A Behavior-Based Intelligent Control Architecture with Application to Coordination of Multiple Underwater Vehicles \(^1\) \(^2\)

Ratnesh Kumar  
Dept. of Electrical Engineering, Univ. of Kentucky  
Lexington, KY 40506  
Email: kumar@engr.uky.edu  

James A. Stover  
Applied Research Lab., Pennsylvania State University  
University Park, PA 16802  
Email: jjs5@psu.edu  

November 20, 2005

\(^1\)The work was completed while the first author was on a sabbatical leave at Applied Research Laboratory (ARL) on an invitation from Prof. Asok Ray of PSU and Dr. Shashi Phoha of ARL. The support was provided by the University of Kentucky, and by ARL through an Office of Naval Research (ONR) grant. The authors wish to thank Dr. Jerry Shelton of ARL for his valuable inputs in formalizing some of the concepts of the intelligent control architecture, and to Mr. Atilla Kiraly for assisting with developing the figures that model the response controller of the underwater vehicles.

\(^2\)This research is supported in part by the National Science Foundation under Grants NSF-ECS-9409712, NSF-ECS-9709796, and in part by the ONR under the Grant ONR-N00014-96-1-5026.
Abstract

The paper presents a behavior-based intelligent control architecture for designing controllers which, based on their observation of sensor signals, compute the discrete control actions. These control actions then serve as the “set-points” for the lower level controllers. The behavior-based approach yields an intelligent controller which is a cascade of a perceptor and a response controller. The perceptor extracts the relevant symbolic information from the incoming continuous sensor signals, which enables the execution of one of the behaviors. The response controller is a discrete event system that computes the discrete control actions by executing one of the enabled behaviors. The behavioral approach additionally yields a hierarchical two layered response controller, which provides better complexity management. The inputs from the perceptor are used to first compute the higher level activities, called behaviors, and next to compute the corresponding lower level activities, called actions. This paper focuses on the discrete event subsystem, namely the response controller. We illustrate the intelligent control architecture by discussing its application to the design of intelligent controllers for autonomous underwater vehicles used for ocean sampling missions. A complete set of discrete event models of the response controller of the underwater vehicles for the above application has been given, and their formal verification discussed.

Keywords: Intelligent control, discrete event control, hierarchical control and coordination, autonomous underwater vehicles, ocean sampling
1 Introduction

Many of the dynamical systems that need to be controlled, called plants, are complex, large-scale, highly non-linear, time-varying, stochastic, and operate in an uncertain and unpredictable environment. As a result of these characteristics, dynamical systems are not amenable to accurate modeling. Hence, conventional control techniques that are model-based, i.e., rely on the plant model, are not suitable for the controller design for such systems. There are many difficulties in extracting a model of a plant using physical laws which include: (i) plant behavior is too complex to understand, (ii) models are difficult or expensive to evaluate, (iii) plant behavior is subject to unpredictable environmental disturbances, (iv) plant behavior is distributed, non-linear, time-varying, stochastic.

Moreover, conventional controllers are designed primarily for the purposes such as stabilization, regulation and tracking, robustness, disturbance rejection, model matching, performance optimization, and use the plant model for control design. However, the control requirements for complex systems are far beyond those mentioned above and include additional requirements such as reconfigurability, learning capability, safety, failure and exception handling, capability to manage dynamically changing mission goals, multi-system coordination, increased autonomy, many of which are elastic (such as control of room temperature).

The inadequacy of the conventional control techniques for the reasons described above, namely, the need to (i) control systems without their accurate models, and (ii) meet specifications beyond the scope of conventional control, has led to research into the “non-conventional” control techniques, also called intelligent control. Intelligent control offers an alternative to conventional control for designing controllers whose structure and consequent outputs in response to external commands and environmental conditions are determined by empirical evidence, i.e., observed input/output behavior of the plant, rather than by reference to a mathematical or model-based description of the plant. For an exposure to intelligent control techniques readers are referred to the edited volumes [10, 21, 5, 9].

There is little to be gained by intelligent control when the plant model is well known and control requirements fall within the scope of conventional control. For this reason the control is generally hierarchically structured as is shown in Figure 1, where at the lower level conventional control is exercised, whereas at the higher level intelligent control is used. The

![Figure 1: Control hierarchy with an intelligent controller](image-url)
lower level conventional controllers are model-based, offering conventional control capabilities; the higher level intelligent controllers on the other hand, operate on models constructed through empirical evidence offering control capabilities beyond the purview of conventional control. The tasks are delegated from the higher level to the lower level, whereas the sensory feedback is passed on from the lower level to the higher level.

Given that intelligent control is used when plant models are ill-defined and control requirements are beyond the scope of conventional control, such controllers are inherently non-linear. Several techniques for such non-linear controller design have been proposed in literature, which include expert systems, fuzzy logic systems, artificial neural networks, genetic algorithms.

Although various alternative techniques for intelligent control are being actively researched, there is little research effort directed towards the design of intelligent control architectures.

One such architecture by Saridis [19] is hierarchical with three layers: the execution layer at the bottom, the coordination layer in the middle, and the organization layer at the top. One of the main ideas of this architecture is the principle of increasing intelligence with decreasing precision. Meystel [17] has proposed a nested hierarchical control architecture for the design of intelligent controllers. A model-based autonomous systems architecture by Zeigler-Chi [22] consists of models of planning, operations, perceptions, diagnostics, and recovery. These models are systematically used for achieving control with high degree of autonomy. A general approach to task-based model development is also presented. An architecture consisting of a network of intelligent nodes is proposed by Levis [16] as a model for distributed intelligent system. Intelligence in each node is the consequence of its five-stage model, namely, situation assessment, information fusion, task processing, command interpretation, and response selection.

