Multi-Layered Vehicle Control via Payload Autonomy

Andrew Bouchard

I. ABSTRACT

While autonomy developers have been able to take advantage of external control interfaces on intelligent vehicles for years, they have typically had to choose between control architectures and interfaces that only allowed them to use the existing behaviors of the vehicle (task-level control) or those that allowed them to set desired control values (setpoint control) but then required them to re-invent and re-tune behaviors that might already be part of the vehicle’s native programming. As external control interfaces become increasingly open and allow developers access to both types of control, a control architecture is necessary that takes advantage of the strengths of each method and can switch between them as appropriate. We present one solution to this challenge that not only accomplishes these goals but is also vehicle agnostic and extendable to an arbitrary number of low-level controllers.

II. INTRODUCTION

Past efforts to navigate intelligent vehicles using an external control interface have relied on control paradigms that either use low-level setpoint commands for immediate action or the delegation of entire tasks to the vehicle. The use of setpoint commands (e.g., desired heading, speed, and depth for an unmanned underwater vehicle) provides the greatest degree of control over the actions of the vehicle, but also requires the external controller to re-invent and re-calibrate capabilities that already exist on the platform. Delegation of full tasks to the vehicle allows the external controller to leverage high-level behaviors already proven on the platform or that cannot be accomplished using setpoint commands, but limit the external controller to those behaviors already defined. It would be preferable to be able to take advantage of the benefits of both modes of control without losing the advantages of the other.

In this work, we present a new paradigm that allows an autonomy control architecture to dynamically switch between task and setpoint commands to a vehicle by switching controllers on a per-task basis. Essentially, the system determines whether a given task will be decomposed into more low-level behaviors by the vehicle or by the external controller. This is accomplished by using a task discrimination component as a multiplexer, which uses a vehicle-specific configuration to route tasks to either a task controller that delegates the whole operation to the vehicle or to a behavior manager that decomposes the task into behaviors that produce the setpoint commands to accomplish the task. While both of these types of controllers have been demonstrated in other applications, we were not able to find any existing examples of both being used on the same platform.

Delegation of the entire task requires the payload computer to issue a command and wait for completion to be reported by the vehicle; setpoint control requires continuous recalculation and transmission of commands. In order to support these different modes of commanding, monitoring, and responding to the vehicle, the vehicle interface required modification as well. The interface was built on iFrontSeat, an open source vehicle interface component from GobySoft that already provided software for setpoint commands without monitoring for feedback. This was extended to add a state for accepting task commands that also monitors for task updates and completion as reported by the vehicle. Logic to switch between these methods uses the states of both controllers and input from the Task Discriminator.

This paradigm has been successfully tested in simulation with a variety of tasks and configurations, and is able to switch back and forth between task and setpoint commands without loss of control. In this paper, we present an overview of the design of our system, challenges of implementation, and experimental results, as well as a discussion of future work to extend this capability.

This paper will explain the implementation of this multi-layered control system for payload autonomy from the top down. Section III provides a summary of existing payload autonomy work and Section IV describes the overall architecture used for multi-level control. Section V details the design of the parallel controllers and Section VI the structure to route task execution messages to the appropriate controller, while Section VII explains implementation in the vehicle interface. Some results are provided in Section VIII and Section IX summarizes the work done and future opportunities.

III. BACKGROUND

Payload autonomy, also known as “backseat” control, is a paradigm in which control of a robotic vehicle is split between the main vehicle computer (MVC) containing some core capabilities and a payload computer that collects information and issues requests and commands via an external interface defined by the vehicle manufacturer. Early implementations were developed to support relatively simple adaptive behaviors such as plume following and reactive...
obstacle avoidance[4]. Since then, however, research and commercial groups around the world have begun to leverage the technology for more sophisticated purposes.

A number of groups are investing generally in robust autonomy control architectures[5], [6], [7], [8] that take advantage of these payload interfaces to permit increasingly higher levels of capability. These architectures tend to be agnostic to any particular vehicle, and require an interface to convert the decisions of the architecture into commands for the specific host vehicle. Other efforts have applied the payload autonomy paradigm to address specific challenges, such as the establishment of communication networks[9], multi-vehicle collaboration[10], and trajectory estimation[11].

An increasing number of vehicle platforms offer an external control interface for payload control[12], [13], [14], [15], and as customers request more access to and capabilities from the base vehicle, the methods of interfacing with them will have to be both more flexible and more robust to take full advantage of the potential they offer. This is especially true as development moves into higher levels of cognition and reasoning such as goal-based autonomy[16]; developers should be able to use as many existing capabilities as possible to minimize the additional work required.

