Loiter_P.Idle_ds

/*All paths eventually lead to final state from the present state */

A<> Loiter_P.ReportTO_ds imply Loiter_P.Idle_ds

/*All paths eventually lead to final state from the present state*/

E<> Loiter_P. GoToLoiterPt_ds and Loiter_P.Timer<=2

/*Is the information regarding reaching the loiter point updated every 2 seconds*/

C.3 Verification of GPSFixer module

All the subsystems at level 1 are now verified. So the value of *i* changes to the next level which is 2. The next module selected is the GPSFixer subsystem at level 2. The

environment for GPSFixer susbsystem is now abstracted. The GPSFixer subsystem is shown in Figure 15, the abstracted commanding environment is shown in Figure 16 and the abstracted commanded environment is shown in Figure 17.

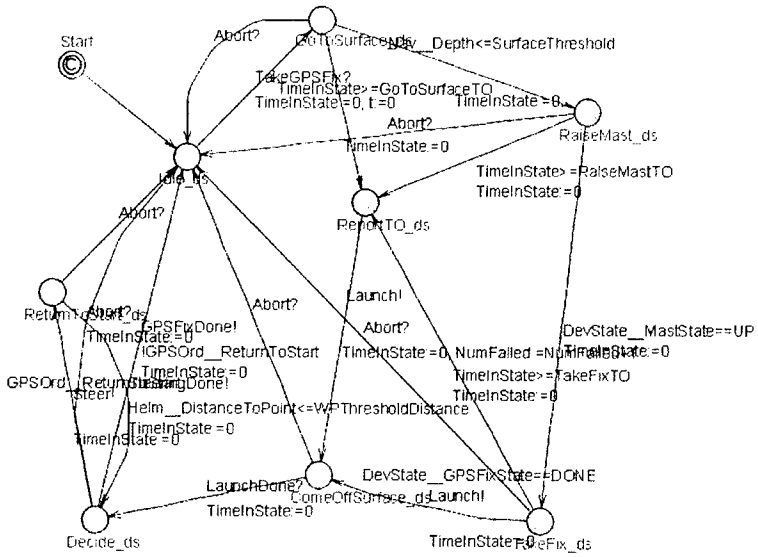

Figure 15: GPSFixer module in UPPAAL

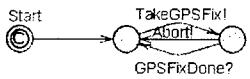

Figure 16: Driver for GPSFixer module in UPPAAL

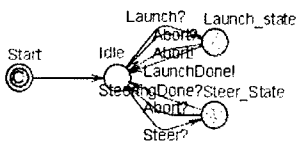

Figure 17: Stub for GPSFixer module in UPPAAL

Queries:

A<> GPSFixer_P.Idle_ds

/* All paths eventually lead to the final state (All paths not always lead to final state as ReportTO_ds doesnt connect to the final state but it eventually goes to the final state).*/

E<> (GPSFixer_P.GoToSurface_ds and Nav__Depth<=GPSFixer_P.SurfaceThreshold) imply GPSFixer_P.RaiseMast_ds

/*Is it possible to reach the next state (RaiseMast_ds) when its at present state GoToSurface_ds and guard condition is Nav__Depth<=SurfaceThreshold */

E<> DevState__MastState==GPSFixer_P.UP and GPSFixer_P.RaiseMast_ds imply GPSFixer_P.TakeFix_ds

/*Is it possible to reach the next state (TakeFix_ds) when its at present state RaiseMast_ds and the guard condition checking whether the mast is raised is satisfied */

E<> DevState__GPSFixState==GPSFixer_P.DONE and GPSFixer_P.TakeFix_ds imply GPSFixer_P.ComeOffSurface_ds

/*Is it possible to reach the next state (ComeOffSurface_ds) when its at present state TakeFix_ds and the guard condition checking whether the GPSFix is done is satisfied*/

A[] not deadlock

/*Does there exist a deadlock*/

E<> GPSFixer_P.ComeOffSurface_ds imply GPSFixer_P.Decide_ds || GPSFixer_P.Idle_ds

/*Is it possible to reach the next state (Decide_ds or Idle_ds) when its at present state ComeOffSurface_ds */

E<> GPSFixer_P.Decide_ds and GPSOrd__ReturnToStart and GPSFixer_P.ReturnToStart_ds

/* Is it possible to reach the next state (ReturnToStart_ds) when its at present state Decide_ds and the guard conditon checking whether to return to start stae is satisfied*/

E<> GPSFixer_P.Decide_ds and !GPSOrd__ReturnToStart imply GPSFixer_P.Idle_ds

/* Is it possible to reach the next state (Idle_ds) when its at present state Decide_ds and the guard conditon checking whether not to return to start state is satisfied */

E<> GPSFixer_P.Decide_ds and GPSOrd__ReturnToStart and Steering_P.SteerToPoint_ds

/* Is it possible to reach SteerToPoint_ds in Steering when the guard condition is satisfied when in the Decide_ds state in GPSFixer */

E<> GPSFixer_P.Decide_ds and !GPSOrd__ReturnToStart and Steering_P.SteerToPoint_ds

/* is it possible to reach from Decide_ds in GPSFixer to Steering (SteerToPoint_ds) when the guard condition is not satisfied */

E<> GPSFixer_P.Decide_ds and !GPSOrd__ReturnToStart and Steering_P.Idle_ds

E<> GPSFixer_P.TakeFix_ds and DevState__GPSFixState==GPSFixer_P.DONE and Launcher_P.RetractMast_ds

/*is it possible to reach from TakeFix_ds in GPSFixer to Launcher (RetractMast_ds) when the guard condition is not satisfied and the synchronous event occurs */

C.4 Verification of Waypointnavigator module

The value of level *i* remains 2. The next module selected is the Waypointnavigator subsystem at level 2. The environment for Waypointnavigator susbsystem is now abstracted. The Waypointnavigator subsystem is shown in Figure 18, the abstracted commanding environment is shown in Figure 19 and the abstracted commanded environment is shown in Figure 20.

Queries for the verification of Waypointnavigator is shown belo.