Another architecture, called real-time control system (RCS) reference model architecture is by Albus [4]. RCS is also arranged in a hierarchy, where each node in the hierarchy performs sensor processing, value judgment, world modeling, and behavior generation at a level of abstraction and resolution appropriate for the position of the node in the intelligent control hierarchy. The sensor processing subsystem receives sensor inputs from the environment and their predicated values from the world modeling subsystem, and provides updates for the world modeling by comparing the two signals. It also extracts relevant features out of the sensor inputs, which are evaluated for significance by the value judgment subsystem and then passed on to the world modeling subsystem for updating its database. The behavior generation subsystem generates a plan-of-actions based on both the current state available from the world modeling subsystem and its knowledge of the mission goals. The world modeling subsystem receives the plan from the behavior generation subsystem, simulates it, and sends the results of simulation to the value judgment subsystem for evaluation. The value judgment subsystem then passes on its plan evaluations to the behavior generation subsystem. Once a plan is finalized, the behavior generation subsystem issues actuator commands. Another control computation architecture, called cerebellar model articulation controller (CMAC), was also proposed by Albus [3, 2] to model control computations in
intelligent biological systems.

A *structure-based hierarchical architecture* is proposed by Acar-Ozguner [1]. It embeds intelligence in control via a special hierarchical organization based on the physical structure of the system. Each node in the hierarchy is systematically assigned its intelligence or functionality through its accomplishable tasks and the procedures to accomplish them.

Another architecture, called *subsumption architecture*, is by Brooks [6]. This architecture is based on the idea of levels of increasing competence of an intelligent system, which need to be identified in the beginning of the design phase. The design then proceeds by constructing a control system that achieves the level-zero competence. Next, a new control layer is added which examines signals from level-zero and also injects signals into level-zero to either inhibit or to replace an existing signal. This layer with the aid of level-zero achieves the level-one competence, and so on.

In this paper we present a new architecture, called *behavior-based intelligent control architecture*, for designing intelligent controllers. The controllers based on their inputs of sensor signals and mission goals compute control actions. As shown in Figure 1, these control actions then serve as “set-points” for the lower level conventional controllers. The intelligent control architecture is a cascade of four subsystems: the input interface, the perceptor, the response controller, and the output interface. The perceptor extracts the relevant symbolic information from the incoming continuous sensor signals, while the response controller is a discrete event system [7, 12] that computes discrete control actions in response to the discrete inputs from the perceptor.

Our approach to the design of intelligent controllers is *behavioral*. Behaviors are certain high level activities (that are independent of each of other) that determine the manner in which the system reacts to changing external/environmental conditions and thereby executes subtasks of the given mission tasks. Behaviors can be configured in different modes, and each behavior mode can be executed by the execution of certain primitive activities, called actions. This behavioral approach to intelligent control design naturally yields:

1. The perceptor subsystem which has the task of extracting those features from incoming continuous sensor signals that enable the execution of one of the behaviors.

2. The response controller subsystem with a two layered hierarchy, where the lower layer consists of behavior controllers (one per behavior) and the higher layer consists of a single coordinator. An implicit advantage of such a hierarchically structured response controller is that it provides a better complexity management.

Thus in our approach the design of an intelligent controller requires identifying certain high level activities, the behaviors, that the system under control should exhibit; next identifying the different modes of each behavior; and finally identifying the primitive actions which, when executed in an appropriate sequence, execute a specific behavior mode. An action is a primitive activity that is executed by executing an unique associated algorithm that computes the set-points for the lower level controllers. The perceptor is designed so as to identify the external/environmental conditions that lead to the enablement of the appropriate behaviors in the response controller.
This paper focuses on the details of the response controller subsystem; an introduction to the perceptor subsystem can be found in [20]. Section 2 describes an application—ocean sampling using a network of autonomous underwater vehicles, where the intelligent controllers for the underwater vehicles have been designed in the architecture presented here. An overview of the intelligent control architecture is presented in Section 3. The architectural details of the response controller is discussed in Section 4. Section 5 discusses the interacting automata based discrete event system model of the response controller that is used for its analysis and verification. Section 6 details the intelligent control design for the autonomous underwater vehicles in our architecture and gives the discrete event model of the response controllers of the underwater vehicles, whereas the control (mission) specification models used for verification are presented in Section 7. Section 8 concludes the work presented here, and a preliminary introduction to discrete event systems is presented in the appendix. Some results of this paper were presented in the conference papers [13, 14].

2 An application of intelligent control architecture

We use a SAMpling MObile Network (SAMON) of autonomous underwater vehicles (AUVs) to illustrate the intelligent control architecture [18]. The controller for each of the AUVs in this application is designed in the intelligent control architecture presented here. A schematic diagram of SAMON is shown in Figure 2.

A typical ocean sampling mission begins by seeding the ocean floor of the region of interest with a number of fixed sensor packages (FSPs) and deploying a group of autonomous underwater vehicles (AUVs) to explore this region and gather data from FSPs. FSPs are capable of sensing parameters of interest, data recording, and communicating the recorded data by sonar to AUVs, which explore the region. They may also serve as position references for the group of AUVs. AUVs are capable of navigating from one point to another, and can communicate with each other and with FSPs by sonar. The group of AUVs consists of a certain number of supervisory AUVs (SAUVs) which can communicate by radio to the tactical coordinator (TC), located on shore, and can communicate by sonar with their subordinate AUVs.

Figure 2: Schematic diagram of Ocean SAMON
As shown in Figure 2, SAMON is a four layered hierarchy consisting of the TC, SAUVs, AUVs, and FSPs. Once the group of AUVs are deployed, they perform an initial self-organization in which each SAUV determines the number and locations of its subordinate AUVs through sonar queries, and then surfaces to report its availability for ocean sampling operations to the TC by radio. Once the TC has been contacted by all SAUVs and has determined the resources available, it issues mission orders (such as gather data, relocate, etc.) to each SAUV, and also provides search region coordinates. Each SAUV distributes the search region among its subordinate AUVs, and determines a rendezvous point and a time to meet afterwards. Each subordinate AUV then develops an itinerary for exploring its own subregion, and gathers sampled data from FSPs within that subregion. At the rendezvous point each SAUV makes sonar contacts with its subordinates, collects any data, and surfaces to make radio contact with the TC to download the data and receive further mission instructions.

