N94-30556

6051

AIAA-94-1205-CP

530-63
207653
P. 7

A STREAMLINED SOFTWARE ENVIRONMENT FOR SITUATED SKILLS

Sophia T. Yu, Marc G. Slack and David P. Miller
The MITRE Washington AI Technical Center, Mail Stop Z401
7525 Colshire Drive, McLean, Virginia 22102-3481
syu or slack or dmiller @starbase.mitre.org (703) 883-7738

Abstract

This paper documents a powerful set of software tools used for developing situated skills. These situated skills form the reactive level of a three-tiered intelligent agent architecture under development at the MITRE Corporation. The architecture is designed to allow these skills to be manipulated by a task level engine which is monitoring the current situation and selecting skills necessary for the current task. The idea is to coordinate the dynamic activations and deactivations of these situated skills in order to configure the reactive layer for the task at hand. The heart of the skills environment is a data flow mechanism which pipelines the currently active skills for execution. A front end graphical interface serves as a debugging facility during skill development and testing. We are able to integrate skills developed in different languages into the skills environment. The power of the skills environment lies in the amount of time it saves for the programmer to develop code for the reactive layer of a robot.

1 Introduction

Within the short history of robotics research, many different approaches have been proposed for creating the intelligent component of an autonomous entity. The majority of them were considered unsatisfactory for developing an R2-D2-like robot. For instance, the traditional school of thought, grounded on real-world modeling and planning, was criticized for the discrepancy between the real-world and the computer model. Although more recent approaches based on simple reactivity algorithms produced surprisingly intelligent appearing robot behaviors [2], many argue that these robots are incapable of complex tasks [9].

Despite the absence of an R2-D2 legacy, the two approaches lay the foundation for a middle ground approach. This approach incorporates both a deliberative and a reactive component. Most people will agree that an architecture that includes both components seems to be a sensible way to build the robot brain. After all, human beings appear to have both reactive and deliberative faculties [5]. Deliberation allows the robot to make plans and predictions, and reactivity allows the robot to respond effectively to uncertainties. However, the tough question is how to put together a system with both a deliberative and reactive component?

A number of systems of this nature have been proposed and tested. Rosenschien and Kaelbling developed the situated automata theory for robot control [6]. The authors describe a system which would allow high-level description of the environment to be translated into situation appropriate low-level control activities. Arkin's AURA builds an accurate global model of the world and passes this information to a vector summing control layer [1]. Payton and Rosenblatt's architecture has four layers: task-level planning, global path planning, local path planning, and reflexive control [8]. The first three layers are equivalent to a traditional planning layer. The fourth layer consists of numerous reactive experts whose influence on the actuators is arbitrated by a central module. Erann Gat proposes a three-tiered architecture ATLANTIS [4]. In this architecture, a sequencing layer integrates and pipelines the deliberative and the reactive functions, which ultimately control the robot.

The MITRE autonomous systems laboratory also participates in the pursuit of an intelligent robot system characterized by deliberation and reactivity [10]. Our system is similar to an architecture originally proposed by Firby [3] and further developed by Gat [4]. The MITRE intelligent agent architecture (MIAA) consists of three interacting layers. The deliberative layer makes high-level decisions which require reasoning about resource and time constraints. The reactive layer consists of situated skills that react to the situation at hand. Finally, a sequencer which acts as a temporal and syntactic differential between the reactive and deliberative layers, decomposes planning structures into the appropriate activations and deactivations of the robot's skills (See Figure 1).

The design of the three-tiered architecture is based on two important concepts: heterogeneity and asynchrony. The system is heterogeneous in that it is composed of components that are structurally different, e.g., time-consuming decision-making modules versus instant response reactive modules. The system is asynchronous in that deliberative, sequencing, and reactive modules are executed in parallel and communicating to each other via asynchronous message events. This guarantees that reactive modules respond to the situation at hand in a timely fashion and that an adequate amount of time is allocated for deliberative processes.

This paper describes the design of an environment for constructing situated skills (the term appears in [10]) which form the lowest layer of MIAA. The environment must establish communication channels with the sequencer in order to accept commands from the deliberative layer. The environment must also be able to streamline the skills so that each skill provides a timely response for given sensor data. Moreover, the environment must provide a means for integrating the individual skills into a dynamically reconfigurable library of robotic skills.

In this paper, we shall answer explicitly three questions in sequential order. What are the necessary requirements for engineering a desirable skill development environment? How is the environment implemented? Finally, why does the implemented skills environment constitute a valuable set of tools?