A[] not deadlock

/*Does there exist a deadlock*/

A<> WaypointNavigator_P.Idle_ds

/* All paths eventually lead to the final state (inidicating progress or diagnosis of failure if aborted) */

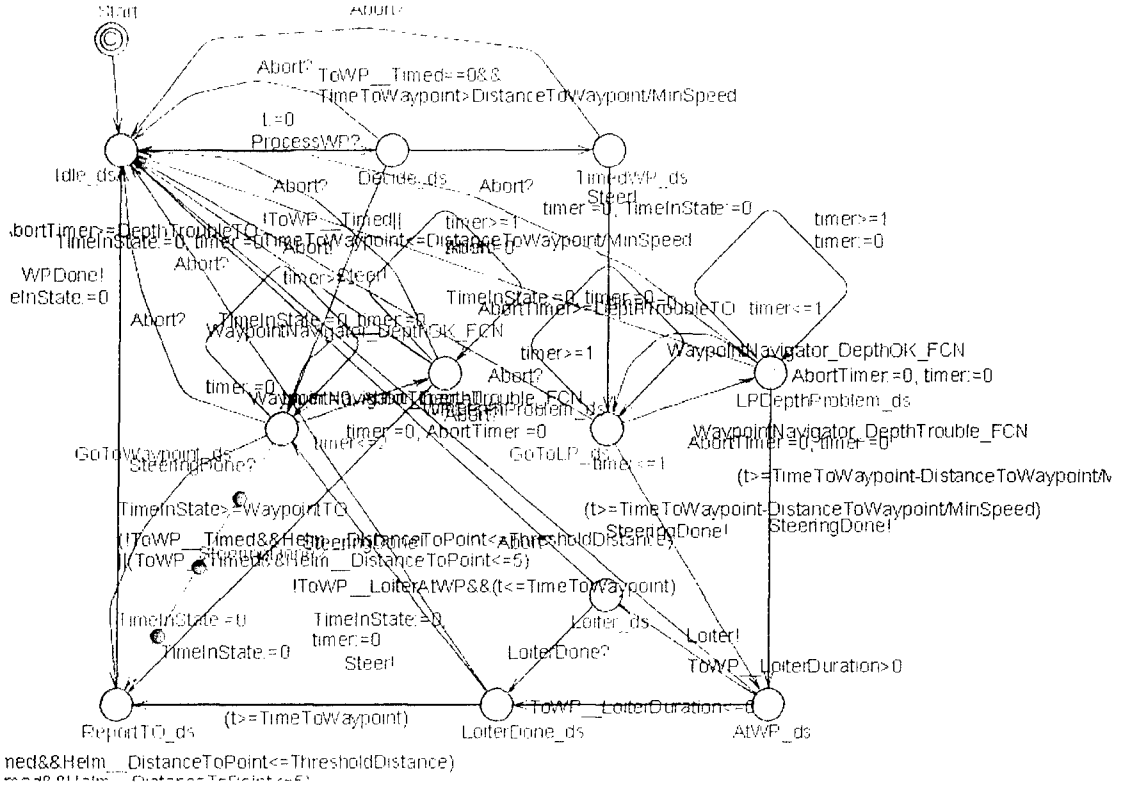

Figure 18: Waypointnavigator module in UPPAAL

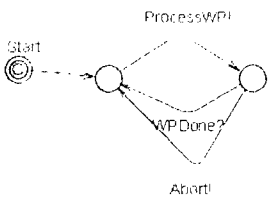

Figure 19: Driver for Waypointnavigator module in UPPAAL

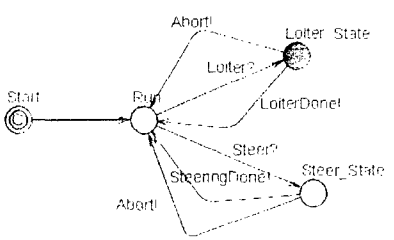

Figure 20: Stub for Waypointnavigator module in UPPAAL

E<> ToWP__Timed && WaypointNavigator_P.TimeToWaypoint > WaypointNavigator_P.DistanceToWaypoint / WaypointNavigator_P.MinSpeed and WaypointNavigator_P.TimedWP_ds

/* Is it possible to do a timed waypoint if the guard conditions are satisfied (Yes indicates correct implementation) Change: TimeToWaypoint DistanceToWaypoint MinSpeed to check the various situations with the guard condition */

E[] (WaypointNavigator_P.TimeToWaypoint > WaypointNavigator_P.DistanceToWaypoint /WaypointNavigator_P.MinSpeed) imply \ WaypointNavigator_P.TimedWP_ds

/* Whenever the guard signifying time is satisfied does it do timed waypoint */

E<> !ToWP__Timed || WaypointNavigator_P.TimeToWaypoint <= WaypointNavigator_P.DistanceToWaypoint/WaypointNavigator_P.MinSpeed and WaypointNavigator_P.GoToWaypoint_ds

/* Is it possible to do an untimed waypoint if the guard conditions are satisfied (Yes indicates correct implementation) Change: TimeToWaypoint DistanceToWaypoint MinSpeed to check the various situations with the guard condition */

E[] (WaypointNavigator_P.TimeToWaypoint < WaypointNavigator_P.DistanceToWaypoint / WaypointNavigator_P.MinSpeed) imply WaypointNavigator_P.GoToWaypoint_ds

/*Is it possible to do untimed waypoint when time to reach the point by the vehicle is more than desired time*/

WaypointNavigator_P.TimedWP_ds --> WaypointNavigator_P.GoToLP_ds

/*Is it possible to reach go to loiter point state once timed waypoint is started (as the next step is to go to loiter point indicating progress)*/

A<> *(WaypointNavigator_P.GoToWaypoint_ds* *and* *WaypointNavigator_P.WaypointNavigator_DepthTrouble_FCN) imply* *WaypointNavigator_P.WPDepthProblem_ds*

/* For all paths at state GoToWaypoint_ds if depth trouble occurs it goes to rectify it.*/

E<> *(WaypointNavigator_P.WaypointNavigator_DepthOK_FCN* *and* *WaypointNavigator_P.WaypointNavigator_DepthTrouble_FCN) imply* *WaypointNavigator_P.GoToWaypoint_ds*

