Abstracting The Planner Down: An Architecture for Planner Based Control of Autonomous Vehicles

Nick Johnson and David Lane
Ocean Systems Laboratory
Heriot-Watt University
Edinburgh, UK

Abstract

Planners provide extremely powerful functionality, allowing the user to state the goals of a system, and allowing it to formulate the specific actions which are required to complete them. They can act more powerfully still when allowed to continually reassess a plan concurrently with its execution. What is required in this instance, however, is a bridge to allow the output of the planner to be directly implemented by a vehicle system, without the need for further offline processing. Such a system is presented here, it's implementation described and successful results relayed.

Introduction

Planners and schedulers provide extremely powerful functionality and allow for the user to simply state a set of goals and have the system work out which actions should be carried out in order to complete them.

Even more powerful functionality is provided when a planner is used to constantly reassess a plan which is concurrently being executed. This requires additional infrastructure, however, as a system is required to convert the output of the planner into instructions which can actually be carried out by an autonomous system, without the need for additional offline processing. For example, a planner may output the action that a vehicle should move to a different location. This location must be resolved to an actual physical location, and then the task of navigating the vehicle to this position begun, carried through, and then completed.

This paper describes components within the executive layer of an implementation of a hybrid, or three layer, architecture. These components provide two levels of abstraction. Firstly, they allow an instance of a particular action in a high level symbolic plan to be converted to the instructions required for the vehicle to directly carry it out. A generalised manner for specifying this correlation is also supplied. Secondly, an abstraction is provided over the actual capabilities of the vehicle, particularly with respect to differing sensor and actuator payloads.

The system presented here is designed to fit into the larger framework for user experience based plan creation posited in (Johnson, Patron, and Lane 2007). Since the publication of this paper, various advances have been made towards the completion of this framework, including the development of a dynamic planning system, a schema for multi modal input and various user interface based components. The majority of this is outside the scope of this paper, however.

The remainder of this paper is divided into four sections. Firstly, some of the previous work in this field will briefly be described. Next the implementation of the system will be covered, this will be followed by the results which have been gained from using the system and lastly intended future work will be described.

Previous Work

Planning is one of the fundamental artificial intelligence problems with one of the first and most often cited systems being the STRIPS system (Fikes and Nilsson 1971). The technology has developed significantly since this system was first created, though. An excellent illustration of this is demonstrated each year when the International Conference on Automated Planning and Scheduling (ICAPS) hosts the International Planning Competition (IPC). Entrants to this competition take input from the Planning Domain Description Language (or PDDL, see (Gerevini and Long 2005)) for various problems and compete to create the most complete and efficient plans, trying to do so as quickly as possible. Notable due to it's influence on this project is the SGPlan system (Chen, Wah, and Hsu 2006) which is based on the Metric-FF system (Hoffmann 2003). As the IPC champion at the time this research began, it was selected as the basis for the planning components of the system.

The IPC largely focuses on advances in planning technology itself, rather than the technology for integrating planners to real systems. A lot of work on continual planning across multiple systems has also concentrated on the planner itself, rather than on the actual means of taking advantage of the output from the planner ((desJardins et al. 2000), (Nau, Smith, and Erol 1998)) or has focused on the implementation of bespoke planning solution for the control of particular vehicle systems (Pêtrès et al. 2007).

Hybrid, or three-layer, architectures have become one of the the most popular and successful approaches to autonomous vehicle control. They are able to deploy the strengths of both reactive and deliberative architectures, whilst minimising the weaknesses found in both. Although there is some difference in the naming of the three layers, and the components which are placed in each, certain

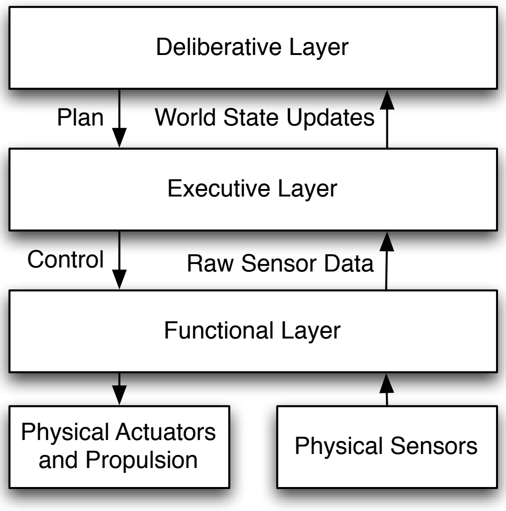

Figure 1: Simplified Hybrid Architecture.

commonalities exist across most (if not all) implementations. Figure 1 shows a representation of a hybrid architecture, which has been somewhat simplified for illustrative purposes.