2 Requirements

There are two sets of requirements for engineering a desirable software environment for the situated skills. The first set of requirements address the specifications for an infrastructure capable of controlling skills. The second set of requirements focus on software engineering related issues. This second set of requirements is especially important since robotic systems are becoming increasingly complex, raising information management as a central issue.

In order to better understand the first set of requirements, it is necessary to examine what is required of the reactive layer separately from the rest of the architecture or more accurately from the sequencing layer. The main function of the sequencing layer is to provide runtime situation-driven execution [3]; its job is to transform a set of partially ordered plan steps from the deliberative layer into the necessary set of skill activations and deactivations to accomplish the high level plan for the current situation. To achieve this, the reactive layer must accommodate the demands of the sequencing layer. For example, the reactive layer must provide perceptual information continuously and without delay. In addition, the reactive layer must provide a mechanism that allows the sequencing layer to tailor the behavior of a skill according to different situations.

The skills environment divides naturally into two main parts: a skill library and a mechanism which pipelines the skills and interfaces with the sequencer. The skill library provides the basic functionality of the instantiated autonomous agent. Skills include not only the computational elements of perception but the interfaces with the robot's hardware. This latter feature provides a mechanism for the interchangeability of physical devices and portability of the architecture. Each skill also provides conditional mechanisms which allow the sequencer to adapt to the situations at hand. Finally, each individual skill provides control switches for activation and deactivation.

A number of temporal and dependency requirements dictate the design of the mechanism used to control the skills. First, all components of the architecture must have access to relevant perceptual information and have the capability to control system actuators. Secondly, the perceptual information should be available continuously and that at every instance in time, actuation commands are assigned to external actuators for given sensor data. Thirdly, there should be an asynchronous and distributed execution of skills. In other words, it should be possible to execute skills in parallel as long as there is no contention of resources among the parallel skills. Lastly, unlike the deliberative functions, these situated skills should have low-commit and fast response time since real-time performance is necessary.

The second set of requirements for the skills environment is derived from the vantage point of a good software engineering project. One of the main goals of a good software project is to increase the number of users of the program. The design strategy taken is to encapsulate the details of the skills environment, therefore those without knowledge of the architecture should be able to program skills tailored to the domain of their robot. Another design requirement taken into consideration is the reuse of old software written in different languages. The implementation phase for configuring a robot can be substantially cut if existing code can be reused as part of the skills library and thus integrated into the overall control paradigm. This requirement can be satisfied by implementing standard encapsulations