/*Is it possible to get depth rectified and move on to normal state*/

E[] WaypointNavigator_P.GoToWaypoint_ds imply Stub.Steer_State

/*There exists a path where always steering needs to be done whenever waypoint needs to go to a point*/

E<> *(!ToWP__Timed&&Helm__DistanceToPoint<=* *WaypointNavigator_P.ThresholdDistance* *)* || (ToWP__Timed&&Helm__DistanceToPoint <= 5) imply Stub.Run

/* The guard conditions being satisfied steering is done, the point is reached*/

E<> *(WaypointNavigator_P.GoToWaypoint_ds* || *WaypointNavigator_P.WPDepthProblem_ds)&&(!ToWP__Timed&&Helm__DistanceTo* *Point* *<=* *WaypointNavigator_P.ThresholdDistance* *)||(ToWP__Timed&&Helm__DistanceToPoint<=5)* *imply* *Stub.Run* *&&* *WaypointNavigator_P.ReportTO_ds*

/* Is it possible to go to the Idle state in steering and ReportTO state in Waypoint from go to waypoint or depth problem rectifying state in waypoint when the guard conditions are satisfied*/

E<> WaypointNavigator_P.GoToLP_ds imply Stub.Steer_State

/* Is it possible to steer to desired point when doing timed waypoint */

E<> (WaypointNavigator_P.GoToLP_ds and WaypointNavigator_P.WaypointNavigator_DepthTrouble_FCN) imply WaypointNavigator_P.LPDepthProblem_ds

/* Is it possible to go to depth correction state when the guard indicating depth trouble is TRUE (yes indicates correct performance) */

E<> WaypointNavigator_P.LPDepthProblem_ds and WaypointNavigator_P.timer>=2

/*Is it possible to be at state LPDepthProblem_ds with timer greater than 2 (if yes indicating wrong model) couldnt find result as taking lot of time need to check for long */

E<> WaypointNavigator_P.LPDepthProblem_ds and WaypointNavigator_P.timer<=1

/* Is it possible to be at state LPDepthProblem_ds with timer less than equal to 1 (if yes indicating correct model) couldnt find result as taking lot of time need to check for long */

E<> WaypointNavigator_P.GoToLP_ds and WaypointNavigator_P.timer>=2

/* Is it possible to be at state GoToLP_ds with timer greater than 2 (if yes indicating wrong model) couldnt find result as taking lot of time need to check for long*/

E<> WaypointNavigator_P.GoToLP_ds and WaypointNavigator_P.timer<=1

/* Is it possible to be at state GoToLP_ds with timer less than equal to 1 */

E<> WaypointNavigator_P.GoToWaypoint_ds and WaypointNavigator_P.timer>=3

/* Is it possible to be at state GoToWaypoint_ds with timer greater than 3 */

E<> WaypointNavigator_P.GoToWaypoint_ds and WaypointNavigator_P.timer<=2

/*Is it possible to be at state GoToWaypoint_ds with timer less than equal to 2 */

E<> WaypointNavigator_P.WPDepthProblem_ds and WaypointNavigator_P.timer>=2

/*Is it possible to be at state GoToWaypoint_ds with timer greater than 2 */

E<> WaypointNavigator_P.WPDepthProblem_ds and WaypointNavigator_P.timer<=1

/* Is it possible to be at state GoToWaypoint_ds with timer less than equal to 1 */

E<> WaypointNavigator_P.TimedWP_ds imply (WaypointNavigator_P.Loiter_ds || WaypointNavigator_P.LoiterDone_ds)

/*Is it possible to loiter if executing a timed waypoint */

E<> WaypointNavigator_P.AtWP_ds and ToWP__LoiterDuration<=0 imply WaypointNavigator_P.LoiterDone_ds

/* Is it possible to reach the desired region when no time is left for loitering while doing a timed waypoint */

E<> WaypointNavigator_P.AtWP_ds and ToWP__LoiterDuration>0 imply WaypointNavigator_P.Loiter_ds

/* Is it possible to loiter when time is left to reach the desired region while doing a timed waypoint */

E<> WaypointNavigator_P.LoiterDone_ds and (WaypointNavigator_P.t >=WaypointNavigator_P.TimeToWaypoint) imply WaypointNavigator_P.ReportTO_ds

/* Is it possible to reach the data reporting state after loitering is done */

E<> WaypointNavigator_P.ReportTO_ds imply WaypointNavigator_P.Idle_ds

/* Is it possible to reach the final state when the desired point is reached */

C.5 Verification of Rendezvous module

The value of level *i* remains 2. The next module selected is the Rendezvous subsystem at level 2. The environment for Rendezvous susbsystem is now abstracted. The Rendezvous subsystem is shown in Figure , the abstracted commanding environment is shown in Figure and the abstracted commanded environment is shown in Figure .

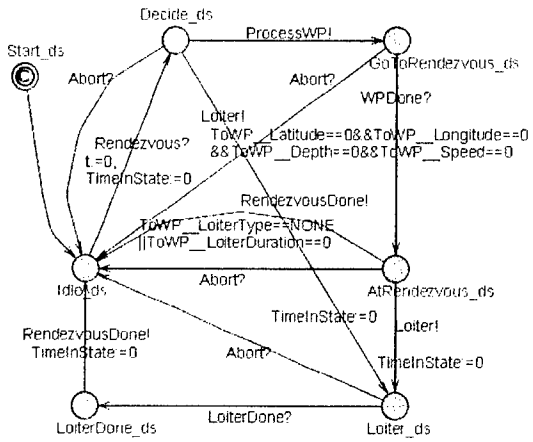

Figure C.12: Rendezvous module in UPPAAL

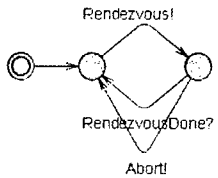

Figure C.13: Driver for Rendezvous module in UPPAAL

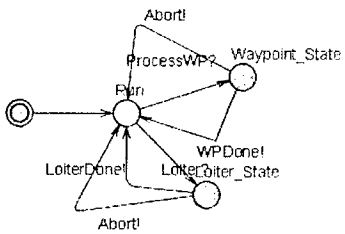

Figure C.14: Stub for Rendezvous module in UPPAAL

Queries as temporal logic formulas to verify properties for the Rendezvous susbsystem.