We name our three layers *deliberative*, *executive* and *functional*. We restrict the functional layer to containing only components which control the physical systems of the vehicle. Reactive control operates within this layer, allowing the system to react quickly to sensor data when required. One example of this would be an emergency evasive manoeuvre in response to the sudden appearance of a large object in front of the vehicle (or at least the detection of one).

We also limit the deliberative layer to containing only the planning components. Thus the deliberative layer receives changes to the vehicles perception of the world (represented at a symbolic level) and outputs a plan which is designed to achieve the mission goals.

As such, all other components required to make the system work are contained in the executive layer. These may include, but are not limited to: an autopilot, high level sensor processing and fusion, navigation, mission monitoring, action selection, and a system to convert high level actions into commands which control the vehicle's physical systems (which is what we present here).

An example of a multiple robot control system which operates in the executive layer was demonstrated in (Sotzing, Evans, and Lane 2007) (with results from trials with multiple real vehicles to be published later). This system is capable of carrying out a plan constructed in the custom BIIMAPS plan representation (Sotzing, Johnson, and Lane 2008), as well as limited modification of such a plan. However, this system relies on the plan being made up of a set of built in behaviours at present.

A more in depth review of hybrid architectures, together with a recent implementation, can be found in (Ridao et al. 2000).

Implementation

Figure 2 shows the complete architecture. The remainder of this section is divided into subsections which describe each of the elements of this diagram in more detail.

The Dynamic Planner

The architecture described here is designed to facilitate planner based control of autonomous systems, and as such a planner forms its uppermost layer. A dynamic planner has been implemented for the purpose, but the details of this implementation are beyond the scope of this paper. The important facts about this system related to the architecture described here are:

- It takes a compiled representation of the Planning Domain Description Language (PDDL) as input.
- At present the system handles basic propositional planning, with no support for metric or temporal planning
- It produces partial order plans in the form of a list of predicate propositions.
- It is able to control multiple distinct autonomous vehicles and is completely decentralised.
- It is able to change the plan in response to changing circumstances.

It is important that additional information should be attached to many of the PDDL constructs used in the planning system, a facility which PDDL itself does not allow for. To this end, an XML schema was created, which uses XML tags to represent atomic constructs (such as predicate and action definitions), but raw PDDL (enclosed within XML tags) to represent compound clauses, such as goals and preconditions. The most important function provided to the control system is the attachment of the data required to tell the lower level systems how to implement each action. An example of one of the used action definitions is shown below[1]:

```
<action name="move">
 <parameter name="who" type="auv"/>
 <parameter name="to" type="location"/>
 <precondition>
  (and (forall (?what - locatable)
               (not (at ?what ?to)))
      (not (imobilised ?who)))
 </precondition>
 <start>
  (forall (?from - location)
          (not (at ?who ?from)))
 </start>
 <end>
  (at ?who ?to)
 </end>
 <control file="MoveControlScript.groovy"/>
</action>
```

This replaces the following definition in standard PDDL:

[1]In practice these definitions contain considerably more information, which is used for user interface purposes, but this have been stripped out here for clarity of meaning.

