An Abstract Behavior Representation for Robust, Dynamic Sequencing in a Hybrid Architecture

Jeffrey P. Duffy and **Gilbert L. Peterson**
Air Force Institute of Technology*
2950 Hobson Way
WPAFB OH 45433-7765
{jeffrey.duffy , gilbert.peterson}@afit.edu

Abstract

Hybrid robot control architectures try to provide reactive and deliberative functionality by decomposing a complex system into layers of increasing abstraction and temporal complexity. In building complex systems that react quickly to dynamic environments, the division of components are generally characterized as a Deliberator, a Sequencer, and a Controller. This paper presents a descriptive representation of reactive robot behaviors and the environment that the behaviors anticipate and affect. This paper also presents a control algorithm that uses this representation to create a robust link between the Sequencer and Controller in a hybrid reactive control architecture. The behavior representation promotes robustness and modularity as being a semantic suggestion rather than a syntactical burden like that of the task-level control languages currently in use. Thus, it allows for reduced development overhead and duplication of work for system modifications.

Introduction

With the advent of hybrid robot control architectures, which generally try to provide reactive and deliberative functionality, many systems have separated plans, coordination, and actions into separate processing layers. This approach promotes more complex systems that perform well in dynamic and goal-oriented environments. In various architectures however, the connections between these layers are typically hardcoded so changes within one layer require modifications in other layers. By creating robust connections at each layer, the software becomes more maintainable and updates within layers cause minimal updating of others. The majority of hybrid architectures link planning and execution through the use of task-level control languages (Simmons & Apfelbaum 1998). These languages require that each behavior (i.e. task-tree, task net, . . .) is expressed explicitly by the syntax of the language. This limits the implementation to the constructs of the language. Our proposed representation provides a way to describe a behavior for sequencing, but the actual implementation is left to the creativity of the behavior architect.

This paper presents a descriptive representation of reactive robot behaviors and the environment and objectives the behaviors anticipate and accomplish. This paper also presents a control algorithm that uses this representation to create a robust link between the Sequencer and the Controller in a hybrid reactive control architecture. The control algorithm uses the representations to search and select appropriate behaviors to activate and deactivate for completing system objectives. The Sequencer implements the control algorithm and seamlessly links to the Controller. Using the control algorithm and representations, behavior addition and functional system changes reduce Sequencer coding to a description of the behaviors and minimal changes to the state for associated state variables.

The remainder of this paper presents a background discussion on mobile robot architectures. This is followed by related work in behavior representation and plan execution. Next, is our proposed representations for behaviors. Then, we present the control logic that utilizes the proposed representations to link the Sequencer to the Controller. Finally, a brief conclusion and details of future work.

Three-Layer Reactive Control Architectures

The idea behind three-layer architectures (TLAs) is to merge deliberative planning and reasoning with a reactive control unit to accomplish complex, goal-directed tasks while quickly responding to dynamic environments. These architectures typically have three main components (or layers): a reactive feedback control mechanism (Controller), a slow deliberative planner (Deliberator), and a sequencing mechanism that connects the first two components (Sequencer) (Gat 1998). Figure 1 shows the layout of a TLA and how the layers interact with the state, robot controls, the environment (sensors), and other layers. As shown, the architecture is hierarchical and state-based. The state receives sensor data updates and makes the data available to the whole system.

Controller

Layers within hybrid architectures are typically separated by abstraction and temporal complexity. Therefore, the Con-

*This research was sponsored by an AFRL Lab Task 06SN02COR from the Air Force Office of Scientific Research, Lt Col Scott Wells, program manager. The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or the U.S. Government.

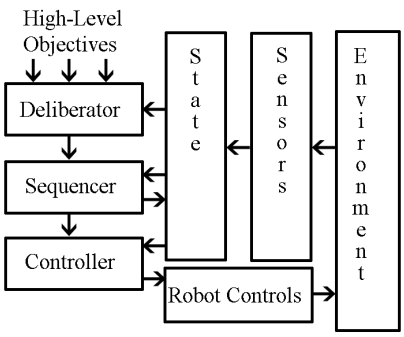

Figure 1: Architectural Layout of a TLA

troller is responsible for the low-level, reactive functionality that is accomplished in real-time without knowledge of high-level goals. A software package called the Unified Behavior Framework (UBF) satisfies these criteria (Woolley 2007) and is ideal for the defining line of the Controller.