SAMON is comprised of a reconfigurable four layered hierarchical and distributed architecture of command-control-communication AUVs. Each node in this hierarchy is an intelligent controller that, based on its observations of the sensor signals and signals from lower level controllers, computes the discrete control actions which serve as set-points for the lower level controllers and its own effector subsystems. We apply the intelligent control architecture presented in this paper for designing the controller for a typical AUV/SAUV.

3 Intelligent control architecture

As described above, the intelligent control architecture is used for designing controllers, which based on their observations of sensor signals, compute discrete control actions that serve as set-points for the lower level controllers. The intelligent controller is a cascade of four subsystems—input interface, perceptor, response controller, and output interface—as shown in Figure 3 We discuss the functionality of each of the subsystems in the following.

![Figure 3: Systems within the intelligent control architecture. By convention, rectangular blocks depict systems, whereas the oval blocks depict signals.](image-url)
**Input interface:** The intelligent controller receives sensor data in the form of certain data packets over a network after the raw sensor signals have been processed by an appropriate signal processing system. The input interface reads the sensor data packets off of the network and stores them in an appropriate form in `SENSOR_DATA`. This is achieved by the execution of the function-call event `get_sensor_data`.

**Perceptor:** Perceptor reads the data from `SENSOR_DATA`, extracts the features of interest, and stores them in appropriate forms in `STIMULUS_DATA` by transforming continuous signals into discrete symbols. A certain fuzzy pattern classifier, called *continuous inferencing net (CINET)* [20], is used for this purpose. The perceptor is invoked by the execution of the function-call event `do_perception`.

**Response controller:** Response controller reads the data from the `STIMULUS_DATA`, computes the discrete control actions, and stores them in `CONTROL_DATA`. It is a discrete event system that receives discrete sensor symbols, maintains discrete states, and outputs discrete control actions. The response controller computes the control actions hierarchically to manage complexity, where the `STIMULUS_DATA` inputs are used to first compute the higher level activities, called `behaviors`, and next to compute the corresponding lower level activities, called `actions`. This correspondence is shown in Figure 5, where the response controller consists of: (i) a top level `coordinator` that based on the `STIMULUS_DATA` inputs selects the behaviors to be exhibited, and (ii) several lower level `behavior controllers` (one for each behavior) that determine the control actions for exhibiting the selected behaviors. The response controller computes the control action each time the event `compute_response` is executed. This computation, in turn, is achieved by a sequence of events (in form of function-calls) shown in Figure 6. Details of the functions are given below.

**Output interface:** The output interface reads the control actions stored in `CONTROL_DATA` and generates a data packet in an appropriate form for the network. The network receives the control action data from the intelligent controller and transmits them as the set-points for the lower level controllers. This process is achieved by the execution of the function-call event `send_control_data`.

A more detailed description of the two interfaces and the perceptor is beyond the scope of this paper. The response controller only is described in detail. An introduction to the CINET based perceptor is presented in [20].

The sequence of function-calls `get_sensor_data`, `do_perception`, `compute_response`, and `send_control_data` is executed to invoke the input interface, the perceptor, the response controller, and the output interface, respectively. These executions are done each time either a new sensor data packet arrives or a time-out occurs. In other words, the intelligent controller waits for a new sensor data packet to arrive for a maximum time determined previously, and when a sensor data packet arrives within this time or when the time-out occurs, it executes the sequence of function-call events. It is possible for a new sensor data
packet to arrive before the last complete controller cycle is executed, in which case a new controller cycle is initiated pre-emptying the execution of the last one. The diagram is shown in Figure 4.

![Event sequence the controller cycles through](image)

Figure 4: Event sequence the controller cycles through

4 Response controller

The response controller is the subsystem in the intelligent control architecture that is responsible for making the control decisions, i.e., computing the control actions.

Our approach to designing the response controller is a “behavioral approach”. Behaviors are certain high level activities (that are independent of each of other) that determine the manner in which the system reacts to changing external/environmental conditions and thereby executes subtasks of the given mission tasks. For example, in the SAMON application, the overall mission is to sample a given region of the ocean floor by using a network of SAUVs and AUVs. In order to perform this mission, each SAUV should be able to exhibit the following behaviors:

- Form its team of AUVs
- Communicate with the TC
- Communicate with the AUVs in its team
- Reconnaissance
- Refuel
- Avoid collisions
- Loiter

Similarly, each AUV should be able to exhibit the following behaviors:

- Communicate with its SAUV
- Communicate with the FSPs
- Reconnaissance
- Refuel
- Avoid collisions
- Loiter

Thus the design phase begins with identifying such higher level activities, called behaviors, that the system under control should exhibit for executing subtasks of the given missions. A certain behavior may be configured in different modes. As an example of modes in the SAMON application, the SAUV behavior “communicate with the TC” may be configured in one of the following modes:

- Communicate with the TC to establish a contact
- Communicate with the TC to download the gathered data

Once the set of behaviors and the associated modes have been identified, we next identify the lower level primitive activities, called actions, with the property that each behavior mode can be executed by executing a certain sequence of primitive actions. For example in the SAMON application, the SAUV behavior mode “communicate with the TC to establish contact” is executed by executing the following sequence of primitive actions:

- surface, send hello by radio, receive acknowledgment on radio

On the other hand, the SAUV behavior mode “communicate with the TC to download the gathered data” is executed by executing the following sequence of primitive actions:

- surface, send download request by radio, receive download acknowledgment on radio, transmit gathered data on radio

Thus actions are primitive or atomic activities whose execution in a certain sequence results in the execution of a particular behavior mode. An action is primitive in the sense that it specifies a unique set-point for the lower level controllers that are controlled by the intelligent controller (refer to Figure 1). For example in the SAMON application, the “surface” action specifies the set-point of zero depth, and zero speed for the auto-pilot controller. Similarly, the action “send hello by radio” action specifies the set-point of “hello” message for the radio communication controller.