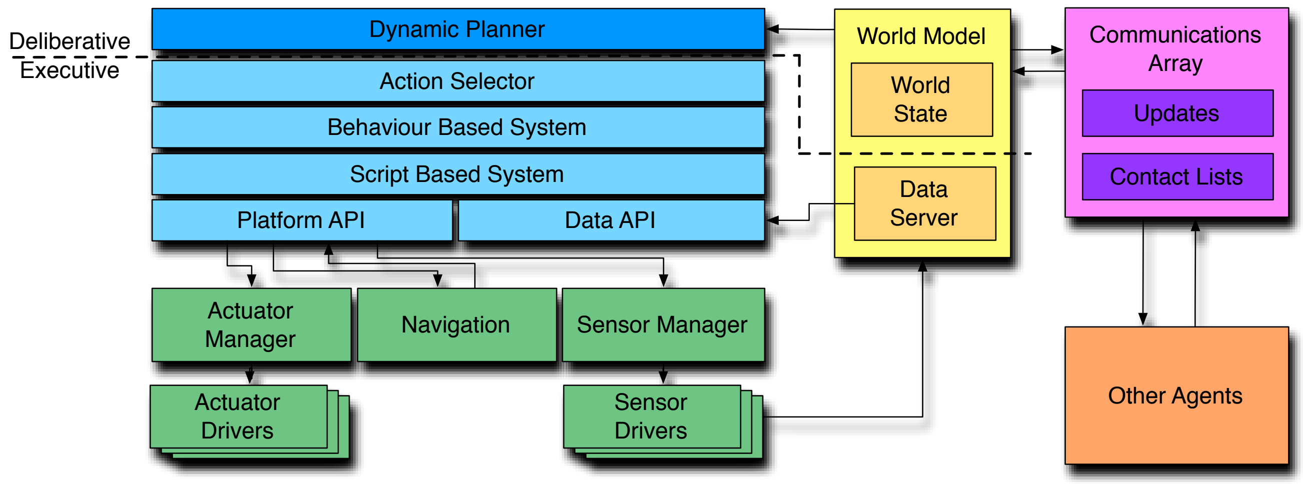

Figure 2: The architecture for a planner based control system for multiple real or simulated AUVs

```
(:action move
 :parameters (?who - auv ?to - location)
 :precondition (and (forall (?what - locatable)
                              (not (at ?what ?to)))
                         (not (imobilised ?who)))
 :effect (and (forall (?from - location)
                        (not (at ?who ?from)))
                   (at ?who ?to))
)
```

Another difference which can be seen here is that the effects of each action are separated into those which happen at the start, and those that happen at the end. Although it has been previously noted that temporal planning is not used, they are separated to allow the control system to more accurately reflect changes in the real world. When passed to the actual planner, the start and end effects are simply combined. However, this methodology allows some of the consequences of the action to be asserted in the system's copy of the world state as soon as an action starts, and the remainder as soon as it finished. In the above example, an autonomous vehicle is asserted to have left its current location as soon as it begins to move towards another, thus if a replan is triggered when the vehicle is halfway between two locations the world is accurately represented. The vehicle is then asserted to be at its new location as soon as it arrives.

The Action Selector

Below the dynamic planner in the architecture is the action selector. In out current implementation, this is very simple and merely selects the the first action in the plan which is currently available (i.e. is not required to be preceded by any other actions in the plan) and is intended to be carried out by the current agent. A more complex system, such as (Sotzing, Evans, and Lane 2007), might perform an analysis of the current state of the world and select an action intended to lead to the most efficient plan execution (by one metric or another). In either case, the result is essentially the same from the point of view of the next layer: an action represented by a predicate term and a set of parameters is passed down.

The Behaviour Based Layer

The next level down in the architecture is a behaviour based system. This is based on instances of Java (or Java compatible) classes which must implement the methods specified in the following interface:

```
public interface BehaviourScript {
 public void init(PlatformAPI platform,
                  DataAPI data,
                  String[] parameters);
 public String update(PlatformAPI platform,
                      DataAPI data);
 public void clean(PlatformAPI platform,
                   DataAPI data);
}
```

The `init` method initialises the script with the parameters which are passed to it. These parameters are the same as the parameters of the predicate term which represents the action the script is intended to carry out. The second method is the update method, which is run by the control system each times it cycles. This sends out the necessary instructions to the vehicle systems and has a return value which tells the control system whether the action still needs to run or has completed. In the current system this method will be called approximately five times a second. Finally, the `clean` method runs when an action has finished and releases any resources the script was using to implement the action, as well as leaving the vehicle in a stable state.

The Script Based System and Below

The next layer down in the architecture is the script based system. This layer provides some basic backbone, allowing scripts in a Java Virtual Machine compatible language to be executed. To facilitate this, it also provides access to the two APIs[2] which are supplied to the methods specified by the behaviour based system. These APIs are detailed in the following sections.

The Platform API

The first of the APIs mentioned above is the platform API. This gives the script the control it needs over the vehicle systems, such as requesting movement to specific locations and changing the mode of sensors and actuators. The platform API also provides feedback from the vehicle systems, such as the vehicle's current location and heading (in world frame) and the current mode of the sensors and actuators. All data sent to and received from sensors and actuators is routed through the sensor and actuator managers. These are simply small databases which index each of the sensors and actuators by name, allowing information to be passed back and forth as quickly as possible.

Vehicle navigation is controlled via waypoint requests. A waypoint represents a position in space, defined relative either to the vehicle or a fixed origin, a set of tolerances which decide when the vehicle has achieved the waypoint, a set of enables which define which axes the controller should take notice of, and also (depending on the navigation mode) a target attitude and a travel speed.