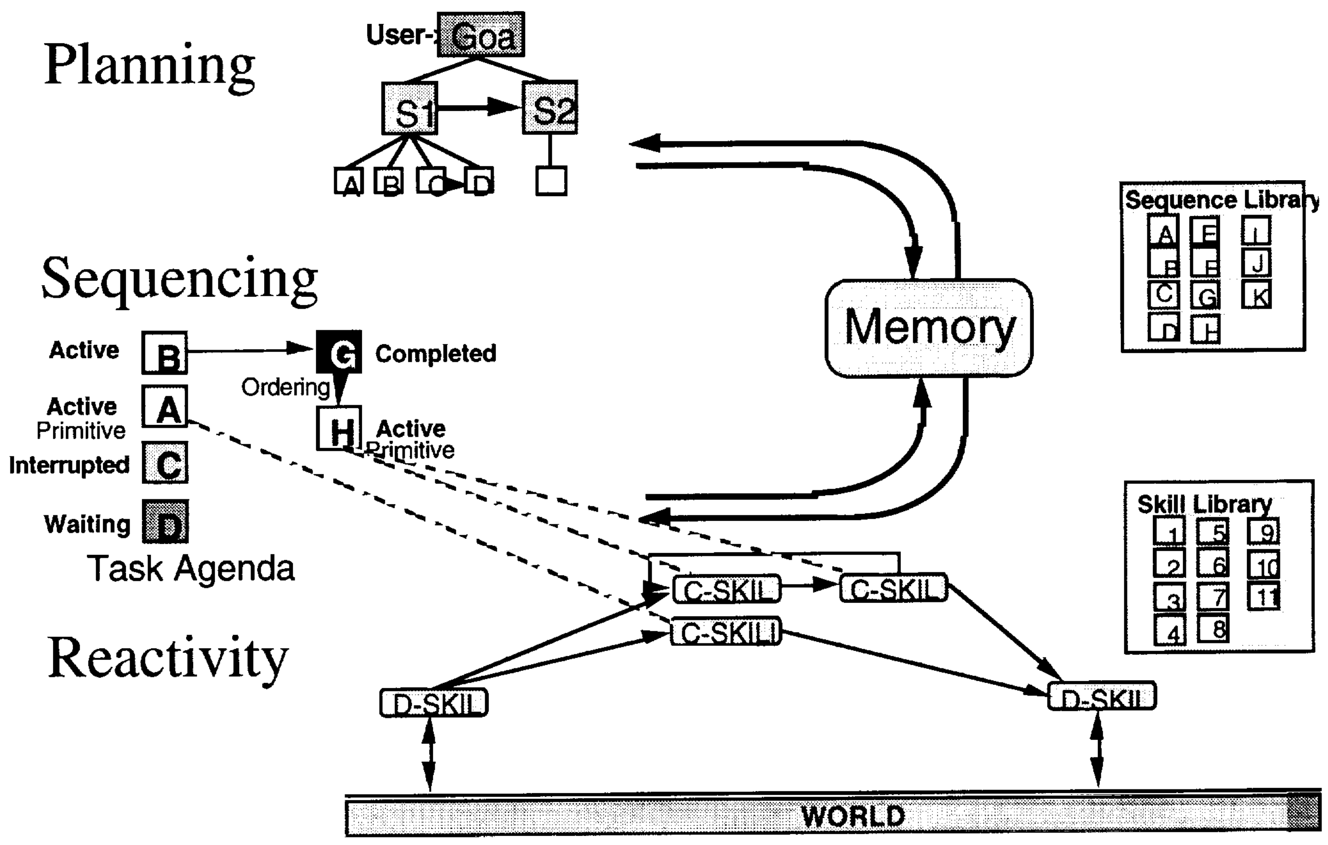

Figure 1: The flow of information through MIAA.

and interfaces for skills. In addition, there should be support mechanisms to facilitate the painful process of debugging. Lastly, it should be easy for users of the skills system to activate and deactivate skills manually, so that skills can be debugged and tested in isolation prior to their integration into the rest of the architecture.

3 Structure

This section discusses the design of the skills environment. We took an object oriented approach in designing the skills environment. Essentially, the skills environment is composed of two classes of structures: skill and skill manager. Each class is associated with objects and functions, and two instances of a class have access to the same class objects and functions. We will discuss the two classes of skills by focusing on their associated objects and functions.

3.1 Skills

The skills object class defines a set of structures which support the reactive modules. These structures serve as encapsulations of the reactive modules, to provide a standard interface to all of the reactive modules. Additionally, this interface includes structures for textual or graphical display of parameters, parameter logging and debugging purposes.

The objects for the skill class are the common data structures among every reactive module. For example, the input and output data structures of reactive modules are objects. A good portion of objects are used to support the scheduling of skill operations. There are also objects which are used for interfacing the skills with the sequencer. A priority slot is also available for resolving conflicts among skills in contention for resources. In addition, memory locations are allocated for recording parameter values and for displaying information associated with a skill.

A skill's parameters are important data objects which deserve special attention. Each skill has its own unique set of parameters. These parameters are singled out because they play an important role in bringing about situation-driven execution. These parameters are used and modified by the sequencing layer to tailor the behavior of a skill to the situation at hand. For instance, the sequencing layer may increase the priority value of a skill to give it temporal precedence over other skills. Another example of a skill parameter is the `safe-distance` parameter in the `runaway` skill. When an instantiated robot encounters an obstacle-dense region, the sequencer may decide to decrease the value of the `safe-distance` parameter so that the robot is able to pass through the obstacle region, then restore the parameter later when the robot is out of the congested region.

Because each skill will generally have a different set of inputs and outputs, there are numerous functions that are automatically generated by the skill object class when a specific skill is instantiated. This allows the environment to be skill independent and frees the skill designer from the concerns of interfacing the skill to the other skills in the skill library. The designer must only be concerned with the input and output requirements and the necessary computation; all interfacing issues are automatically handled by the development environment.

The skill class is further categorized into two sets of skills, those which are purely computational (or cskills) and those which interface with devices (or dskills).

3.1.1 Cskills

Cskills are skills which obtain their inputs from other skills and perform a computational transform on the inputs and pass the transformed values to other skills. Cskills can be operated either synchronously or asynchronously. In the synchronous case, a cskill will run its computational transform whenever it is given a new set of inputs, blocking until the computation has completed and a new set of outputs has been generated. In the asynchronous case, the cskill will continuously perform its transform on the currently available inputs and will respond immediately (like dskills) with the latest answer. The asynchronous cskills are especially useful when cskills are being executed on a distributed computer network.