A[] not deadlock

/*Does there exist a deadlock*/

E<> ToWP__Latitude==0&&ToWP__Longitude == 0 && ToWP__Depth == 0 && ToWP__Speed == 0 imply Rendezvous_P.Loiter_ds

/* Is it possible to loiter when rendezvous is not specified*/

E<> ToWP__LoiterType==Rendezvous_P.NONE ||ToWP__LoiterDuration==0 imply Rendezvous_P.Idle_ds

/* Is it possible to be idle when no loiter type is given and no loiter duration is specified*/

E<> Rendezvous_P.GoToRendezvous_ds and Stub.Waypoint_State

/*Is it possible to start waypoint navigation by the rendezvous controller */

E<> Rendezvous_P.AtRendezvous_ds and Stub.Run

/* Is it possible for rendezvous to synchronize with waypointNavigator to process way point navigation */

E<> Rendezvous_P.Loiter_ds and Stub.Loiter_State

/* Is it possible to synchronize Rendezvous with Loiter for loitering*/

E<> Rendezvous_P.LoiterDone_ds and Stub.Run

/* Is it possible to synchronize rendezvous with loiter to get loitering done */

Rendezvous_P.LoiterDone_ds --> Stub.Run

/*Is it possible by the rendezvous controller to execute loiter successfully*/

E<> Rendezvous_P.LoiterDone_ds and Stub.Run

/*Checking on synchronization of rendezvous with loiter*/

C.6 Verification of Launcher module

The value of level *i* remains 2. The next module selected is the Launcher subsystem at level 2. The environment for Launcher susbsystem is now abstracted. The Launcher subsystem is shown in Figure , and the abstracted commanding environment is shown in Figure C.16.

Queries as temporal logic formulas to verify the properties satisfied by the Launcher subsystem.

A[] not deadlock

/*Does there exist a deadlock*/

E<> Launcher_P.Idle_ds imply Launcher_P.RetractMast_ds

/* Is it possible to retract mast using the launcher */

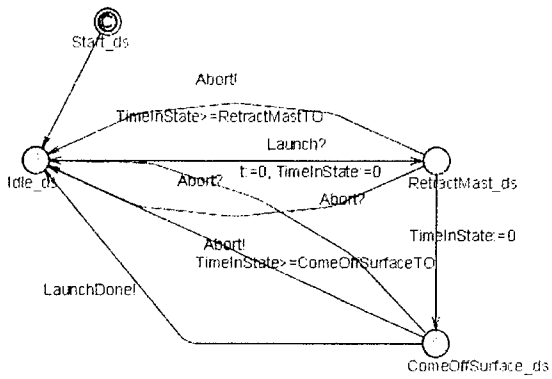

Figure C.15: Launcher module in UPPAAL

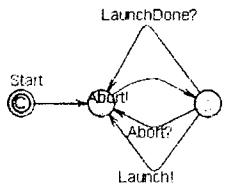

Figure C.16: Driver for Launcher module in UPPAAL

E<> Launcher_P.RetractMast_ds and Launcher_P.TimeInState >= Launcher_P.RetractMastTO imply Launcher_P.Idle_ds

/*Is it possible to abort launcher operation when time at that state is greater than time to retract mast indicating some failure */

E<> Launcher_P.RetractMast_ds imply Launcher_P.ComeOffSurface_ds

/* Is it possible to dive down using the launcher */

E<> Launcher_P.ComeOffSurface_ds imply Launcher_P.Idle_ds

/*Is it possible to reach the final state once diving of the surface is done */

C.7 Verification of PayloadDelivery module

The value of level *i* remains 2. The next module selected is the PayloadDelivery subsystem at level 2. The environment for PayloadDelivery susbsystem is now

abstracted. The PayloadDelivery subsystem is shown in Figure 21, the abstracted commanding environment is shown in Figure 22 and the abstracted commanded environment is shown in Figure 23.

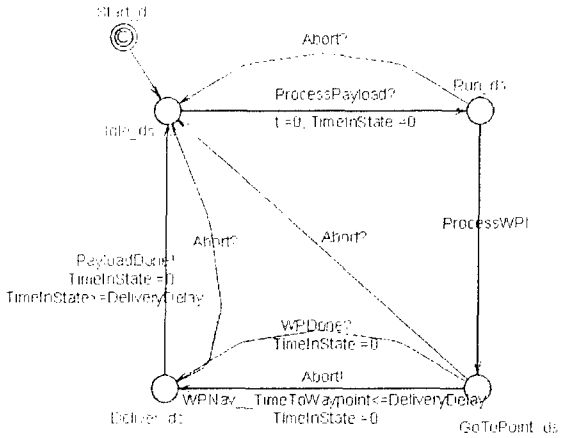

Figure 21: PayloadDelivery module in UPPAAL

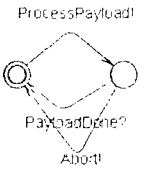

Figure 22: Driver for PayloadDelivery module in UPPAAL

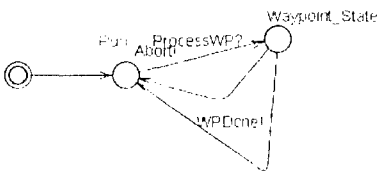

Figure 23: Stub for PayloadDelivery module in UPPAAL

Queries to verify properties satisfied by the PayloadDelivery subsystem

A[] not deadlock

/*Does there exist a deadlock*/

E<> PayloadDelivery_P.Idle_ds

/* Is it possible to reach the final state from any other state */

E<> PayloadDelivery_P.GoToPoint_ds and Stub.Waypoint_State

/*Is it that Payload module passes control to waypointnavigator to go to the desired location*/

E<> PayloadDelivery_P.Deliver_ds and Stub.Run

/*Synchronizes with waypointnavigator to go to the desired location successfully*/

E<> PayloadDelivery_P.GoToPoint_ds and WPNav__TimeToWaypoint <= PayloadDelivery_P.DeliveryDelay imply PayloadDelivery_P.Deliver_ds and Stub.Run

/*With the guard conditions being satisfied does the payloaddelivery synchronize with waypointnavigator successfully */

C.8 Verification of DeviceCommander module

The value of level *i* remains 2. The next module selected is the DeviceCommander subsystem at level 2. The environment for DeviceCommander susbsystem is now abstracted. The DeviceCommander subsystem is shown in Figure 24, and the abstracted commanding environment is shown in Figure 25.

