MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 291 December 1986

Critical Analysis of Programming in Societies of Behaviors

Peter Cudhea

Abstract: Programming in societies of behavior-agents is emerging as a promising method for creating mobile robot control systems that are responsive both to internal priorities for action and to external world constraints. It is essentially a new approach to finding modularities in real-time control systems in which module boundaries are sought not between separate information processing functions, but between separate task-achieving units. Task achieving units for complex behaviors are created by merging together the task-achieving units from simpler component behaviors into societies with competing and cooperating parts. This paper surveys the areas of agreement and disagreement in four approaches to programming with societies of behaviors. By analyzing where the systems differ, both on what constitutes a task-achieving unit and on how to merge such units together, this paper hopes to lay the groundwork for future work on controlling robust mobile robots using this approach.

1 Introduction

Programming in societies of behavior-agents is emerging as a promising method for creating mobile robot control systems that are responsive both to internal priorities for action and to external world constraints. This paper hopes to survey both the areas of agreement and the areas of disagreement in four approaches to programming with societies of behaviors. Programming in behaviors is not yet a common idea. Only a few systems exist to be compared, and these systems exhibit more differences with each other than similarities. Even so, by comparing and contrasting the features they choose to emphasize, I hope to form an image of what a behavioral programming system should encompass.

Section 2 presents general background on the idea of programming in behaviors, and its advantages compared to more traditional ways of decomposing mobile robot control systems. In addition, it introduces five issues that can be used to make comparisons between the surveyed approaches:

- underlying models of distributed computation
- differing ideas of what constitutes a behavior
- methods of mediating between behaviors
- abstractions of behaviors
- integration of perception into behaviors.

Section 3 introduces the four systems and describes what motivates each group to adopt the idea of societies of behaviors. Section 4 compares and contrasts the treatment of the above issue in each of the four systems to derive lessons for future work in this area. Finally, section 5 concludes by discussing the relevance of this approach.

2 Background on Societies of Behaviors

2.1 General Comments

Programming in behaviors is essentially a new approach to finding modularities in mobile robot control systems. Modularity is the division of a complex problem into pieces that can be implemented separately. The assumption is that the connections between modules are looser in some sense than the connections within a module. Where this is true, modular systems become easier to implement and understand than systems lacking modular structure. In addition, finding good module boundaries is a prerequisite for making good use of distributed computation. Good module boundaries are thus related to communication requirements: it makes sense to draw module boundaries where communication is of low information complexity.

Traditional mobile robot systems use a top-down *horizontal* decomposition into information processing units [Giralt, *et al* 1984] [Moravec 1983] [Nilsson 1984]. For example, there are modules for perception, world modeling, planning, task execution, and motor control. The assumption of this decomposition is that connections between separate information-processing units do indeed have low information complexity. However, systems operating in complex environments seem to require many alternative strategies for use in different situations. To satisfy the requirements of these strategies, the interfaces between information-processing units grow more and more complex. The traditional horizontal decomposition impedes the idea of developing multiple alternative strategies.

The programming in behaviors approach rejects the information-processing decomposition in favor of an alternate *vertical* decomposition that isolates each of the multiple alternative strategies in its own module, a task-specific unit. Each task-specific unit contains tightly coupled perception and action components that select and guide actions to achieve some specific task. Both perception and action descriptions are viewed as being task-specific. The perception and action required for interesting behaviors are very closely related to each other and shouldn't be in separate modules. In addition, separate behaviors interact only weakly, mainly when competing for resources that are to be used in separate ways. Thus, the vertical decomposition into separate behavior-units seems an appropriate vehicle for implementing control systems for a complex world.

A simple behavior is a unit comprising linked perception and action capabilities to achieve a distinguished goal or subgoal. Each behavior has a limited responsibility for suggesting ways to achieve its task in the actual world. Complex behaviors are created by building societies of simpler behaviors that support each other's weak points.

Mechanisms for creating societies of agents mirror ways of decomposing a desired overall behavior into simpler pieces. In many cases, separate behaviors can run in parallel: relying on each other to maintain related goals but interacting only by manipulating the state of the world. For other situations, the suggestions of one level of agents can be carried out by another level of agents, leading to a hierarchical decomposition of control. Finally, several behaviors may make competitive suggestions for action using independent strategies for responding to the world. Mediating between the competitive strategies based on information about the relevance of each strategy can lead to a very robust combined strategy that quickly adapts to changes in the world situation.

Programming in behaviors suggests an approach to dealing with surprising events. A surprising event is reflected in the data used to select among competing strategies, and is handled by selecting the strategy that seems best able to handle the new situation. Each strategy can be fairly simple, because it can assume other behaviors are watching out for the cases that they can handle better. Multiple overlapping strategies support robustness, for once a failed strategy gets out of the way other strategies can take over.

Behavior programming opens up a new approach to implementation of perception, by removing the requirement for a single clean interface between perception and control. Each type of action can receive its own version of the perceptual information. Different strategies may even receive information that looks contradictory, since the information has been processed in the context of competing strategies. The output of perception is not a single coherent world model, but many separate world models, each appropriate to some task.