The UBF is a reactive controller that abstracts the specific implementation details of specialized behaviors and permits behavior reconfiguration during execution (Woolley & Peterson 2007). Traditionally, a mobile robot design implements a single behavior architecture, thus binding its performance to the strengths and weaknesses of that architecture. By using the UBF as the Controller, a common interface is shared by all behaviors, leaving the higher-order planning and sequencing elements free to interchange behaviors during execution to achieve high-level goals and plans. By establishing behavior architectures in the context of the UBF, one can dynamically interchange between architectures capitalizing on the strengths of popular reactive-control architectures, such as the Subsumption (Brooks 1985) and Utility Fusion (Rosenblatt 1998) architectures. Thus, exploiting the tight coupling of sensors to actions that reactive-control architectures achieve.

Sequencer

The Sequencer transforms the plans from the Deliberator to the actions of the Controller and maintains an abstract state accordingly for Deliberator replanning. Its job is to select the behaviors that the controller uses to accomplish the objectives set forth by the Deliberator (Gat 1998). This requires that the Sequencer set parameters for the behaviors and changes the active behaviors at strategic times to meet objectives. To do this, the Sequencer must monitor and update the state as appropriate. As seen in Figure 1, aside from sensors updating their data in the state, the Sequencer also sends information into the state. This allows for the setting of parameters and state variables that behaviors and other layers use. Our vision for the Sequencer is of a robust software module that, after initial implementation, requires minimal maintenance and modifications for system changes. We believe that this is accomplished by using the behavior representation and control logic proposed in this paper. Like the behavior control logic in the UBF, the Sequencer uses the behavior representation as an abstract interface for sequencing the behaviors without any knowledge of the concrete implementation. The transition from the Sequencer to the Controller is the passing of a composite behavior module that represents an arbitrated hierarchy of behaviors that accomplish high-level objectives.

Related Work

Hybrid architectures differ in where the separation between the layers occur and how the layers are connected. It is ideal to integrate components into complete robot control systems (Ghallab *et al.* 2003). Thus, integrating the components of hybrid systems requires layers and associated connections within the system to be modular and robust. Wherever the connection line is drawn however, it is typical that the connection mechanism is not explicitly documented. Most notable systems are not modular when it comes to functional decomposition of the Sequencer and Controller. This is due to the trend of continuous interleaving of planning and execution (Ghallab *et al.* 2003). Interleaving allows for more efficient transitions from planning to control but also makes this mechanism more complex and coupled. This places some Sequencer and Controller elements at the same level with unclear connections. These systems generally use specialized languages for representing actions to perform planning and execution simultaneously.

Many autonomous robot systems employ sophisticated plan execution systems that continuously interleave planning and execution (Ghallab *et al.* 2003), also referred to as task-level control languages (Simmons & Apfelbaum 1998). Some languages have no clear division between the Sequencer and Controller because they interchangeably perform the functions of both components. For example, the 3T architecture's Sequencer is a RAP interpreter that decomposes RAPs until they define a *set of skills* that, when activated by the Controller, accomplish their particular task (Bonasso *et al.* 1997). Languages like RAP (Firby 1989) and RPL (McDermott 1991), which is a direct descendant of RAPs, are meant to be implemented as the Controller using a Lisp-based interpreter. However, these languages require knowledge of the goals that they must accomplish and require more planning by the implementer. A robust Controller, on the other hand, utilizes low-level behaviors that perform specialized tasks without regard to any of the high-level objectives that it is accomplishing, as is the case of the UBF (Woolley & Peterson 2007). Other plan execution systems and architectures that use them include: The PRS language (Ingrand *et al.* 1996) is used within the Sequencer of the LAAS architecture (Alami *et al.* 1998). The Execution Support Language (ESL) is designed to be the implemention substrate for the Sequencer of hybrid architectures, such as 3T (Gat 1997). Although the Task Control Architecture (TCA) (Simmons 1994) did not use a control language, it was a major influence for the Task Development Language (TDL) (Simmons & Apfelbaum 1998), which is used for most of the CLARAty architecture's (Volpe *et al.* 2001) executive functionality, which is analogous to the Sequencer (Estlin *et al.* 2001).