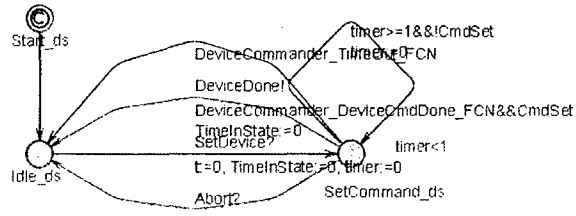

Figure 24: DeviceCommader module

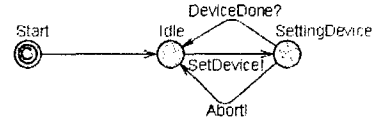

Figure 25: Driver for DeviceCommander module in UPPAAL

Queries for DeviceCommander module is as given below.

A[] not deadlock

/*Does there exist a deadlock*/

E<> DeviceCommander_P.SetCommand_ds and DeviceCommander_P.timer<1

/* s it possible for time to be greater than 1 at state setcommand*/

E<> DeviceCommander_P.DeviceCommander_DeviceCmdDone_FCN && DeviceCommander_P.CmdSet imply DeviceCommander_P.Idle_ds

/*Is device set successfully*/

E<> DeviceCommander_P.DeviceCommander_TimeOut_FCN && DeviceCommander_P.SetCommand_ds imply DeviceCommander_P.Idle_ds

/* Is it possible to time out when going to set a device*/

E<> DeviceCommander_P.SetCommand_ds and DeviceCommander_P.timer>1

/* Is it possible to remain at state SetCommand without updation for more than 1 second*/

C.9 Verification of Pause module

The value of level *i* remains 2. The next module selected is the PayloadDelivery subsystem at level 2. The environment for PayloadDelivery susbsystem is now abstracted. The PayloadDelivery subsystem is shown in Figure 26, and the abstracted commanding environment is shown in Figure 27.

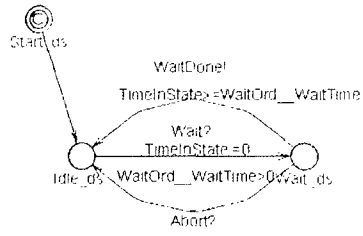

Figure 26: Pause module in UPPAAL

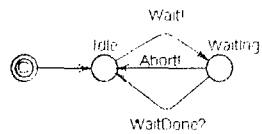

Figure 27: Driver for pause module in UPPAAL

Queries to verify properties satisfied by the Pause subsystem

A[] not deadlock

/*Does there exist a deadlock*/

Pause_P.Idle_ds and WaitOrd__WaitTime>0 --> Pause_P.Wait_ds

/* Is it possible to go to wait state to keep the coordinator waiting */

E<> Pause_P.Wait_ds and Pause_P.TimeInState>=WaitOrd__WaitTime imply Pause_P.Idle_ds

/* Is it possible to successfully complete pause operation*/

C.10 Verification of Sequential coordinator module

The value of level *i* changes to level 3. The next module selected is the Sequential Coordinator subsystem at level 2. The environment for sequential coordinator susbsystem is now abstracted. The Sequential Coordinator subsystem is shown in Figure 28, the abstracted commanding environment is shown in Figure 29 and the abstracted commanded environment is shown in Figure 30.