To summarize, the design of an intelligent controller requires identifying certain high level activities, the behaviors, that the system under control should exhibit; next identifying the different modes of each behavior; and finally identifying the primitive actions which, when executed in an appropriate sequence, execute a specific behavior mode. An action is a primitive activity that is executed by executing an unique associated algorithm that computes the set-points for the lower level controllers. The perceptor is designed so as to
identify the external/environmental conditions that lead to the enablement of the appropriate behaviors in the response controller.

This behavioral approach for designing intelligent controllers naturally leads to a response controller that has a two layered hierarchy. The lower layer consists of behavior controllers, one for each behavior, and the higher layer consists of a single coordinator. This hierarchy is shown in Figure 5.

The coordinator receives sensor inputs (orders) from the STIMULUS_DATA, selects the behavior to execute, suggests the mode of the behavior execution, monitors the progress of behavior execution, modifies its selections in case of problems, and sends reports out in case of failures. The task of a behavior controller on the other hand is to monitor sensor inputs and accordingly advise the coordinator of its enablement, receive mode configuration inputs from the coordinator, select the behavior mode to execute, issue set-point inputs for the lower level controllers for executing the corresponding sequence of primitive actions, monitor the progress of action sequence execution, and inform the coordinator in case of failures.

The functionality of the coordinator and the behavior controllers within the two layered response controller described in the previous paragraph is achieved by a combination of functions (drawn in rectangular blocks in Figure 5) and the signals they process/generate (drawn in oval blocks in Figure 5). Figure 5 depicts the domain and range spaces (drawn as oval blocks) of each function (drawn as rectangular blocks) within the response controller. We next discuss these functions individually.

In order to maintain uniformity, the coordinator and the behavior controllers are structured identically. Each consists of two functions, the enabler function and the planner function, and maintains internal state variables called directives.

Coordinator enabler: This is the enabler function in the coordinator. It checks for the arrival of any new orders in the STIMULUS_DATA from a higher level (such as the TC level for the SAUVs, and the SAUV level for the AUVs in the SAMON application), and enables the appropriate behaviors by setting the appropriate COORDINATOR_DIRECTIVES in the coordinator. One of the enabled behaviors is later selected for execution.

Behavior enabler: Each behavior has its own behavior enabler function which checks for the arrival of any relevant input in the STIMULUS_DATA that implies the particular behavior be enabled. The behavior enabler function then sets the appropriate COORDINATOR_DIRECTIVES, informing the coordinator that the particular behavior is “enabled” in response to the arrival of a certain input in the STIMULUS_DATA.

Coordinator planner: The task of the coordinator planner is two fold. Firstly, depending upon its current state of COORDINATOR_DIRECTIVES, which contains the information about the set of enabled behaviors, it generates a COORDINATOR_PLAN, a sequence of enabled behaviors, in which the enabled behaviors should be executed. It thus assigns an order to the set of enabled behaviors. Secondly, it sets BEHAVIOR_DIRECTIVES in the enabled behaviors for configuring their modes of execution.
Figure 5: Details of the response controller
Behavior planner: Associated with each behavior is its own planner. Whenever the current COORDINATOR PLAN begins with planner’s owner behavior, then based on the current BEHAVIOR_DIRECTIVES, which contain information about the enabled behavior modes, the planner selects a behavior mode and generates the BEHAVIOR_PLAN corresponding to that mode. A BEHAVIOR_PLAN is a sequence of actions, in which the selected behavior mode is to be executed.

Action executor: The action executor checks the current BEHAVIOR_PLAN and executes the first action in it. The execution of the action computes the appropriate set-points for the lower level controllers using any pertinent information from the STIMULUS_DATA, and outputs them as CONTROL_DATA. The output interface function send_control_data, when executed, reads the CONTROL_DATA, forms an appropriate data packet, and sends it out on the network as set-points for the lower level controllers.

Action evaluator: This function evaluates the success of the last action executed. It checks the current BEHAVIOR_PLAN to determine the first action in it. (Initially, when the BEHAVIOR_PLAN is empty, the action_evaluator acts vacuously.) It then compares the assigned set-point in the CONTROL_DATA with the result of the lower level control reported as a new sensor data in the STIMULUS_DATA. If the action was successfully executed, it deletes the action from the BEHAVIOR_PLAN, otherwise it sets an appropriate BEHAVIOR_DIRECTIVE to inform the behavior controller.

Behavior evaluator: This function evaluates the success of the last behavior executed. It checks the current COORDINATOR_PLAN to determine the first behavior in it. (Initially, when the COORDINATOR_PLAN is empty, the behavior_evaluator acts vacuously.) It then checks the corresponding BEHAVIOR_DIRECTIVES to determine if there was any problem with the execution of the last action. If there was a problem that the behavior controller was unable to resolve, it sets an appropriate COORDINATOR_DIRECTIVE to inform the coordinator. Otherwise, it checks the BEHAVIOR_PLAN, and if empty, determines that behavior was successfully executed, in which case it deletes the behavior from the COORDINATOR_PLAN.

To summarize, we list each function in the response controller along with its domain and range space:

```
coordinator.enabler : STIMULUS_DATA →
                     COORDINATOR_DIRECTIVE
behavior i.enabler : STIMULUS_DATA →
                     COORDINATOR_DIRECTIVE
coordinator.planner : COORDINATOR_DIRECTIVE →
                     COORDINATOR_PLAN × Π_i BEHAVIOR_i_DIRECTIVE
behavior i.planner : COORDINATOR_PLAN × BEHAVIOR_i_DIRECTIVE →
                     BEHAVIOR_PLAN
```
As described above the response controller gets invoked through the execution of the event (function-call) \( \text{compute\_response} \), which gets executed each time the controller cycles through the sequence of events shown in Figure 4. This happens each time the intelligent controller receives a new sensor data packet or a time-out occurs. The function \( \text{compute\_response} \) is a “macro-event” whose functionality is achieved by the functions of the response controller executed in a certain sequence shown in Figure 6.