2.2 Specific Issues

Each system surveyed here builds societies of behaviors as a set of conventions and abstractions on top of some underlying model of distributed computation. Assumptions arising from the underlying model have wide-ranging effects on the mechanisms proposed. In particular, the amount of variation in the topology of the distributed agents deeply affects many control issues. Moreover, behaviors alone cannot capture the entire problem of building control systems, so it is interesting to note the ways in which systems escape back to the underlying computational model.

Different mechanisms for programming in societies of behavior-agents have different models for what a behavior agent should do. All the systems surveyed here agree that the fundamental behavioral unit includes perception, decision, and action components to achieve or maintain some goal. But is there some structure to the way perception and action are linked, or are they linked *ad hoc*? Moreover, what kinds of inputs and outputs does a behavior-agent have? Inputs might include world data, control requests, and parameters for action; outputs might include new commands for other agents, new world data, or status feedback to other behaviors.

Mechanisms for mediation between behaviors must support effective coordination without violating modularity. Expertise for resolving the competing claims can be centralized in a distinguished *mediator* that unifies the competing behaviors into a coherent unit. Alternatively, the expertise can be distributed back to the individual behaviors, allowing the mediator to be very simple. Both mechanisms have key advantages which will become apparent in the discussion.

Abstractions allow dealing with a unit without knowing its internal structure. At compile-time, we need structural abstractions to tell us how to build a complex control-system from simpler pieces. At run-time, we need functional abstractions that allow agents to suggest actions in abstract terms that are translated by other agents into a more concrete implementation. Anything that acts like a behavior should be able to be treated like a behavior to build up new structures. We will try to point out the tension between goal abstractions and structuring abstractions.

Perception as well as action can be task-specific. Moreover, successful perception must be able to merge multiple points of view to achieve a combined strategy. Thus, it is possible to view most of the perceptual processing as using the same sorts of mechanisms that motivate the behavioral viewpoint. The four approaches take different stances on how tightly perception is integrated with behaviors. The underlying question is really whether modeling the world is a separate sort of process that should be separated in some way from selecting and guiding actions.

3 Four Systems for Programming in Behaviors

The next section of the paper introduces four systems that embody different approaches to programming in behaviors. For each system, we first describe its overall goals to put the design tradeoffs into perspective, and then survey the system's approach to the issues described above. Each of the systems is only a partial model of programming in societies of behaviors. None of the systems addresses all the issues described here. In some cases descriptions of the issues are extrapolated from what has been published.

3.1 The Hughes AI Center Approach

David Payton at Hughes AI center has developed a mobile robot control system that arrives at many of the features of a society of behaviors based on analyzing the tradeoff between immediacy and assimilation [Payton 1986]. The project goal is an autonomous vehicle for accomplishing defined missions without direct operator involvement. The overall structure is imagined as a four-layer hierarchy with a mission planner, map-based planner, local-planner, and a reflexive planner. Only the lowest level has been implemented; only it is discussed here.

To achieve immediacy, Payton claims higher levels must delegate authority to lower levels to respond in real time, i.e. the lower level must include active agents. Specifically, Payton carves the reflexive planning level into independent behaviors that tightly link perception and action. Each behavior specifies:

- a perceptual part consisting of inputs from *virtual sensors*
- manipulable parameters, and
- a strategy, written as arbitrary Lisp code, that uses the sensors and parameters to suggest appropriate actions.

The only actions supported are setting the forward speed and turn rate of the vehicle. In addition, behaviors can apparently signal failure or success to whatever agent invoked them. Example behaviors are `follow-road-edge`, `slow-for-obstacle` and `turn-for-obstacle`.

Perception in Payton's system is encapsulated in *virtual sensors*. Each virtual sensor provides task-specific information at a logical level. For example, a typical virtual sensor would output the location of an obstacle relative to the robot. In implementation, a virtual sensor is a process that describes how to derive the required data from an actual sensor, or from a combination of other virtual and actual sensors.

The underlying model of computation regards behaviors and virtual sensors as processes, say in a Lisp Machine system. Both behaviors and sensors are instantiated and deinstantiated dynamically as the situation changes. Behaviors are assumed to be independent; they communicate with each other only to compete for control of the vehicle. Communication is mediated through a blackboard, which fits well with dynamically changing process topology.

Mediation between behaviors occurs in two ways. First, a higher level planner selects the active behaviors by choosing an activity-set of the type described below. Second, competition between the active behaviors is mediated by a simple priority scheme. Each behavior attaches a priority to its suggestions, and the highest priority ones are acted upon.

Abstraction of behaviors permits one level of hierarchy–the activity-set. Each activity-set groups together a coherent set of behaviors to encode a particular method of acting in the world. By switching between activity-sets, the local planning level can adjust the reflexes of the vehicle to suit the perceived situation. Besides the list of behaviors, the activity-set includes other information to make the amalgamated behaviors work well together. This information includes relative priorities between behaviors, parameters for the behaviors, and initialization and termination actions to run when switching from one activity-set to another.