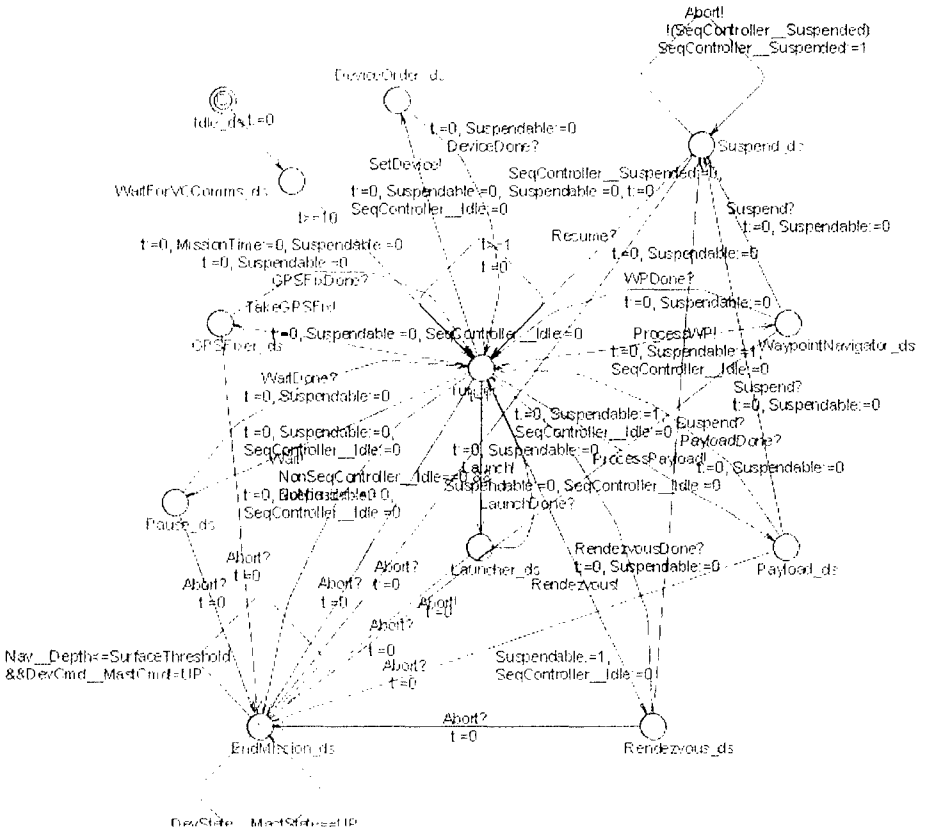

Figure 28: Sequential coordinator module in UPPAAL

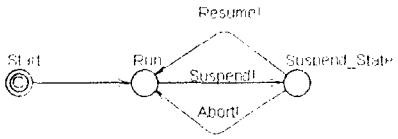

Figure 29: Driver for sequential coordinator module in UPPAAL

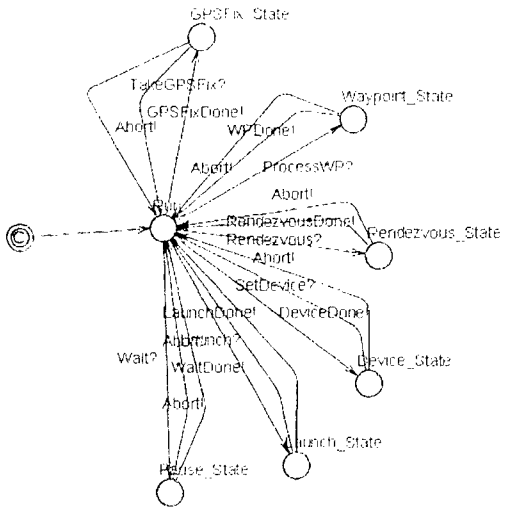

Figure 30: Stub for Sequential coordinator module in UPPAAL

Queries to verify properties satisfied by the Sequential Coordinator subsystem

A[] not deadlock

/*Does there exist a deadlock*/

E<> Controller_P.EndMission_ds

/*Is it possible to finally reach the end state*/

E<> Controller_P.WaitForVCComms_ds and Controller_P.t>10

/*Does the sequential controller wait for 10 seconds before it goes to run state*/

E<> Controller_P.run_ds and Controller_P.t<=1

/*Does the controller check the missionqueue every 1 second to check for pending requests*/

E<> Controller_P.GPSFixer_ds and Stub.GPSFix_State

/*Does the controller pass control to GPSfixer operation controller when the order is to perfom GPSFix*/

E<> Controller_P.GPSFixer_ds and Controller_P.Suspendable==0

/*Is it that GPSFixer is non suspendable(Yes indicating the design is correct as it should not be suspendable)*/

E<> Controller_P.GPSFixer_ds and SeqController__Idle==0

/*Is it possible for sequential coordinator to transfer control to waypoint navigator and wait*/

E<> Controller_P.run_ds imply Stub.Run

/*Is it possible that when controller is in run state GPSFixer is in idle state indicating that GPSFix has been done*/

E<> Controller_P.WaypointNavigator_ds and Stub.Waypoint_State

/*Does the controller pass control to WaypointNavigator operation controller when the order is to perform Waypoint navigation*/

E<> Controller_P.WaypointNavigator_ds and Stub.Run

/*Does the WaypointNavigator perform the operation successfully*/

E<> Controller_P.WaypointNavigator_ds and Controller_P.Suspendable==1

/*Is it that WaypointNavigator is suspendable(Yes indicating the design is correct as it should be suspendable)*/

E<> Controller_P.WaypointNavigator_ds and SeqController__Idle==0

/*Is it possible for sequential coordinator to transfer control to waypoint navigator and wait*/

E<> Controller_P.Pause_ds and Stub.Pause_State

/*Does the sequential controller pass control to pause*/

E<> Controller_P.Pause_ds and SeqController__Idle==0

/*Is the Secontroller idle when it passes control to Pause*/

E<> Controller_P.Pause_ds and Controller_P.Suspendable==0

/*Is the pause operation suspendable*/

E<> Controller_P.Launcher_ds and Stub.Launch_State

/*Does the controller pass control to Launcher module */

E<> Controller_P.run_ds and Stub.Run

/*Is Launch command completed successfully*/

E<> Controller_P.Launcher_ds and SeqController__Idle==0

/*Is the Seqcontroller idle when it passes control to Launcher*/

E<> Controller_P.Launcher_ds and Controller_P.Suspendable==0

/*Is the Launch operation suspendable*/

E<> Controller_P.Rendezvous_ds and Stub.Rendezvous_State

/*Is it possible to pass control to Rendezvous*/

E<> Controller_P.run_ds and Stub.Run

/*Is the Rendezvous mission completed successfully */

E<> Controller_P.Rendezvous_ds and SeqController__Idle==0

/*Is the Seqcontroller idle when it passes control to Rendezvous*/

E<> Controller_P.Rendezvous_ds and Controller_P.Suspendable==1

/*Is the Launch operation suspendable*/

E<> Controller_P.DeviceOrder_ds and Stub.Device_State

/*Is it possible to pass control to DeviceCommander module*/

E<> Controller_P.DeviceOrder_ds and SeqController__Idle==0

/*Is the Seqcontroller idle when it passes control to Devicecommander*/

E<> Controller_P.DeviceOrder_ds and Controller_P.Suspendable==0

/*Is the DeviceCommander operation suspendable*/

E<> Controller_P.Payload_ds and SeqController__Idle==0

/*Is the Seqcontroller idle when it passes control to Payload*/

E<> Controller_P.Payload_ds and Controller_P.Suspendable==0

/*Is the Payload operation suspendable*/

E<> Controller_P.Suspend_ds and SeqController__Suspended==1

/*Is there a method to test whether the seq.Controller is suspended*/

E<> Controller_P.Suspend_ds imply Controller_P.run_ds or Controller_P.EndMission_ds

/*Is it possible to return back to normal operation or end the mission after suspension*/

E<> Nav__Depth<=Controller_P.SurfaceThreshold && DevCmd__MastCmd != Controller_P.UP

/*Is it possible to come to surface of water and raise mast to indicate end of mission*/