These languages all have their own specific syntax and

control semantics. They require the programmer know how the system functions as a whole, which limits the programmer to the constructs of the language. Although most languages utilize abstract representations to generate or select appropriate actions, the linking from action representation to actual control mechanism is not documented or appears to be hardcoded. Instead of a language that spans from deliberative planning to action execution, we propose a behavior representation and control algorithm that begins with a goal-set (or objectives plan) and dynamically sequences a library of behaviors to accomplish the complex tasks of the goal-set in a dynamic environment.

Example Domain

Consider a mobile office janitor robot tasked with discovering trash and placing it into its appropriate bin. This system is supplied with sonar and laser range data, a pan-tilt camera with image recognition capabilities, a gripper, and a navigation system. However, its audio output has been removed. The robot receives high-level tasks from an outside source that describes objects that are considered trash or changes the known location of the trash bin. The robot is programmed with the following library of behaviors (`Name`; *Required Sensors*; Function).

- `greeting`; *Camera, Audio*; Audibly greet employees
- `avoid-obstacle`; *Laser, Map*; Avoid obstacle in an optimal path to a target location, if given
- `wall-follow`; *Sonar, Map*; Follow walls and ensure all areas are searched
- `scan-for-trash`; *Camera*; Scan for trash
- `get-object`; *Grippers, Camera*; Pick up target object
- `release-object`; *Grippers*; Release held object
- `deliver-object`; *Odometry*; Go-to target location

Proposed Representations

All behaviors are created to accomplish a specific task or goal. By creating a standardized way of abstractly describing the characteristics of a behavior, one can create a mechanism that continuously searches and selects appropriate behavior activations and deactivations for accomplishing desired system objectives. This allows for a robust implementation of a Sequencer that handles abstract behavior descriptions in a uniform manner.

Behaviors (B)

Keeping with the general rule that reactive behaviors tightly couple sensing to action, a behavior's actions are described by what it senses and how it affects the environment. A behavior is informally defined as the set of motor commands that trigger in response to sensor readings for accomplishing a programmed task. For example, if a set of sensors indicates that there is an obstacle impeding forward motion, the `avoid-obstacle` behavior applies appropriate motor settings to steer the robot away from the obstacle without collision. Behaviors are as complex or simple as the architect determines appropriate.

A simple behavior has just one set of outputs that is triggered by just one set of input conditions. Whereas, a complex behavior has multiple output sets that are in response to multiple input sets. We refer to these input-to-output paths as *activation-paths*. Therefore, each *activation-path* must include the initial conditions that it assumes to be true before execution and the output conditions that it creates.

However, the Sequencer requires more information due to concurrent behavior execution. Each *activation-path* represents a behavior function and is represented as the tuple:

$$A = \{D, G, I, O, C, v\}$$

where D is list of sensor, or computed, data required for the behavior to function properly. G is the set of high-level abstract goals that the *activation-path* accomplishes. I and O are the set of input and output state conditions. C is the set of system controls that the behavior modifies. Finally, v is the vote that the behavior generates when it delivers an action recommendation for this *activation-path*. Each of these components is presented in more detail.

Required Data (D) Since a behavior is a tight coupling of sensor readings to motor commands, D represents the set of sensor data d required for the behavior to function properly. This data includes computed data that is directly related to the environment, but not directly from a sensor. For example, a basic `avoid-obstacle` behavior implementation may require just sonar data so that it can arbitrarily move away from an object. Whereas, an advanced `avoid-obstacle` behavior may require laser and map data to avoid the obstacle and remain on track to a specific target location. The simple implementation has a set D with one element (d_{sonar}), and the complex implementation has a set D with two elements (d_{laser}, d_{map}).