Control of sensors and actuators is achieved through very simple mode change requests. These modes are simply defined as text strings. The present implementation requires a driver for each sensor or actuator to be used in a real vehicle system to be implemented in Java. For simulated trials the sensors and actuators are specified via XML, and it is intended that this system should also be extended to allow the control of real sensors and actuators to be defined similarly. Methods are provided by the Platform API to allow the system to request a mode change and also to query the current mode of a sensor or actuator.

In simulation, the components of the architecture can be directly connected and information passed via simple method calls. On a real vehicle this is often not possible or even desirable, as different components below the level of this archtecture (such as navigation or SLAM[3] systems, and low level drivers) may be implemented in different languages, or distributed across multiple physical systems. For this reason, the system uses an UDP packet based communication system called OceanSHELL to communicate with everything below the level of the Platform API.

The Data API and World Model

The second of the APIs is the Data API. This provides the script with access to the Data Server, allowing it to obtain the locations, areas, scalar values and times to which the existences in the world state relate. This API also allows the script to add additional key/value pairs to the Data Server, allowing some persistence of data between the different scripts.

[2]Application Programming Interface

[3]Simultaneous Localisation And Mapping

Input from the vehicle sensors is assumed to have been fully processed before it reaches the control system. The receipt of this data will also most likely change the world state, either by adding new instances or predicates, or both of these. It may also add or update elements in the Data Server. As an example, the detection of a possible target might result in a `target` instance being added to the world, as well as a new `location` and an `at` predicate indicating that the target is at the location. These additions suffice for the planner to be able to adapt the mission accordingly. A new co-ordinate might also be added to Data Server, indexed to the name of the new location in order to provide a binding to the real world.

Communication

In the presently implemented system, communication is handled at a high level and consists of a set of algorithms to synchronise the World State and Data Server of all agents. These are currently at an early stage of development and are beyond the scope of this paper.

Example Behaviour Class

The class used for the implementation of each action is specified in the XML definition. In the case of the examples shown here, the classes are described in a language called groovy (G2One, Inc. 2008), a dynamic scripting language which can be compiled to pure Java at runtime. The backend needed to ensure that this functionality is available at runtime is provided by the scripting layer. The groovy class definition for the example used here is shown below:

```
class MoveControlScript
  implements BehaviourScript {

  LocalCoordinate3D waypoint
  int number

  void init(PlatformAPI platform,
            DataAPI data,
            String[] parameters) {

      waypoint =
       data.getVector(parameters[1])
      number =
       platform.getCurrentWaypointNo()++

      platform.setTolerences(1, 1, 1, 10)
      platform.setEnables(1, 1, 1, 1)
      platform.absoluteWayPointRequest(
       number, TRACK_MODE,
       waypoint.getNorth(),
       waypoint.getEast(),
       waypoint.getDepth(), 0, 1f)
  }

  String update(PlatformAPI platform,
                DataAPI data) {

    if (platform.inPosition())
     return "succeed"

      platform.absoluteWayPointRequest(
        number, TRACK_MODE,
        waypoint.getNorth(),
        waypoint.getEast(),
        waypoint.getDepth(), 0, 1f)

      return "continue"
  }

  void clean(PlatformAPI platform,
             DataAPI data) {

    platform.stay()
  }
}
```

In this example, the class contains two member fields, one for the vehicles destination and one for the number of this waypoint. These are both initialised at the beginning of the `init` method, the first by obtaining from the Data Server the co-ordinate which is referred to by the second parameter[4] of the action predicate, and the second by obtaining the current waypoint number and incrementing it. Next the tolerances and enables of the waypoint are set, indicating that the vehicle must be within one metre of the destination for the waypoint to be met and that all axes are to be used. Finally, the waypoint request itself is sent to the navigation system.

The `update` method first checks whether the waypoint has been achieved. If it has then `"succeed"` will be returned and the action will complete. Otherwise, the waypoint request will be repeated and `"continue"` will be returned.

When the `clean` method runs (on completion of the action) the vehicle is instructed to hold its current position.

Summary

As can be seen here, the system provides two distinct levels of abstraction. First of all, the actual vehicle systems are abstracted in the Platform API, allowing the system to be deployed across multiple vehicles with a minimal requirement for re-implementation of components, as well as allowing the system to be deployed unchanged in simulation, with simulated sensors and actuators taking the place of the drivers for their real counterparts.

Secondly the system provides a higher level abstraction, allowing to the simple propositional output of a planning system to be directly carried out by a vehicle's systems, provided that complementary class instances have been implemented for each of the action types the planner is able to produce.