![Event sequence that achieves compute\_response](image)

Figure 6: Event sequence that achieves \( \text{compute\_response} \)

This sequence of execution of the functions can be classified into the phases of activities of situation evaluation, operation planning, and operation execution as discussed below:

**Situation evaluation:** This is achieved in two steps. Firstly, the effect of the last action is evaluated by executing the \( \text{action\_evaluator} \) followed by the \( \text{behavior\_evaluator} \). The execution of these functions modifies the \( \text{COORDINATOR\_PLAN} \) and \( \text{BEHAVIOR\_PLAN} \) if the action was successful, and generates appropriate \( \text{COORDINATOR\_DIRECTIVES} \) and \( \text{BEHAVIOR\_DIRECTIVES} \) otherwise.

Secondly, the arrival of a new input in the \( \text{STIMULUS\_DATA} \) is evaluated and then the appropriate behaviors are enabled by executing the \( \text{coordinator\_enabler} \) followed by the \( \text{behavior\_enabler} \) for each of the behaviors. The execution of these functions generates appropriate \( \text{COORDINATOR\_DIRECTIVES} \) and \( \text{BEHAVIOR\_DIRECTIVES} \).

**Operation planning:** This is achieved by executing the \( \text{coordinator\_planner} \) followed by the \( \text{behavior\_planner} \) for each individual behavior. Based on the current state of \( \text{COORDINATOR\_DIRECTIVES} \) and all \( \text{BEHAVIOR\_DIRECTIVES} \), the planners generate a \( \text{COORDINATOR\_PLAN} \), followed by a \( \text{BEHAVIOR\_PLAN} \).
**Operation execution:** This is achieved by executing the `action_executor`, which executes the first action in the `BEHAVIOR_PLAN` and generates the set-points for the lower level controllers in the `CONTROL_DATA`.

The response controller goes through the phases of situation evaluation, operation planning, and operation execution to make its control decision.

## 5 Discrete event system model of response controller

The response controller is a discrete event system since it has discrete states which change in response to discrete events (the function calls). We model the response controller by a collection of interacting automata, which is described in detail in the appendix. The derivation of the formal discrete event model of the response controller aides the analysis and verification in several ways:

1. The models can be used for the *formal verification* of the controller to see whether or not it meets the mission specification. For this, the mission specification is also represented as a discrete event system that defines the sequences of behavior modes that the controlled system should execute in order to accomplish the mission task. The control design is verified by showing that the sequences that are allowed by the design automata are also allowed by the specification automata. The formal verification problem is thus one of *language containment* [11], and can be performed using standard tools such as COSPAN [15].

2. The models can be used for *automated code generation*. Once the control design has been formally verified, standard tools such as DIADEM [8] can be used to automatically generate the software code that implements the behavior that is modeled by the interacting automata.

3. Availability of automata models allows easy *redesign* and *maintenance*. Any design change due to either changes in the system or in the mission specification can be more easily incorporated if the formal model of the controller is available. Once the changes have been incorporated the design can be easily verified and then converted into executable software code. Availability of the model thus also facilitates *software reuse*.

Each automaton in the discrete event system model of the response controller tracks the evolution of one particular state variable of the response controller, i.e., there is one automaton per state variable of the response controller. The response controller has the following state variables:

- `COORDINATOR_DIRECTIVE`
- `BEHAVIOR1_DIRECTIVE`, for each $i$ in the set of behaviors
Thus if there are \( n \) behaviors, then the discrete event system model of the response controller consists of a collection of \( n + 3 \) interacting automata. In practice, however, in order to simplify the complexity of any individual automaton it may be convenient to represent it by a collection of interacting automata. So in practice, the response controller model may consist of more than \( n + 3 \) automata.

Since there are no directives or plans initially in the response controller, the initial state of each of the above automata is the empty set.

The event set of an automaton is the set of those controller functions whose range space contains the state variable the automaton is tracking. The following table gives the event set of each automaton listed above:

<table>
<thead>
<tr>
<th>Automaton</th>
<th>Event set</th>
</tr>
</thead>
<tbody>
<tr>
<td>COORDINATOR_DIRECTIVE</td>
<td>coordinator_enabler, behavior_evaluator, behavior_i_enabler</td>
</tr>
<tr>
<td>BEHAVIORi_DIRECTIVE</td>
<td>coordinator_planner, action_evaluator</td>
</tr>
<tr>
<td>COORDINATOR_PLAN</td>
<td>coordinator_planner, behavior_evaluator</td>
</tr>
<tr>
<td>BEHAVIOR_PLAN</td>
<td>action_evaluator, action_executor, behavior_i_planner</td>
</tr>
</tbody>
</table>

This interacting automata based DES model of the response controller consists of two shared variables:

- **STIMULUS_DATA**, and
- **CONTROL_DATA**

The value of the shared variable **STIMULUS_DATA** is modified by the perceptor, whereas the value of the shared variable **CONTROL_DATA** is modified by the response controller.

The guard condition of a certain state transition that occurs in response to a certain event is a predicate over the set of those shared and state variables that are in the domain space of the response controller function to which the event corresponds. The following table lists the “guard variables” (the set of variables over which a predicate of the guard condition is defined) for each event:

<table>
<thead>
<tr>
<th>Event</th>
<th>Guard variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordinator_enabler</td>
<td>STIMULUS_DATA</td>
</tr>
<tr>
<td>behavior_i_enabler</td>
<td>STIMULUS_DATA</td>
</tr>
<tr>
<td>coordinator_planner</td>
<td>COORDINATOR_DIRECTIVE</td>
</tr>
<tr>
<td>behavior_planner</td>
<td>BEHAVIORi_DIRECTIVE, COORDINATOR_PLAN</td>
</tr>
<tr>
<td>action_executor</td>
<td>STIMULUS_DATA, BEHAVIOR_PLAN</td>
</tr>
<tr>
<td>action_evaluator</td>
<td>CONTROL_DATA, STIMULUS_DATA, BEHAVIOR_PLAN</td>
</tr>
<tr>
<td>behavior_evaluator</td>
<td>BEHAVIOR_PLAN, COORDINATOR_PLAN, BEHAVIORi_DIRECTIVE</td>
</tr>
</tbody>
</table>
Finally, the state transitions that occur in response to the event `action_execute` have assignment functions associated with them and modify the shared variable `CONTROL_DATA`.