IV. AUTONOMOUS VEHICLE ARCHITECTURE (AVA)

While the concept described in this paper could be implemented in a variety of different architectures, the Autonomous Vehicle Architecture (AVA) developed at the Naval Surface Warfare Center Panama City Division[17] was used for convenience. This architecture provides a division between higher-level tasks and lower-level behaviors, which allows the division between setpoint and task generation to be segregated to the lowest level of control. The baseline architecture is implemented in either the Mission-Oriented Operating Suite (MOOS) or Robotic Operating System (ROS) and already incorporates the Interval Programming (IvP) method of multi-objective optimization for generation of setpoints. The baseline form of the architectures is shown in Figure 1.

Extending AVA to include task-level control required the addition of a low-level controller in parallel with the IvP Behavior Manager to generate task commands rather than setpoint commands. Next, in order to deliver the instruction from the higher-level task to the correct controller, a task discriminator component was created. Finally, the vehicle interface, which previously only handled setpoint commands, was expanded to support passing of task commands as well.

V. PARALLEL LOW-LEVEL CONTROLLERS

In the AVA architecture, the IvP Behavior Manager handles the job of configuring the IvP behaviors in order to accomplish the execution instruction from the higher-level task and produce the desired setpoint values. Likewise, the Vehicle Task Manager (VTM) is charged with creating the task commands that will produce the desired result from the vehicle based on that instruction. Based on task-level commands accessible via payload interface on commercially available vehicles, the VTM currently supports four task types: Transit, Loiter, Bottom, and Cancel.

The VTM is designed to receive a task execution message of one of these types from a higher-level task component. This message is described in Listing 1, and describes where the vehicle should execute the desired task and under what constraints. Task constraints are monitored by the VTM during execution, and if violated can cause the task to be aborted. The instructions for task execution are put into a generic CommandRequest message with a DesiredTask, shown in Listing 2. This is the message provided to the vehicle controller.

![Figure 1: A high-level diagram of the Autonomous Vehicle Architecture (AVA).](image)

Listing 1: Partial description of the Task message for execution instruction from high-level tasks to low-level controller.

```plaintext
message Task {
    enum EType { Transit, Loiter, Bottom, Cancel }
    enum ExecutionMode { Start, Delay, Pause, Restart, Abort, Completed, Failed, InProgress }
    uint32 identifier
    EType type
    ExecutionMode execution_mode
    ExecutionMode execution_status
    Constraint[] constraints
    Waypoint[] points
}
```

An execution message from a higher-level task may contain a sequence of tasks, such as multiple waypoints to traverse. In order to increase the flexibility of the system and support vehicles whose interface might only allow for execution of one task at a time, the current implementation requires the VTM to break each task execution into the smallest level of discrete tasks to create a queue. It must then monitor each task for completion before either sending the next task from the queue or reporting completion of the entire queue back up to the higher-level task. The VTM also handles task timeouts when no response is received from the vehicle and repeating requests rejected because the vehicle was not yet ready to accept commands.
message CommandRequest {
    DesiredCourse desired_course
    DesiredTask desired_task
    bool response_requested
    int32 request_id
}

message DesiredCourse {
    double time
    double heading
    double speed
    double depth
}

message DesiredTask {
    enum TaskType {Transit, Loiter, Bottom, Cancel}
    double time
    uint32 identifier
    double speed
    double depth
    Waypoint[] waypoints
    Constraint[] constraints
}

Listing 2: Partial description of the CommandRequest message for task-level commands from low-level controller to vehicle interface.

VI. TASK DISCRIMINATION

With multiple low-level controllers capable of executing tasks, some means is necessary that will indicate which task should go to which controller. While this could simply be handled by the Task Manager, it was decided that breaking the function into a separate process would allow greater flexibility and modularity in the long run. Because the IvP Behavior Manager and IvP Helm are a part of the default architecture, they are assumed to always be present and are used as the default controller. In addition, we imagine that future projects may wish to extend the system by creating low-level controllers that handle tasks differently; therefore, we also set out to build a discriminator that would support an arbitrary number of low-level controllers with minimal changes to the rest of the system.

The result is a Task Discriminator (TD) that, when started, builds a map of low-level controllers and their parameters based on a configuration block, like that shown in Listing 3. Each controller is described by a prefix that is used to generate a unique publish-subscribe connection between the TD and the controller through which the discriminator sends tasking and the controller reports progress. Each controller is assigned a control type, currently either Setpoint or Task, and a list of task types that will be assigned to that controller. When the TD receives a new Task from the Task Manager, it queries the map to determine which controller should receive the Task and sends it accordingly. That controller’s prefix is then registered as the active controller, and any status messages that come from all the low-level controllers will be filtered, with only those from the active controller proceeding to the Task Manager. In addition, the active controller prefix and control type are posted to the vehicle interface, which uses this information to enter the correct control state and ignore commands from all but the active controller. This is described in greater detail in Section VII.