E<> Controller_P.EndMission_ds and DevState__MastState==Controller_P.UP

/*Is it possible to raise mast at end of state*/

E<> Controller_P.run_ds && NonSeqController__Idle==0 && NoMission==0 imply Controller_P.EndMission_ds

/*Does the Seq. Controller check for the status of other controllers before ending the mission*/

E<> Controller_P.EndMission_ds and Controller_P.Suspendable==0

/*Is the end mission state suspendable*/

E<> Controller_P.EndMission_ds and SeqController__Idle==0

/*Is the Seq. Controller idle at end mission state*/

E<> Driver.Suspend_State imply Controller_P.Suspend_ds

/*Is it possible to suspend Seq. Controller by Non Seq. Controller*/

E<> Driver.Run and Controller_P.run_ds

/*Is it possible for both the Seq. and Non Seq. Controller to be ready at the same time*/

C.11 Verification of Timed Coordinator module

The value of level *i* remains at level 3. The next module selected is the Timed Coordinator subsystem at level 2. The environment for Timed coordinator susbsystem is now abstracted. The Timed Coordinator subsystem is shown in Figure 31, and the abstracted commanded environment is shown in Figure 32.

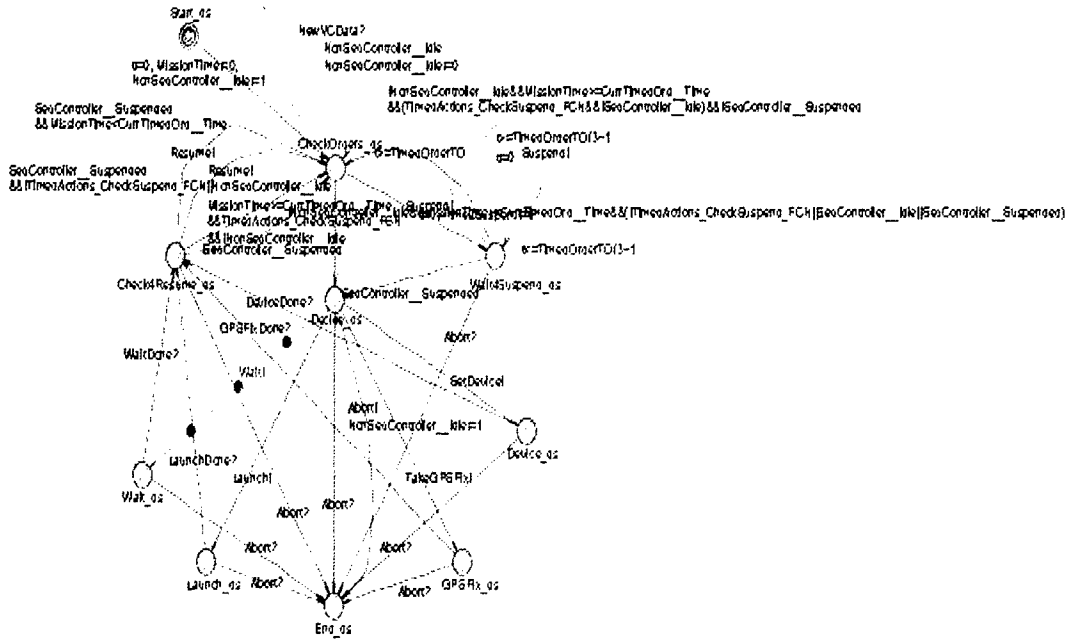

Figure 31: Timed coordinator module in UPPAAL

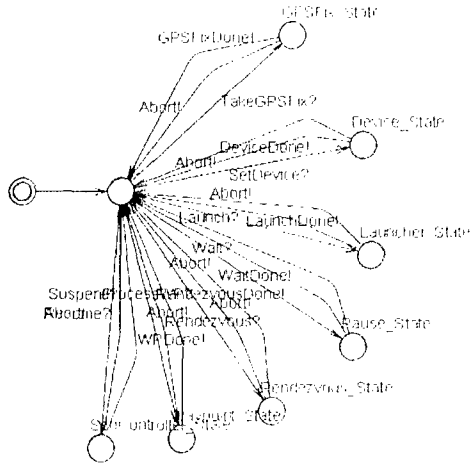

Figure 32: Stub for timed coordinator module in UPPAAL

Queries to verify the the properties satisfied by the Timed coordinator

A[] not deadlock

/*Does there exist a deadlock*/

E<> TimedActions_P.End_ds

/*Is it possible to reach the final state*/

E<> TimedActions_P.Device_ds imply Stub.Device_State

/*Does the Timed action pass the control to Device commander when it needs to set or start a device*/

E<> TimedActions_P.Launch_ds and Stub.Launcher_State

/*Does the Timed action pass the control to launcher when it needs to comeoffsurface or act with the mast*/

E<> TimedActions_P.Wait_ds imply Stub.Pause_State

/*Does the Timed action pass the control to pause when it needs to wait*/

E<> TimedActions_P.GPSFix_ds imply Stub.GPSFix_State

/*Is it possible to execute a timed GPSFix*/

E<> TimedActions_P.Wait4Suspend_ds and TimedActions_P.t == TimedActions_P.TimedOrderTO/3+1

/*Does the timed actions try to suspend the seq. coordinator every desired time*/

E<> TimedActions_P.Wait4Suspend_ds imply TimedActions_P.t <= TimedActions_P.TimedOrderTO/3+1

/*Does the timed actions try to suspend the seq. coordinator every desired time*/

E<> TimedActions_P.CheckOrders_ds && TimedActions_P.MissionTime >= CurrTimedOrd__Time&&(TimedActions_P.InterruptCoordinator_CheckSuspend_FCN&&SeqCoordinator__Suspendable&&!SeqCoordinator__Idle)&& !SeqCoordinator__Suspended imply TimedActions_P.Wait4Suspend_ds

/*Is it possible to suspend the seq. controller*/

E<> TimedActions_P.MissionTime>=CurrTimedOrd__Time && TimedActions_P.TimedActions_CheckSuspend_FCN && !NonSeqController__Idle imply TimedActions_P.CheckOrders_ds