Figure 7(a), Figure 7(b), Figure 7(c), and Figure 7(d), depict the possible transition types in the automaton tracking the evolution of the state variables, `COORDINATOR_DIRECTIVE`, `BEHAVIOR_PLAN`, `BEHAVIORi_DIRECTIVE`, and `COORDINATOR_PLAN`, respectively.

![Diagram of transition types](image)

Figure 7: Types of transitions in DES model of response controller

### 6 Application to SAMON

As described in Section 3, SAMON is a sampling mobile network of autonomous underwater vehicles. Their intelligent controllers are designed in the intelligent control architecture presented here. In the present design each vehicle is endowed with six different behaviors that are deemed necessary for accomplishing the mission goal of sampling a certain region of ocean floor, which are:

**S-COM**: S-COM stands for “sonar communication”. This behavior is exhibited by an SAUV to communicate with AUVs and FSPs, and also by an AUV to communication with SAUVs and FSPs. Since these communications occur in the water, they are sonar based. There are three modes in which S-COM can be configured:

**Send-query**: This is exhibited to send a query by an SAUV to AUVs and by an AUV to FSPs to establish contact.
**Send-orders:** This is exhibited by an SAUV to issue orders to AUVs or by a AUV to issue orders to FSPs for task assignments.

**Send-reports:** This is exhibited by an AUV or a FSP to report acknowledgments to queries, or to report compliance to orders, or to report gathered samples.

**R-COM:** R-COM stands for “radio communication”. This behavior is executed by an SAUV only to communicate with the TC. Since this communication occurs in the air, it is radio based. There are two modes in which R-COM can be configured:

- **Send-report1:** This is exhibited by an SAUV to provide status report to the TC.
- **Send-report2:** This is exhibited by an SAUV to report gathered samples to the TC.

**RECON:** RECON stands for “reconnaissance”. This behavior is exhibited by a vehicle to survey a certain designated region of ocean floor. This operation is the only mode of this behavior.

**GOTO:** As the name suggests, this behavior is exhibited by a vehicle to move to a certain designated point. This operation is the only mode of this behavior.

**REFUEL:** As the name suggests, this behavior is exhibited by a vehicle to visit a refueling dock and refuel. This operation is the only mode of this behavior.

**LOITER:** As the name suggests, this behavior is exhibited by a vehicle to remain idle maintaining its current position, which is the only mode of this behavior.

Since each vehicle is endowed with six behaviors the response controller for each vehicle in the SAMON application consists of six behavior controllers, and a single coordinator. The coordinator and the behavior controllers, through their enablers, monitor the **STIMULUS_DATA** and generate **COORDINATOR_DIRECTIVES** to enable an appropriate behavior mode. The automata representing the evaluation of the **directives** and the **plans** in response to various events (function calls) is depicted in Figures 8-15. For convenience, the set of directives is partitioned into the **long-term directives**, the **short-term directives**, the **count directives**, and the **points directives**.

The **STIMULUS_DATA** is one of the shared variables, and for the SAMON application consists of the following:

**ORDERS:** For each vehicle this consists of orders received from the next higher level, namely, the TC level for the SAUVs, and the SAUV level for the AUVs. It is monitored by the vehicle’s **coordinator_enabler**. The list of orders in the SAMON application consists of:

- Resume
- Search
- Goto-point
- Dump-samples

The variable NEW-ORDERS is set true when a new order is received.

**QUERIES:** For each AUV, this consists of the list of SAUVs from whom it has received queries for establishing contacts. It is monitored by the AUV’s S-COM_enabler. The variable NEW-QUERY is set true when a new query is received.

**CONTACTS:** For each SAUV (resp., AUV), this consists of the list of AUVs (resp., FSPs) with whom it has established contact. It is monitored by the vehicle’s S-COM_enabler. The variable AUV-CONTACT (resp., STATION-CONTACT) is set true when SAUV’s (resp., AUV’s) contacts list is nonempty, and the variable NEW-AUV-CONTACTS (resp., NEW-STATION-CONTACT) is set true when the SAUV (resp., AUV) establishes a contact with a new AUV (resp., FSP).

**SAMPLES:** For each vehicle this consists of the list of sample data that it has gathered.

**AUTHORITY:** For each vehicle this is assigned one of the two values—SAUV or AUV—depending on whether the vehicle is to take the role of a SAUV or a AUV. Thus by changing this variable, the role of the vehicle can be changed dynamically.

**SYS-TIME:** This is the value used for setting the time when the system timer times out.

**FUEL:** This is the value that determines the minimum fuel level that a vehicle is allowed to have before it must refuel.

**DH:** This is the value that determines the maximum amount of drift in the position error is allowed.

By monitoring the STIMULUS_DATA consisting of orders, queries, contacts, samples, authority, sys-time, fuel, and dh, the various enablers generate the appropriate coordinator directives. By using these directives, the coordinator planner generates a coordinator_plan, and also configures each enabled behavior in its appropriate mode by generating the corresponding behavior_directives. The coordinator_plan in this case is a sequence of enabled behaviors belonging to the set S-COM, R-COM, RECON, GOTO, REFUEL, and LOITER. The planner for the first behavior in the coordinator_plan, generates a behavior_plan using its behavior_directives. The first action in this plan is then executed by the action_executor, generating the appropriate CONTROL_DATA, which provides the set-points for the lower level conventional controllers. In the SAMON application, the lower level conventional control exists for three subsystems, namely, the navigation subsystem, the sonar communication subsystem, and the radio communication subsystem.