3.1.2 Dskills

A dskill serves as an interface to a physical device (see Figure 2). This interface brings actuation commands to and obtains sensory data from the physical device. Rather than performing a computational transform on the inputs of the skill, a dskill buffers its inputs and provides a mechanism for sending those inputs to a device. A dskill will also buffer the latest sensor readings from the device. These sensor readings are provided as the users output from the dskill. The reason behind the creation of a separate skill class is to

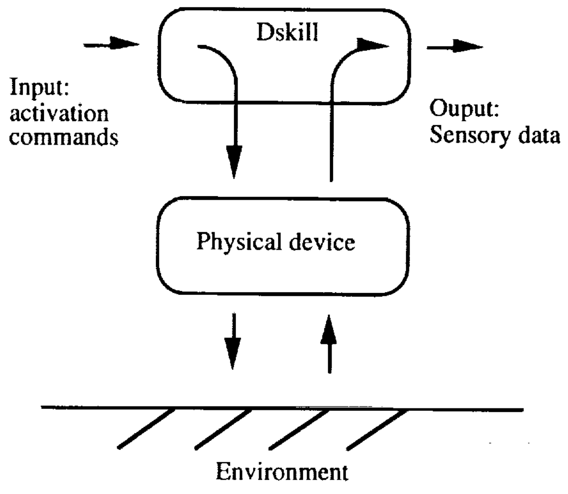

Figure 2: Information flow of a dskill.

provide the developer with mechanisms for handling the delay in communications associated with device drivers. Thus a dskill will always respond immediately and will not block for communications events as device interfacing is handled in a separate asynchronous process.

3.2 Skill Manager

Instances of the skill manager class are responsible for the timely activation and deactivation of reactive modules. They provide inter-skill communications as well as communications with the sequencing layer. The paradigm used for scheduling the execution of the skills is a data flow mechanism.

3.2.1 Data Flow

The objects of the skill manager class are mainly data structures supporting the data flow mechanism. For instance, there are allocations that store a list of instantiated dskills and cskills. A slot is reserved for counting the number of cycles the data flow mechanism has executed. There is also a state slot which regulates the sequence of function invocation in a cycle. In addition, there are data structures that keep track of the activation and deactivation requests made by the sequencing system.

There are two critical functions central in understanding the workings of the data flow. They are `do-periodic-step` and `update-record`. The `do-periodic-step` is the driving function behind the data flow. Once this function is invoked, an infinite loop starts. During each cycle of the loop, two functions are called in sequential order:

do-next-skill, and cleanup. The first function creates an environment for the executing the set of activated skills, and the last function destroys the environment created preparing it for the next cycle.

The order in which the individual skills are executed during a cycle depends essentially on input readiness. The update-record function is instrumental in preparing for the readiness of skills. Every time a skill produces an output, the update-record function checks these output data structures against the input data structures to skills yet to be executed. If there is at least a partial match in data structure, the update-record function avails the matched data structures to the yet to be executed skills. When all the inputs to a skill are available to it, the skill is ready and is executed the next time that the skill is considered for execution.

3.2.2 External Control

For the system to be of use there must be a mechanism for determining which set of skills should be active at any moment of time or the overhead of this paradigm is wasted as one could have simply hacked up a large C file to provide the necessary utility. However, as mentioned earlier, there are different syntax and semantics required to construct the different levels of abstraction necessary for constructing autonomous agents. In our system, the sequencing layer is assumed to handle the process of activation and deactivation of skills. The reason for handling skill activity in the sequencing layer is that the sequencer is maintaining an explicit representation of the robot's current situation (e.g., navigating down a hall, opening a door, etc.). It is beyond the scope of any individual skill in the currently active network of skills to be able to interpret the context and decide how the current sensor information should be interpreted with respect to the current task.