Abstract Goals (G) When behaviors are programmed, they are programmed to perform a specific function or accomplish a specific task. These high-level views of what the behavior accomplishes are represented in G. Using the example above, the basic `avoid-obstacle` behavior has one item in G (*avoid-obstacle*), and the advanced implementation also has one item in G but it is a different abstraction (*avoid-obstacle-target*). These values are used to give a high-level representation without moving to the decomposition level of the output conditions.

Initial Conditions (I) The initial conditions of a behavior (I) represent the set of environment variables that, when true, generates an action recommendation and vote for the behavior's activation. I also represents the conditions required for the *activation-path* to produce the action outputs in (O), the affects to the controls of (C) and the vote v for accomplishing the abstract goals in G. For example, the `avoid-obstacle` behavior does not vote to control movement until it reads that there is an object within its projected path. Thus, its set I consists of one element (*obstacle*). The set I is an abstract representation of the initial conditions and does not dictate the implementation of detecting an *obstacle*. Therefore, this representation accepts any abstraction of information and places the burden of identifying

these abstract conditions on the behavior's programmer.

The complexity of a behavior is dictated by the number of *activation-paths* it contains. Each *activation-path* is required to contain a different set I. Therefore, a separate *activation-path* is required for each set of initial conditions that cause the behavior to generate a different output/vote. This representation allows for arbitrated behavior hierarchies to be described as a composite behavior with multiple functionalities dependent upon different initial conditions.

Priority of *activation-path* selection is based first on comparative conditions and then on set order. Consider the janitor robot's `avoid-obstacle` behavior B with two *activation-paths* (B = $\{A_1, A_2\}$). These *activation-paths* are activated when there is an *obstacle* or there is an *obstacle* and a *target-location* established (I_{A_1}=$\{$*obstacle*$\}$ and I_{A_2}=$\{$*obstacle*, *target-location*$\}$). If the two condition sets are met, then they are first compared. Since I_{A_1} is a subset of I_{A_2} and I_{A_2} is more specific, then I_{A_2} is chosen. Conversely, if one was not a subset of the other, then the choice is made by its order in set B and A_1 is chosen over A_2.

A pictorial representation of the `avoid-obstacle` behavior with multiple *activation-paths* is shown in Figure 2. This illustrates that, depending upon the initial conditions, the behavior has different functionality. The behavior votes differently when there is an object in the path, as opposed to, when there is an object in the path and a target travel location has been established.

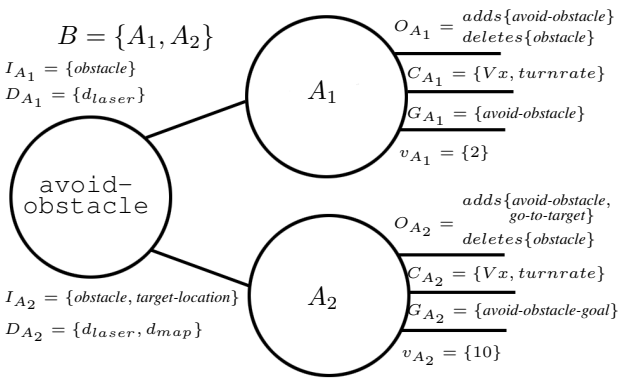

Figure 2: Behavior *Activation-Path* for `avoid-obstacle`

Postconditions (O) The postconditions O represent the set of environment effects that the behavior intends to achieve. This intent is based on action recommendations for the behavior at the given initial state I. It can be viewed as the behavior's function given adequate to deal with uncertain effects. The effects on the environment can add components or remove components. If the behavior is a basic `avoid-obstacle` behavior, then the postcondition adds *avoid-obstacle* but removes *obstacle*. However, if it is a more advanced `avoid-obstacle` behavior that finds the optimal avoidance path to reach a desired goal location, then the postconditions add *avoid-obstacle* and *go-to-target* but removes *obstacle*. For the latter example, Figure 2 shows the postconditions that are associated with each initial condition set I_{A_x} in A_x. With each of the behavior's *activation-paths*, there must be an associate postconditions set that corresponds to an initial conditions set. Therefore, for each A_x in A, there is a corresponding postcondition set O_{A_x} that corresponds to I_{A_x}.