In the SAMON application, the following primitive actions exist, which when executed in a particular sequence, result in the execution of a certain behavior mode:
**s-command:** This is used to send query/order by sonar.

**s-response:** This is used to send status reports/samples by sonar.

**r-query:** This used to send query by radio.

**r-dump:** This used to send samples by radio.

**transit:** This is used to move to a designated position.

**hang-close:** This is used to maintain the present position.

The following table lists the different behavior modes, and the corresponding sequence of primitive actions that executes the behavior mode.

<table>
<thead>
<tr>
<th>Behavior mode</th>
<th>Action sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-COM [send-query]</td>
<td>&lt;s-command(query),s-command(query)&gt;</td>
</tr>
<tr>
<td>S-COM [send-order]</td>
<td>&lt;s-command(order)&gt;</td>
</tr>
<tr>
<td>S-COM [send-report]</td>
<td>&lt;s-response(report)&gt;</td>
</tr>
<tr>
<td>R-COM [send-report1]</td>
<td>&lt;transit(top),r-query&gt;</td>
</tr>
<tr>
<td>R-COM [send-report2]</td>
<td>&lt;transit(top),r-dump&gt;</td>
</tr>
<tr>
<td>RECON</td>
<td>&lt;transit(point),s-command(order)*&gt;</td>
</tr>
<tr>
<td>GOTO</td>
<td>&lt;transit(point)&gt;</td>
</tr>
<tr>
<td>REFUEL</td>
<td>&lt;transit(dock)&gt;</td>
</tr>
<tr>
<td>LOITER</td>
<td>&lt;hang-close&gt;</td>
</tr>
</tbody>
</table>

Figures 8-15 depict the automata equivalents tracking the various state variables of the intelligent controller for the underwater vehicles. The state transitions in these automata take place when events (function calls) indicated in oval boxes occur. These events occur in the order depicted in 6, thus dictating the order of state transitions.

In Figure 8, for example, the state transitions of the long-term coordinator directive are depicted. When the execution of behavior_evaluator occurs, the state variable evolves as depicted in the top portion of the figure, while the middle (resp., bottom) portion depicts the evolution when the coordinator_enabler (resp., RECON_enabler) is executed. Each transition edge has a pair of labels: The label above the transition edge gives the guard condition (a predicate over the shared variables and the set of state variables) under which the transition is executed. The label below the transition edge gives the resulting modification in the state variable. The transition edge of the top most branch in Figure 8, for example, is executed when the event behavior_evaluator occurs, the COORDINATOR_PLAN contains REFUEL as the first behavior (indicated as the guard CP = <REFUEL,...>), and the vehicle is an SAUV (indicated as the guard AUTHORITY = SAUV); and as a result, a state transition that pushes LOCATE-AUVS and AT-DOCK to the long-term coordinator_directives stack takes place. All state transitions are self-descriptive, and can be interpreted easily.
Figure 8: Coordinator long-term directives
Figure 9: Coordinator short-term directives
Figure 10: S-COM long-term directives
Figure 11: S-COM short-term directives
Figure 12: RECON/GOTO long/short-term directives
Figure 13: Points and Count directives
Figure 14: Coordinator plan
Figure 15: Behavior plan
7 Models for specification and its verification

The mission task, i.e., the control specification, that the intelligent controller should achieve is also represented by automata models. The specification automata represents the sequences of behavior modes that the controlled system must execute in order to accomplish the mission task. As an example, Figure 16(a) shows the specification automaton for an SAUV, and Figure 16(b) shows that for an AUV.

![SAUV and AUV specification automata](image)

Figure 16: SAUV and AUV specification automata

The control design is verified by showing that the sequences that are allowed by the design automata are also allowed by the specification automata. The verification problem is thus one of language containment [11] and can be performed using standard tools such as COSPAN [15].

8 Conclusion

We have presented a new intelligent control architecture for the design of intelligent controllers. Intelligent control is exercised when the conventional control is inadequate due to
the (i) lack of accurate plant model, and/or (ii) presence of control requirements that fall beyond the scope of the conventional control. Our approach to the design of intelligent controllers is a behavioral approach, where a behavior is a certain high level activity that determines the manner in which a system must react to external conditions to execute sub-tasks of the mission tasks. This behavioral approach naturally yields a cascaded architecture consisting of a perceptor subsystem and a response controller subsystem. The task of the perceptor is to extract those features from sensor signals that enable the execution of one of the behaviors, and the task of the response controller is to monitor such features and to execute one of the enabled behaviors. The perceptor and the response controller are both hierarchical, allowing complexity management for systems that must operated in complex environments and carry our complex tasks.

The behavior based intelligent control architecture presented in this paper is motivated from the functional model of brain, and hence a controller designed in this architecture will be an “intelligent controller”. This architecture has been successfully applied to design intelligent controllers in practice, and one such application is discussed in this paper. The other applications include ship damage control automation, medical diagnosis, and defense application. Further work is needed to make this architecture a more realistic function model of brain, and to enhance it further so as to allow it to perform tasks such as learning. The controllers designed in the current architecture are able to adapt to the changing environmental conditions in a manner they are programmed to do so; however, they are unable to devise any new control scheme which a system that has the capability to learn would be able to do.

References


A Discrete event systems: preliminaries

A discrete event system (DES) is an event-driven system that has discrete states, which change in response to asynchronous occurrences of certain discrete activities, called events. Examples of DESs include manufacturing systems, communication networks, reactive computer programs, database management systems, automated traffic systems. For a detailed introduction to DESs, readers are referred to [7, 12].

A DES can be modeled by a finite collection of interacting automata. Each automaton has a finite set of states and events, and share a finite set of variables. An automaton transitions from one state to another in response to the execution of an event provided a certain guard condition is satisfied. A guard condition is a predicate over the states of other automata and the values of certain shared variables. A state transition may modify the values of certain shared variables.