To support the use of a sequencer, the skill manager maintains an asynchronous communications link through which requests are made. The skill manager handles not only requests for activation and deactivation of skills, but also requests for state monitoring, parameterization, and value queries. The ability of the reactive layer to take initiate monitoring events is critical to any multi-layered architecture as the sequencer knows which information is critical to the task yet the skill level represents the only location where high frequency information can be captured. For example, the sequencing system could setup a monitor asking the skill manager to send an asynchronous event back when the robot's front sonar reads less than 10 inches. Because the sequencing and skill layers of the architecture operate in parallel, such information is too transient for the sequencing layer to capture directly yet there is insufficient information in the skill layer to determine that the information means that the robot has reached the ticket counter. In a similar fashion, the skill manager also allows the sequencer to make direct queries of the state of the skills in the reactive layer, thus allowing the sequencer to obtain instantaneous primitive value readings (e.g., is barcode 3 currently visible?). Lastly, the skill manager allows the sequencer to set the parameters of individual skills. This allows the skills to be dynamically configured for the situation at hand.

4 Implementation

The current skills development and execution environment is implemented within the Macintosh operating system. The structures discussed in the last section are implemented with the Common Lisp Object System (CLOS) application for two reasons. First, it is easy to realize the object oriented features of the skills environment; the CLOS package has constructs that accommodate class objects, functions, and instantiations. Secondly, the CLOS package has an easy to program graphics package. The graphics package is used to allow the skills programmer to control the operations of the skills environment with ease. The graphical interface is implemented mainly for debugging purposes. The next paragraph explains the implementation of the graphical interface in more detail.

Since there are two classes of structures, two types of graphical interface were designed and implemented: one for activating the individual skills and one for activating the data flow mechanism. The graphical interface for the skills class has mouse-clicking buttons for activation, parameter logging, and textual display. In addition, it has optional buttons for changing the values of skill parameters. The graphical interface for the skill manager class has a button for activating the data flow mechanism and a variable set of buttons for individually activating the set of instantiated skills. Note that these graphical interfaces are objects of the skill and skill manager class. These buttons of the graphical interface allow the skills programmer to activate a skill or a set of skills and to observe runtime execution results.

To utilize the environment, the job of a skills programmer is fairly simple since most of the inter-skill communication and graphical interfacing issues are handled by the generic skill object. To implement a skill, the skills programmer needs to specify only three things: input, output, and computational body. These three items must be properly placed into the template provided by the structures within the skills class.

The skills environment is set up in a way to permit the implementation of a multilingual skills library. This is possible for two reasons. The encapsulated and modular design of the skills environment keeps the differences among skills within the skills themselves, while the uniform and standard features of the skills environment provides a direct way for communications among the disparate skills. We have implemented skill construction facilities to allow a programmer to create skills written from C, C++, Pascal, Assembler, LISP, and REX [7]. To program a skill in a language other than LISP requires slightly more work. In addition to specifying the three things in a different language, the input and output data structures must be specified in LISP so that memory allocations are made properly in the skills environment.

The implemented skills are fairly easy to debug with the help of the graphics interface. To debug a skill, the first step is to instantiate that skill. Next, the user may want to change the values of the skill parameters to the desired values by clicking on the parameter button(s). Pressing the activation button will start the execution of the skill. The user is able to view the runtime parameters of individual skills by clicking on buttons on the graphical interface window. Clicking on the *show data* button invokes a corresponding window which is capable of displaying text. The *show input* and *show output* buttons forces the input and output of skills to be displayed on the secondary window, and the *verbose* button allows the display of any print statements generated internally to the user's skill. In order to debug a skill within the context of a set of skills, it is necessary to activate the data flow mechanism. The first step in this process is to instantiate a set of needed skills. The next step is to instantiate a skill manager for pipelining the instantiated skills. Pressing the run button on the skill manager window will start the data flow mechanism. Runtime results of activated skills can be viewed on the secondary windows of the instantiated skills.

5 A Valuable Set of Tools

The power of the skills environment lies in the amount of time it saves for the skills programmer. During virtually every stage of the skills development cycle, time is conserved. For instance, learning time is shortened. The programmer does not need to have extensive knowledge of the skills environment to program a skill and integrate it into the skills environment. The details of the skills environment are encapsulated; the object-oriented constructs of the skills environment essentially provide templates for programming and instantiating reactive modules.

Moreover, programming time is significantly reduced in two ways. First, the skills programmer does not need to worry about interfacing with other skills or other components of the architecture. Recall, the programmer needs to specify only three things to program a skill. Secondly, the skills environment allows the reuse of existing code. This capability is valuable since rewriting code such as the robot's inverse kinematics is a task one would like to do only once.

In addition, debugging time is decreased. As mentioned in the last section, there are a number of debugging facilities available. The graphic interface allows the easy operation of these facilities. Also, the modular decomposition of the skills allow the individual skills to be debugged in isolation or in the context of other skills.