Control Settings (C) Behaviors are written to affect settings for specialized controls. Most commonly, these are motor controls. A behavior potentially affects the control settings of single or multiple controls. Dependent upon which controls are set, the control loop determines the most appropriate arbiter for use with a set of behaviors. The controls that the behavior affects is denoted by the set C. This set, like postconditions O, requires values at each *activation-path* dictated by the set I. Figure 2 shows the control settings sets for the `avoid-obstacle` behavior.

Votes (v) The value v in an *activation-path* represents the vote for that execution branch. Since each A_x generates an action recommendation, there must be a corresponding vote v_{A_X} for each A_x (Figure 2). These vote values are used to determine the output of different arbitration techniques, which are more thoroughly discussed in a later section.

Control Logic

This section discusses using the formalized description of behaviors to dynamically select appropriate behaviors and arbiter hierarchies for accomplishing desired objectives in sequencing. The Sequencer searches through the library of behaviors and generates an action hierarchy package, which is behavior activations/deactivations that will accomplish the objectives set forth by the Deliberator. Some behaviors may require activation, then deactivation and additional activations later in time. This complicates the search space since it allows cyclic branches. To make the automated link between the Sequencer and the Controller, we create a control algorithm in the Sequencer that generates a behavior hierarchy. The control loop begins by receiving, from the Deliberator, a goal-set (or objective plan) that describes the desired functionality of the system. Informally, the control logic algorithm performs the following control loop:

1. Receive objective plan (*OP*) from Deliberator
2. Load behaviors requiring data the robot can provide
3. Create a partial plan from available behaviors that accomplish the desired objectives
4. Generate a solution to the partial plan
5. Determine arbitration that accomplishes objectives and satisfies the solution plan
6. Generate behavior hierarchy and send to controller
7. Monitor progress, hardware changes, and new *OP*s

Receive Objective Plan From Deliberator

The objective plan (*OP*) that the Sequencer receives from the Deliberator contains a list of goals with a sequence number and an activation priority. The sequence number dictates the order in which the goals should be met (e.g. *go-to-target* is met before *release-object*). If two goals have the same sequence number, then they are expected to be accomplished

before the next sequence but in no particular order. However, if there is a cause for competition, then the activation priority dictates precedence. For example, an *avoid-obstacle* goal and *search-area* goal can be accomplished in the same sequence, but when an obstacle is threatening collision, ideally the *avoid-obstacle* goal has a higher activation priority. This priority is identified by the activation priority of the goal in the *OP*. The *OP* for the example domain is shown in Table 1. The sequence of the *OP* has an ascending precedence and activation priority has a descending precedence. For example, *get-object* happens before *release-object* and *avoid-obstacle* has higher priority than *search-area*.

Goal	Sequence	Activation Priority
avoid-obstacle	1	2
search-area	1	1
find-trash	1	1
get-object	2	1
avoid-obstacle-target	3	3
go-to-target	3	2
release-object	3	1

Table 1: Objectives Plan for janitor robot

Available Behaviors

When an objective plan enters the control loop, the control algorithm places the behaviors B that required only available data into a library of viable behaviors L. This process is completed everytime a new objective plan enters the control loop. It is easy to envision a system that conserves energy by deactivating expensive sensors during critical times or deactivating sensors due to failure. Therefore, a check of available data at every loop is ideal. This step identifies behaviors as viable if all *activation-paths*, from initial condition to postcondition, can be accomplished with the available data. From the example domain, the `greeting` behavior is not selected for search since the system does not have an audio output. All other behaviors are viable.