Results

The system described here has been tested in two separate sets of circumstances. Firstly the complete system has been tested in simulation. Secondly, the lower levels have been tested on a single real vehicle, with a simpler finite state machine based system taking the place of the replanning system.

[4]Parameters are numbered from zero.

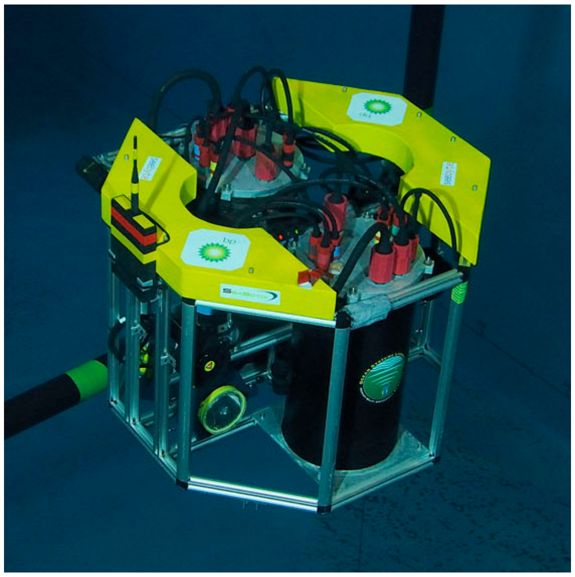

Figure 3: The Nessie III AUV passes through the validation gate at SAUC-E 2008. Image courtesy of Yves Gladu.

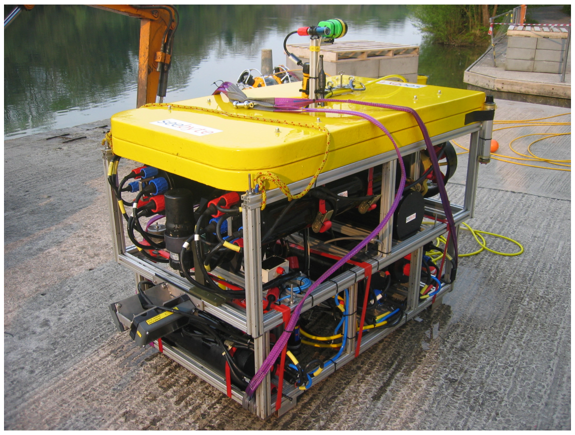

Figure 4: The RAUVER AUV.

The system has been given extensive testing in simulation, with the full dynamic planning system acting as its upper layer. The actual Autopilot System used with the Nessie III (see Figure 3) and RAUVER (see Figure 4) AUVs was used as the navigation system and a full hydrodynamic model of RAUVER used to give the control dynamics of the vehicle. Additionally, a set of simple simulated sensors and actuators were used to allow the vehicles to interact with the simulated environment. The system performed perfectly in these circumstances, allowing two scenarios (one mine counter measures and one installation maintainence) to be completed, and with sufficient efficiency to allow the simulation to proceed at up to one hundred times realtime[5].

The lower level systems were used as the basis of the control system used for the Nessie III AUV (see (Cartright et al. 2008) and Figure 3) as part of The Ocean System Laboratory's entrance into the Student Autonomous Underwater Competition - Europe (or SAUC-E) 2008. The dynamic planning and action selection systems were replaced with a much simpler finite state machine based system, as illustrated in Figure 5. Otherwise, the system was unchanged and employed the same parameterised script based mechanism for control of the vehicle systems. The finite state machine was represented by an XML file, which gave each state as the name of a script and a set of parameters. This more simplified system was used as the task required of the vehicle in the competition was based upon an entirely static environment, and therefore a replanner was not required and would have led to unwarranted additional overhead. The employed system proved to be extremely robust and flexible, helping "Team Nessie" to take the first prize in the competition, as well as "The THALES Special Award for innovation in decision making autonomy". Furthermore, the Nessie AUV completed all of the tasks laid out as part of the competition, making it the first entry in the competition's history to do so. Further information about the competition can be found at the competition website (SAUC-E Committee 2008).

Future Work

The system as described here is essentially complete, barring any updates which are made to increase functionality (to further expose particular vehicle systems for instance). Further testing and deployment is considered desirable, however. The finite state implementation deployed on the Nessie AUV is not suitable for the control of multiple robots, so it is hoped that both the Dynamic Planner based system can be tested on multiple real systems, and also that the DELPHIS system (Sotzing, Evans, and Lane 2007) (also developed at the Ocean Systems Laboratory) can be retrofitted to use this architecture as its vehicle control backend.