The maintenance of the skills library also becomes less time-consuming. Modifying the skill library to adapt to the changing capabilities of the instantiated autonomous agent is quite straightforward. Since the interface to physical devices is encapsulated in dskills, adding, deleting, or replacing dskills is all that is needed to adapt to a change in physical devices. The components of the rest of the skills environment remain unaltered.

6 Proposed Experiment

Before concluding the paper, we relate our experience of programming a set of skills for our proposed experiment with MIAA. We offer this to demonstrate the time conserving way of programming skills. The proposed experiment is for our Denning robot which must deliver a message to our department head.

The robot wakes up in its humble abode: the autonomous systems laboratory in the basement of the building. It wanders around the laboratory, avoiding obstacles and looking for a door to exit. Once the door is found, it exits the door. Since our department head's office is on the fourth floor, the robot must find the elevator first. It directs itself to the elevator using a combination of hallway following, door detection and intersection detection. Once the elevator doors are found, the robot pushes the elevator button and waits for the elevator. When the elevator comes it must determine which of the four elevator cars actually arrived. Upon entering the elevator the robot must push the button for the fourth floor. When the elevator stops on the correct floor, the robot exits.

It directs itself down the corridor to the department head's office and delivers the message.

The set of behaviors described above can be accomplished with a standard set of situated skills and a specialized set of task specific skills. Some examples from the standard set are obstacle avoidance, wall following, wandering, object tracking, and object homing. An example from the specialized skill set is a module that allows the robot to push an elevator button.

As an example of how the sequencer coordinates the situated skills to bring about the set of behaviors described above consider the task of exiting the autonomous systems laboratory. Five skills are key in creating this behavior: obstacle avoidance, wandering, barcode tracking, locating the position of the door, and position homing. A barcode is placed in the vicinity of the door so that the robot can reliably recognize the door's general location. The sequencer first activates wandering, obstacle avoidance, and barcode tracking, so that the robot wanders around the laboratory, avoiding obstacles and looking for the barcode associated with the door. Once the location of the barcode is found, the sequencer activates the barcode homing module, commanding the robot to move in front of the barcode. Finally, after the robot moves to the vicinity of the door, the sequencer activates a module that computes the necessary position clues for isolating the door opening and driving the robot through the door.

From this simple example, you can see the utility of being able to activate and deactivate skills depending on which aspect of the overall task the robot is currently working on. For example, it makes little sense to spend valuable computation time identifying the door opening until the robot is in the vicinity of the door. We are collecting metrical information, to provide evidence of the utility of the explicit sequencing of skills verses the implicit sequencing of skills typical of more ad hoc reactive techniques. It is our hypothesis that as the size of the class of tasks within a given domain increases the utility of taking a more structured and engineered approach to design of robotic intelligence will clearly win out over the "purely" reactive techniques.

7 Final Words

The skills environment is a powerful technology for a number of reasons. It abstracts the skill developer from the details of communications protocols and graphical user interface issues which the environment provides. A person writing code for a sequencer has only to concern themselves with which skills are needed and can ignore all of the inter-skill communications issues as these are handled by the data flow mechanism of the skill manager. We believe that by providing methodology to the creation and use of reactive modules that the work in reactive control of robots can move out of the ad hoc creation of task-specific demonstrations into the world of assisting in the solution of real-world problems.

References

[1] R.C. Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *International Journal of Robotics and Autonomous Systems*, 6(1-2):105–122, 1990.

[2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[3] R. J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University Department of Computer Science, January 1989. see Technical Report 672.

[4] E. Gat. *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots.* PhD thesis, Virginia Polytechnic Institute Department of Computer Science, April 1991.

[5] P.N. Johnson-Laird. Mental models in cognitive science. *Perspectives on cognitive science*, pages 147–191, 1981.

[6] L. Kaelbling and S. Rosenschein. Action and planning in embeded agents. *Robotics and Autonomous Systems*, 6:35–48, 1990.

[7] L. P. Kaelbling. Rex: A symbolic language for the design of parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, 1987.

[8] D. W. Payton. An architecture for reflexive autonomous vehicle control. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1838–1845, April 1986.

[9] M. G. Slack. *Situationally Driven Local Navigation for Mobile Robots.* Department of computer science, Virginia Polytechnic Institute, April 1990. also published as Jet Propulsion Laboratory Publication 90-17.

[10] M. G. Slack. Sequencing formally defined reactions for robotic activity: Integrating RAPS and GAPPS. In *Proceedings of the SPIE Conference on Sensor Fusion*, November 1992.