Preprocess Objectives

This step ensures that the objectives from the *OP* can be accomplished with the available behaviors in L. A behavior is selected as a candidate if it contains a desired objective g from the *OP* in a G_{A_x}. By generating a partial plan from the *OP* with these behaviors, the sequence of goal accomplishment is maintained. We then search for additional behaviors that can link the behavior sequences together and solve the partial plan. For simplicity, this example selects all the available behaviors during preprocessing of the *OP* in table 1.

Generate Solution

This step uses the preprocessed partial plan as the starting point in generating a solution. This solution is a plan with a list of behaviors and their associated ordering constraints. Since we started with the partial plan that satisfies the sequential requirements of the *OP*, the solution will satisfy the *OP* as well but may impose more sequential restrictions. The initial state is either the current state or the projected output state of the currently running behavior hierarchy. The end state is the outputs O of the behaviors that accomplish the goals of the last sequence in the *OP*. For our example, the output is the O's for the `deliver-object` and `release-object` behaviors since they satisfy the goals of the last sequence. Since preconditions I and postconditions O do not represent the atomic processes that partial-order-planning expects (Russell & Norvig 2003), a modified planning algorithm is used to incorporate the complexity of the concurrent taskings that each behavior can encounter.

The benefits of using this planning approach allows breaking sequential tasks into subtasks if necessary. A plan is made for the first subtask while the others are processed after subsequent subtasks complete. Each level of *sequencing priority* can be separated as a subtask using a Highest Activation arbiter where the Sequencer monitors for the correct output-input pairs to advance *sequence priority* level. Or, the Sequencer can search for an arbitration combination capable of combining some, or all, of the sequential and activation priorities into one arbitrated behavior hierarchy (Figure 3).

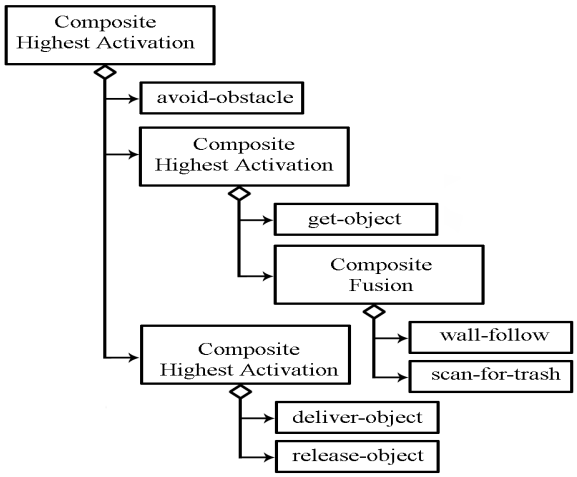

Figure 3: Arbitrated Hierarchy of Behaviors

Arbitration

Arbiter selection is a crucial step. The arbiter ensures proper fusion, priority activation, and ordering of behavior action recommendations. The arbitration selection is mostly based on the controls that the behaviors affect C. The other aspect is how the behaviors vote for each branch and what that branch affects. For example, `scan-for-trash` only controls the pan and tilt of the camera and `wall-follow` controls the forward velocity and turnrate. Thus, a Utility Fusion arbiter is an ideal arbiter choice. Conversely, the `avoid-obstacle` and `deliver-object` behaviors both control the forward velocity and turnrate, but `avoid-obstacle` has a higher activation priority (Table 1). So, a Highest Activation arbiter is best for this combination. An example of a final arbitrated hierarchy of behaviors for our domain is shown in Figure 3.

A second significant component of arbiter choice is the vote weighting of behaviors. A programmer cannot know the behavior's use throughout the life of the system. Therefore, the Sequencer requires the ability to weight the votes of each behavior within an arbiter. Although seemingly simple, this step is the final validation to ensure the behaviors' votes are correct for arbitration to perform as expected. This step traces through the arbitrated hierarchy of behaviors and ensures the weights cause the arbitration to meet the objectives in the desired priority and sequence. Otherwise, the plan is rejected and either a new arbitration search is conducted or the Sequencer triggers a plan failure in the state for the Deliberator to catch.