Listing 3: Example configuration block for the Task Discriminator, describing a system with three low-level controllers with prefixes IVP, VEH, and FOO that provide setpoint control, task control, and setpoint control, respectively. This system would send Transit and Bottom tasks to the VEH controller, Loiter tasks to the FOO controller, and all others to the IvP controller.

Importantly, the filtering mechanisms in both the Task Discriminator and the Vehicle Interface allow the IvP controller to continuously run a safe behavior in case of failure. Currently, that safety behavior is a small loiter circle around the last commanded location. The TD defaults to the IvP controller, meaning that on startup or as soon as the task assigned to any other controller is reported complete, the IvP controller becomes the active controller and the safety behavior takes over. This has proven particularly useful when a computationally intensive job must be completed before execution of the next task, such as dynamic replanning.

VII. VEHICLE INTERFACE

In keeping with the ideas of modularity and flexibility of design, the vehicle interface for the system started with iFrontSeat[1], designed to be a generic vehicle interface implementing key functions of the backseat-frontseat relationship in a general fashion and requiring a vehicle-specific driver only for translation of messages and the most specific adaptations. Much of the core functionality of iFrontSeat as provided by GobySoft is captured by its state machine, shown in Figure 2.

The state machine captures determination of the main vehicle computer (“frontseat”) state, controller state, and the logic transitions that determine which system is in control of vehicle behavior. However, as originally imagined, iFrontSeat was limited to setpoint control, continuously sending target navigation values to be achieved without monitoring for direct feedback from the vehicle. In order to support task-level control, the state machine had to be extended to support another mode of operation that sends
a higher-level command once and then tracks its execution. The result is a new state machine shown in Figure 3.

The extended state machine is shown in Figure 3, and clearly shows the separation of the command state into two: setpoint command and task command. The former functions exactly as the original, continuously sending DesiredCourse information (see Listing 2) to the vehicle as it is provided by the controller. The task command state, on the other hand, sends a request for an entire task from the vehicle and awaits a response. Rejections are forwarded to the VTM, while accepted tasks are monitored for progress and completion. As with the general command state, the state of both the frontseat and the active controller are tracked at all times to detect and respond to errors and ensure that messages are sent only when the necessary components to execute them are able to accept them.

VIII. RESULTS

Using a simulated vehicle supporting the Bluefin Standard Payload Interface (SPI), a simple mission was executed consisting of a Transit of two legs to a location, a Bottom task at the destination, and a Transit back to the start location by reversing the same two legs. The AVA architecture was used with the IvP Helm for setpoint control and Vehicle Task Manager for task control as described above. While both of these controllers are capable of completing a Transit task, only the VTM could execute the Bottom task. The results of two simulated missions are shown in Figure 4. In the first test (Figure 4a), the system was configured to use the IvP Helm to execute Transit tasks, while the second (Figure 4b) used the VTM. Both missions executed the Bottom task using the VTM.

The slight north-east jog in Figure 4a is caused by a slight disparity between how coordinates are calculated by the vehicle versus by the IvP Helm, and is being addressed. Overall, both missions were successful in that the vehicle reached all the desired waypoints and bottomed in the intended location. Loops in the vehicle path seen in both missions are the default loiter described in Section VI taking over while the vehicle transitions between tasks. Therefore, even though all the tasks were configured to be executed by the VTM in the second mission, both missions still saw multiple transitions between setpoint and task control.

IX. CONCLUSION AND FUTURE WORK

In this paper, we have described a new paradigm to allow in-stride switching between setpoint and task-level control depending on the needs of the mission. The implementation supports the addition of an arbitrary number of low-level controllers and is vehicle agnostic, using the generalized state machine from GobySoft iFrontSeat to provide a generic vehicle interface. The paradigm has been tested in simulation, and will be shown in-water in August 2016. The approach allows an external autonomy controller to leverage both existing behaviors on host vehicles and access low-level commands for development of new behaviors interchangeably.

Current efforts to enhance this capability are largely focused on error handling and automated configuration. The system presently fails during execution if a task is configured to be executed by a controller not capable of doing so, and offers no options for recovery. This error is avoidable and should be detected prior to mission start, but requires some
administrative component with knowledge of the configuration of the Task Discriminator and the capabilities of each of the low-level controllers. A more intelligent system would also know the goals of the mission and which controller worked better under which conditions, allowing the system to potentially configure itself autonomously.

REFERENCES