Formally, let \( \{G_i\} \) be a finite collection of automata indexed by \( i \), with \( S \) denoting its finite set of shared variables. Each variable \( s \in S \) takes values over a countable domain \( D(s) \). We use \( D(S) := \prod_{s \in S} D(s) \) to denote the domain of the set of shared variables. Each automata is a quadruple,

\[
G_i := (X_i, \Sigma_i, E_i, x^0_i),
\]

where \( X_i \) is its finite set of states, \( \Sigma_i \) is its finite set of events, \( E_i \) is its finite set of state transitions, and \( x^0_i \in X_i \) is its initial state. Each transition \( e \in E_i \) is quintuple of the form:

\[
e := (x^e, \sigma^e, \mathcal{P}^e(S \times \Pi_i X_i), f^e, y^e),
\]

where \( x^e \in X_i \) is the state where the transition is executed, \( \sigma^e \in \Sigma_i \) is the event that causes the state transition, \( y^e \in X_i \) is the state resulting from the execution of the transition, \( \mathcal{P}^e(S \times \Pi_i X_i) \) is the guard condition—a predicate over the shared variables and the states of the other automata—which must be satisfied for the transition to be executed, and \( f^e : D(S) \to D(S) \) is the shared variable assignment function that assigns new values to the shared variables. A transition \( e \in E_i \) is equivalently also represented as:

\[
e := x^e \xrightarrow{\sigma^e, \mathcal{P}^e(S \times \Pi_i X_i)} S := f^e(S) \to y^e.
\]
Example 1 Consider for example the following simple manufacturing system consisting of two machines and a buffer shown in Figure 17.

In this manufacturing system, the first machine $M_1$ fetches a part from an infinite supply when it is in its idle state. The arrival of a part into $M_1$ is denoted by the event $a_1$, and causes $M_1$ to change its state from idle to working. When $M_1$ finishes processing, the part is deposited in buffer $B$ (event $d_1$). This event causes $M_1$ to change its state from working to idle. The second machine $M_2$ fetches a part from the buffer when it is in its idle state and the buffer is nonempty. This is denoted by the event $a_2$, whose occurrence causes the machine $M_2$ to change its state from idle to working. After $M_2$ finishes processing a part, the part departs from the manufacturing system, denoted by the event $d_2$, and causes $M_2$ to return to its idle state.

We model this manufacturing system as a pair of interacting automata $G_1$ and $G_2$, where for $i = 1, 2$, the automaton $G_i$ models the machine $M_i$. Each automaton has two states, $X_i := \{\text{idle}_i, \text{working}_i\}$, with idle$_i$ being the initial state of $G_i$. The event set of the automaton $G_i$ consists of $\Sigma_i := \{a_i, d_i\}$. The number of parts in the buffer $B$, denoted by $b$, is the shared variable. The set of transitions of the two automata are shown in Figure 18.

$G_1$ starts in the initial state idle$_1$, and when event $a_1$ occurs it transitions to the state working$_1$. It returns to the initial state when event $d_1$ occurs, also incrementing the value of the shared variable, the buffer content, by 1. $G_2$ starts in the initial state idle$_2$, and when event $a_2$ occurs it transitions to the state working$_2$ provided the value of the shared variable exceeds zero, also decrementing the value of the shared variable by 1. $G_2$ returns to its initial state when event $d_2$ occurs.
A collection of interacting automata \( \{G_i := (X_i, \Sigma_i, E_i, x_i^0)\} \) that models a DES can be combined using \textit{synchronous composition} to form a single automaton that is an equivalent model of the DES. Without loss of generality, we define the synchronous composition of two interacting automata \( \{G_i := (X_i, \Sigma_i, E_i, x_i^0)\}_{i=1,2} \) which share variables in a set \( S \). Their synchronous composition, denoted \( G_1 \parallel G_2 \), is the automaton \( G_1 \parallel G_2 := (X, \Sigma, E, x^0) \), where \( X := X_1 \times X_2 \), \( \Sigma := \Sigma_1 \cup \Sigma_2 \), \( x^0 := (x_1^0, x_2^0) \), and the set of transitions \( E = E_\alpha \cup E_\beta \cup E_\gamma \):

\[
E_\alpha := \{ ((x_1^e, x_2^e), \sigma^e, \mathcal{P}^e, f^e, (y_1^e, y_2^e)) \mid 
\exists (x_1^f, \sigma^f, \mathcal{P}^f_1, f^f, y_1^f) \in E_1, (x_2^f, \sigma^f, \mathcal{P}^f_2, f^f, y_2^f) \in E_2 \text{ s.t. } 
\mathcal{P}^e_1 \land \mathcal{P}^e_2 = \mathcal{P}^e \}
\]

\[
E_\beta := \{ ((x_1^e, x_2^e), \sigma^e, \mathcal{P}^e, f^e, (y_1^e, x_2^e)) \mid 
\exists (x_1^f, \sigma^f, \mathcal{P}^f_1, f^f, y_1^f) \in E_1 \text{ s.t. } \sigma^e \in \Sigma_1 - \Sigma_2 \}
\]

\[
E_\gamma := \{ ((x_1^e, x_2^e), \sigma^e, \mathcal{P}^e, f^e, (x_1^e, y_2^e)) \mid 
\exists (x_2^f, \sigma^f, \mathcal{P}^f_2, f^f, y_2^f) \in E_2 \text{ s.t. } \sigma^e \in \Sigma_2 - \Sigma_1 \}
\]

\( E_\alpha \) is the set of transitions which occur synchronously with the participation of both \( G_1 \) and \( G_2 \), whereas \( E_\beta \) and \( E_\gamma \), respectively, are the set of transitions that occur asynchronously with the participation of \( G_1 \) and \( G_2 \) only.

**Example 2** The synchronous composition of automata \( G_1 \) and \( G_2 \) described in Example 1 is shown in Figure 18.