Conclusion

Currently, 90% of the proposed representation and control logic is implemented within our robot architecture and demonstrated in Stage and on a physical Pioneer robot. We have witnessed dynamic behavior sequencing for high-level plan changes and hardware changes. We have also demonstrated sequential objective plans that activate new behavior hierarchies when previous plans meet their end state. Although not fully implemented, the behavior representation and control logic has shown to provide a robust, dynamic sequencing component to our hybrid robot architecture. The control loop and behavior representation that we propose enables the system to utilize simple and complex behaviors. We have described a behavior representation that enables a dynamic, automated behavior activation system and thus is a robust mechanism for coupling the Sequencer and Controller within a hybrid architecture. The representation is unlike task-control languages in that it is more of a semantic suggestion rather than a syntactical burden. Thus, it allows for a formal method to describe and utilize the behaviors without limiting creative software design. This systematic way of describing behaviors enables the Sequencer to select the appropriate hierarchy for accomplishing desired objectives without requiring *a priori* knowledge of the behaviors' implementation. Thus, it allows for reduced development overhead and duplication of work for system modifications.

For future research, we plan to implement an arbiter representation and automated selection process that satisfies the goals of the objectives plan. Currently, this functionality is hardcoded to select between a Utility Fusion or Highest Activation arbiter. Like the behavior representation, arbiter representation will allow programmers to use an abstract interface to create custom arbiters that are seamlessly interchanged within the control loop. Additionally, our implemented behaviors have just one *activation-path* and we plan to extend our implementation and planning to incorporate behaviors with multiple *activation-paths*.

References

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. F. 1998. An architecture for autonomy. *International Journal of Robotics Research* 17(4):315–337. PT:J.

Bonasso, R. P.; Firby, J.; Gat, E.; David, K.; Miller, D. P.; and Slack, M. G. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9(2/3):pp. 237–256.

Brooks, R. A. 1985. A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA.

Estlin, T.; Volpe, R.; Nesnas, I.; Mutz, D.; Fisher, F.; Engelhardt, B.; and Chien, S. 2001. Decision-making in a robotic architecture for autonomy. In *6th International Symposium on Artificial Intelligence, Robotics, and Automation in Space (i-SAIRAS)*.

Firby, R. J. 1989. Adaptive execution in complex dynamic worlds. Technical Report YALEU/CSD/RR #672, Yale University.

Gat, E. 1997. Esl: A language for supporting robust plan execution in embedded autonomous agents. In *Aerospace Conference, 1997. Proceedings., IEEE*, volume 1, pp. 319–324 vol.1.

Gat, E. 1998. On three-layer architectures. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems* pp. 195–210.

Ghallab, M.; Alami, R.; Hertzberg, J.; Gini, M.; Fox, M.; Williams, B.; Schattenberg, B.; Borrajo, D.; Doherty, P.; Morina, J. M.; Sanchis, A.; Fabiani, P.; and Pollack, M. 2003. A roadmap for research in robot planning.

Ingrand, F. F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 43–49.

McDermott, D. 1991. A reactive plan language. Technical Report YALE/DCS/TR-864.

Rosenblatt, J. K. 1998. *Field and Service Robotics*. Springer-Verlag. chapter Utility Fusion: Map-Based Planning in a Behavior-Based System, pp. 411–418.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition.

Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, 1931–1937 vol.3.

Simmons, R. 1994. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation* 10(1).

Volpe, R.; Nesnas, I.; Estlin, T.; Mutz, D.; Petras, R.; and Das, H. 2001. The claraty architecture for robotic autonomy. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, 1/121–1/132 vol.1.

Woolley, B., and Peterson, G. 2007. Genetic evolution of hierarchical behavior structure. *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation* pp. 1731–1738.

Woolley, B. 2007. Unified behavior framework for reactive robot control in real-time systems. Master's thesis, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright-Patterson AFB OH. AFIT/GCS/ENG/07-11.