Abstract

Planners provide extremely powerful functionality, allowing the user to state the goals of a system, and allowing it to formulate the specific actions. They can act more powerfully still when allowed to continually reassess a plan concurrently with its implementation. 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 carried out. 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 and completed.

In this paper we describe a system to create such a bridge between these layers of abstraction. This system is designed to fit into the larger framework posited in (Johnson, Patron, and Lane 2007). Since the publication of this paper, various advances have been made with 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, and the bridge between the planning (deliberative) and vehicle control (executive) layers will be concentrated upon.

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 sited 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 International Planning Competition. Entrants to this competition take input from the Planning Domain Description Language (or PDDL (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 its 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.

This competition largely focuses on advances on 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 ((Petrès et al. 2007)).

An excellent multiple robot deliberative control system 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 BHIMAPS plan representation (an augmented Hierarchical Task Network (see (Sacerdoti 1977), (Tate 1977) and (Erol 1995)) implementation), as well as limited modification of such a plan. This system relies on the plan being made up of a set of built in behaviours at present, however.

Implementation

The architecture described here is designed to facilitate planner based control of autonomous systems, and as such a
planner forms its uppermost layer. Figure 1 shows the complete architecture. 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 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.
- It is able to change the plan in response to changing circumstances.
- The system can control multiple vehicles and is completely decentralised.
- It selects the correct action (or no action at all) to be implemented for a particular vehicle and passes it down to the lower level systems.

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:\footnote{In practice these definitions contain additional information used for user interface purposes, but these have been stripped out here for clarity of meaning.}

```xml
<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 standard PDDL definition:

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

```java
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.

Each of these methods is given access to two APIs in order to carry out it’s task (with the addition of the parameter list in the `init` method). The first of these 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 currently 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 manages. These are simply small data bases which index each of the sensors and actuators my name, allowing information to be passed back and forth as quickly as possible.

Vehicle navigation is controlled via waypoint requests, with a waypoint representing 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 similar definitions. Methods are provided to allow the system to request a mode change and also to query which mode a sensor or actuator is currently in.

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.

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 and scalar values the existences in the world state relate to. This API also allows the script to add additional key/pair values to the Data Server, allowing some persistence of data between the different scripts.

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, but 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.

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 ((gro )), a powerful dynamic scripting language which compiles to pure Java at runtime. The backend needed to ensure that this functionality is available at runtime is provided by the scripting layer (the next layer down from the behaviour based layer). The groovy class definition for the example used here is shown below:

```java
class MoveControlScript
    implements BehaviourScript {
        LocalCoordinate3D waypoint
        int number

        void init(PlatformAPI platform,
                  DataAPI data,
                  String[] parameters) {
            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 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 the "continue" will be returned.

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

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 architecture (such as navigation or SLAM systems, and low level drivers) may be implemented in different languages, or distributed across multiple physical systems. For this reason, the system uses the OceanSHELL ((Oce 2004)) UPD packet based communication system to communicate with everything below the level of the Platform API.

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 vehicles system, provided that complementary instances

---

Figure 2: The Nessie III AUV passes through the validation gate at SAUC-E 2008. Image courtesy of Yves Gladu.

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.

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 below) and RAUVER (rau) AUVs was used as the navigation system and a full hydrodynamic model of RAUVER (citation to follow) 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 interacted with the simulated environment. The system performed perfectly in these circumstances, allowing two scenarios (one mine counter measures and one installation maintenance) to the completed, and with sufficient efficiency to allow to simulation to proceed at up to one hundred times realtime

The lower level systems were used as the basis of the control system used for the Nessie III AUV (Cartright et al. 2008) and Figures 2 and 3) as part of The Ocean System Laboratory’s entrance into the Student Autonomous Underwater Competition - Europe (or SAUC-E) 2008. The replanning system was replaced with a much simpler finite state machine based system, but was otherwise 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 a the name of a script and a set of parameters. This more sim-

3Faster than realtime simulation is a key property of the system described in (Johnson, Patron, and Lane 2007)
plified 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 here:


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.

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


Groovy dynamic scripting language for the java virtual machine.


Petrès, C.; Pailhas, Y.; Patrón, P.; Petillot, Y.; Evans, J.; and Lane, D. 2007. Path planning for autonomous underwater vehicles. IEEE Transactions on Robotics (Accepted for publication).