/*Does the Timed action check for orders when the mission time is greater than the current time (indicating timed mission
can be accomplished) and does Timed action check the need to suspend seq controller or not and timed actions is not idle*/

E<> TimedActions_P.Check4Resume_ds and !SeqController__Suspended imply TimedActions_P.CheckOrders_ds

/*Is it possible for the timed action to chekc for orders when the sequential controller doesnt need to be suspended*/

E<> TimedActions_P.Check4Resume_ds and SeqController__Suspended and TimedActions_P.MissionTime<CurrTimedOrd__Time imply TimedActions_P.CheckOrders_ds

/*Does the Timed action controller check timed orders when Seq. Controller is suspended and Mission time is greater than
current time for timed order*/

E<> TimedActions_P.Check4Resume_ds and SeqController__Suspended && not TimedActions_P.TimedActions_CheckSuspend_FCN || NonSeqController__Idle imply TimedActions_P.CheckOrders_ds

/*Does the Time action controller check timed orders when seq. controller is suspended and timed action doesn't need suspension or timed controller is idle*/

E<>TimedActions_P.CheckOrders_ds and !NonSeqController__Idle && TimedActions_P.MissionTime>=CurrTimedOrd__Time && (not TimedActions_P.TimedActions_CheckSuspend_FCN || SeqController__Idle || SeqController__Suspended) imply TimedActions_P.Decide_ds

/*Does the timed action become ready to execute orders when the timed controller is not idle and mission time is greater than current timed order time and timed action doesnt need suspension or sequential controller is idle or sequential controller is already suspended*/

E<> TimedActions_P.Wait_ds imply not Stub.Pause_State

/*Is it possible to transfer control to Pause moduel*/

E<> TimedActions_P.Decide_ds imply TimedActions_P.End_ds

/*Is it possible to go to the final state from the decide state*/

E<> TimedActions_P.Wait_ds imply TimedActions_P.End_ds

/*Is it possible to go to the final state from the situation where control is passed to the pause controller*/

E<> TimedActions_P.Launch_ds imply TimedActions_P.End_ds

/*Is it possible to go to the final state from the situation where control is passed to the launch controller*/

E<> TimedActions_P.GPSFix_ds imply TimedActions_P.End_ds

/*Is it possible to go to the final state from the situation where control is passed to the GPSFix controller*/

E<> TimedActions_P.Device_ds imply TimedActions_P.End_ds

/*Is it possible to go to the final state from the situation where control is passed to the Device controller*/

E<> TimedActions_P.CheckOrders_ds imply TimedActions_P.End_ds and TimedActions_P.Idle==1

/*Is it possible to go to the final state from state to check orders*/

A<> TimedActions_P.Decide_ds imply (TimedActions_P.End_ds || TimedActions_P.Wait_ds || TimedActions_P.Launch_ds || TimedActions_P.GPSFix_ds || TimedActions_P.Device_ds)

/*All paths eventually lead to final state or pause or launcher or GPSFixer or Device from the decide state in timed actions*/

E<> TimedActions_P.End_ds imply TimedActions_P.Idle==1

/*Is it possible for Non Seq. Controller to be idle at end of mission*/

E<> TimedActions_P.CheckOrders_ds imply TimedActions_P.Idle==0 && TimedActions_P.Done

/*Is it possible for Non Seq.Controller to be idle at CO state*/

E<> TimedActions_P.CheckOrders_ds imply TimedActions_P.MissionTime==0

/*Is the mission time zero at CO state*/

C.12 Verification of Safety Coordinator module

The value of level *i* remains at level 3. The next module selected is the Safety Coordinator subsystem at level 2. The Safety Coordinator checks the voltage, depth of water and the functioning of other devices from the common database. The Safety Coordinator is as shown in Figure 33.

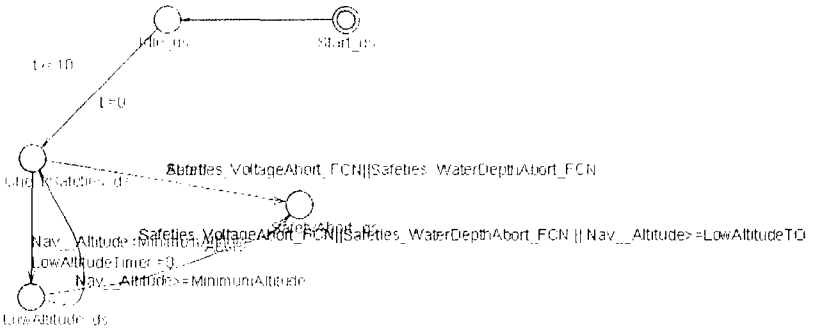

Figure 33: Safety coordinator module in UPPAAL

Queries to verify the properties of the safety module

A[] not deadlock

/*Does there exist a deadlock*/

E<> Safeties_P.CheckSafeties_ds && Nav__Altitude < Safeties_P.MinimumAltitude imply Safeties_P.LowAltitude_ds

/*Is there method a to check water depth safety*/

E<> Safeties_P.CheckSafeties_ds && Safeties_P.Safeties_VoltageAbort_FCN || Safeties_P.Safeties_WaterDepthAbort_FCN) imply Safeties_P.SafetyAbort_ds

/*Is tehr method to check voltage safety*/

E<> Safeties_P.LowAltitude_ds && Nav__Altitude >= Safeties_P.MinimumAltitude imply Safeties_P.CheckSafeties_ds

/*Is it possible to correct the altitude*/

E<> Safeties_P.LowAltitude_ds && Safeties_P.Safeties_VoltageAbort_FCN ||Safeties_P.Safeties_WaterDepthAbort_FCN || Nav__Altitude >= Safeties_P.LowAltitudeTO imply Safeties_P.SafetyAbort_ds

/*Is there a method to abort mission if safety is violated*/

Appendix D: OpenGL Code for Animation/Simulation

Open GL Code available in the following citation :

"Modeling, Verification, and Synthesis of Hierarchical Hybrid Mission Controller for Underwater Vehicles",Siddhartha Bhattacharyya, PhD Dissertation, Department of Electrical and Computer Engineering, University of Kentucky. July 2005.