It also intended that various improvements should be made to the dynamic planning system which forms the upper layer of this architecture. These include, but are not limited to, the improvement and field testing of the current (extremely simple) communications system, and scope for explicitly stated consequences for the failure of actions.

[5]Faster than realtime simulation is a key property of the system described in (Johnson, Patron, and Lane 2007), and so the architecture presented here was required to support this.

Acknowledgment

This work was sponsored by the Engineering and Physical Sciences Research Council. The authors would also like to thank all the staff of both the OSL and SeeByte for their help, advice and support.

References

Cartright, J.; Johnson, N.; Davis, B.; Qiang, Z.; Bravo, T. L.; Enoch, A.; Lemaitre, G.; Roth, H.; and Petillot, Y. 2008. Nessie III autonomous underwater vehicle for SAUC-E 2008. In *Proceeding of the Unmanned Underwater Vehicle Showcase 2008*.

Chen, Y.; Wah, B. W.; and Hsu, C.-W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research* 26:323–369.

desJardins, M. E.; Durfee, E. H.; Jr., C. L. O.; and Wolverton, M. J. 2000. A survey of research in distributed, continual planning. *AI Magazine* 20(4):13–22.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

G2One, Inc. 2008. Groovy dynamic scripting language for the java virtual machine. Web Page : http://groovy.codehaus.org/. Last Checked: 18/11/2008.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical report, University of Brescia and University of Strathclyde.

Hoffmann, J. 2003. The Metric-FF planning system: Translating 'ignoring delete lists' to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Johnson, N.; Patron, P.; and Lane, D. 2007. The importance of trust between operator and auv: Crossing the human/computer language barrier. In *Proceedings of the IEEE Oceans Conference 2007 - Europe*.

Nau, D. S.; Smith, S. J. J.; and Erol, K. 1998. Control strategies in htn planning: Theory versus practice. In *Proceedings of the 1998 Conference on Innovative Applications of Artificial Intelligence*, 1127–1133. American Association for Artificial Intelligence (AAAI).

Pêtrès, C.; Pailhas, Y.; Patrón, P.; Petillot, Y.; Evans, J.; and Lane, D. 2007. Path planning for autonomous underwater vehicles. *IEEE Transactions on Robotics* 23(2):331–341.

Ridao, P.; Yuh, J.; Batlle, J.; and Sugihar, K. 2000. On auv control architecture. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)*, volume 2, 855–860.

SAUC-E Committee. 2008. Student autonomous underwater competition - europe 2008. Web Page: http://www.dstl.gov.uk/news_events/competitions/sauce/08/index.php. Last Checked: 18/11/2008.

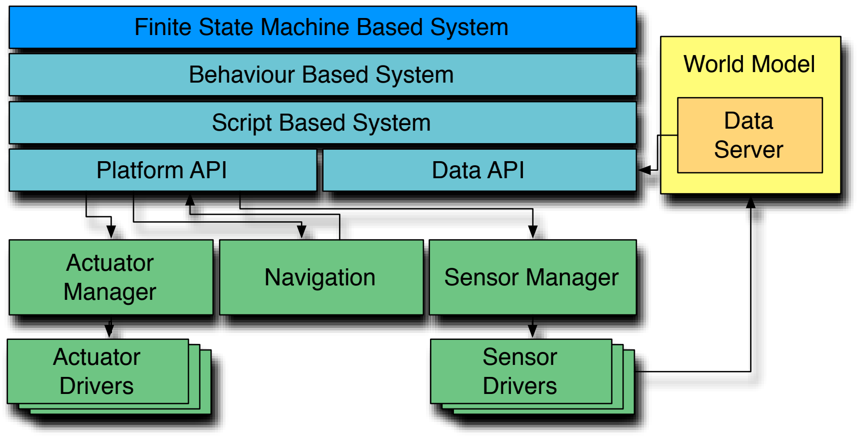

Figure 5: The simplified control system deployed on the Nessie III AUV.

Sotzing, C. C.; Evans, J.; and Lane, D. M. 2007. A multi-agent architecture to increase coordination efficiency in multi-auv operations. In *Proceedings of the IEEE Oceans Conference 2007 - Europe*.

Sotzing, C. C.; Johnson, N. A.; and Lane, D. 2008. Improving multi-auv coordination with hierarchical blackboard-based plan representation. In *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group*. (Accepted for publication).