Several other features of activity-sets allow increased complexity in how the activity-set selects actions to match the situation. Taken together, these features go part way to unifying behaviors and activities into a general hierarchy of active behavior-agents, but fall short of a general mechanism. First, the activity-set can associate a list of known failure conditions with recovery behaviors to run when each failure is detected. Moreover, alternative mediation functions can be specified in cases where the simple priority rule is inappropriate. These two features give the activity-set more responsibility for actively controlling action as opposed to merely structuring the actions of other agents. Another feature relaxes the flat hierarchy of behaviors by introducing a variant form of activity-set that includes a list of sub-activities rather than a set of behaviors. The sub-activities are run in a fixed repeating sequence. This appears not to be a general feature, however, but a way of introducing simple scripts of actions before the local-planning level is implemented.

Since virtual sensors are not directly controlled by behaviors, perception in Payton's model is still mostly general-purpose and not task relative. The activity-set mechanism does select which virtual-sensors should be running but does not guide their attention. Thus, the operation of a virtual sensor cannot be guided by the detailed requirements of the task to be performed.

To sum up, Payton's system makes a number of simplifying assumptions in order to demonstrate the idea of delegating authority to lower levels of the control system for real-time response. A completely general system would include a more hierarchical behavior structure and more mechanisms for mediating and communicating between behaviors. The system is a refreshing counterpoint to more traditional mobile robot control systems, but should not be considered a full attempt at organizing the control system into a society of behaviors

3.2 The SRI Approach

Stan Rosenschein and Leslie Kaelbling of SRI International have developed two pieces of a model of programming in behaviors. The first is a computational model, REX, for creating and describing real-time embedded systems [Rosenschein and Kaelbling 1986] [Kaelbling 1986b]. The second is a proposal by Leslie Kaelbling for how to build modular, composable mobile robot control systems on top of REX by using hierarchically mediated behaviors [Kaelbling 1986a].

The overall motivation of the project is to develop tools for building reliable real-time systems that are situated in the world. A mobile robot is the prime example of such a system. One goal of the REX project is the creation of a new computational architecture that facilitates real-time decision making. In addition, the architecture must facilitate reasoning about *why* a created system works the way it does. This step requires formalizing the idea of the *knowledge* of the machine, where knowledge is a name for the connection between the internal state of the machine and the external environment.

The computational model for REX is divided into two pieces. First, REX specifies a particular type of real-time machine. Second, REX specifies a language in which this kind of real-time machine can be built up by using meta-programs that describe the machine's structure. There is thus a strict division between the run-time machine and the compile-time program that creates it.

The machine-model for REX is basically a special-purpose computing circuit built out of computation boxes and wires. The machine is built once and for all by the compilation process and does not change at run-time. For simplicity of reasoning, the created REX machines assume synchronous clocking of the whole circuit in discrete ticks. Primitive computation boxes are provided to the designer in two forms. First, function boxes compute a specified value of their input ports and put the result on the output port. Computation is assumed to be instantaneous, in the sense that any chain of function boxes can compute a compound function of the inputs within a single tick. Second, delay boxes latch a value from the inputs and present it as an output in the next tick.

The meta-programming model for REX is designed to make it easy to specify circuits by writing programs that describe how the circuit should be laid out. Subroutines in the meta-language can describe complex parameterizable circuit patterns in the layout. The programs can even be recursive, to lay out structures with a repeating or nested structure. Connections between computing boxes are specified using a declarative syntax. This is a key feature, for it allows endpoints of a connection to be created in any order and then unified together, since the order of creation of pieces doesn't affect the finished circuit. This model of meta-programming is a general tool that can be applied to any model of computation with fixed computing boxes and wires with fanout. It is important not to confuse the tool with the particular sort of machines it is currently used to create.

Kaelbling's approach to programming in behaviors arises from the desire for modularity in control systems, specifically the ability to construct control systems incrementally. The idea is to decompose the desired overall behavior of the mobile robot into independent sub-behaviors. The sub-behaviors can be implemented separately, then combined using mediation.

A behavior in Kaelbling's system is essentially a complete control system that reacts to inputs from a world model to suggest appropriate actions for the robot. The world model is the combined result of all the perceptual processes, working together to keep track of the world situation. The idea of task-specific world models tightly integrated with action is not addressed. Another issue not addressed is where the goals for the separate behaviors come from. Each behavior is completely independent from the others, so there is no idea of higher-level behaviors generating goals for lower-level behaviors to pursue. Behaviors can be general REX code. There is no specific model of what the internal structure of a behavior should be.

Mediation between behaviors is characterized by two key ideas: intelligent mediators and hierarchical mediation. Mediation always requires some intelligence to determine the relevance of the proposed actions to the world situation. With simple mediators, such as a weighted voting scheme, the intelligence about when each behavior is relevant is pushed back into the behaviors themselves. Kaelbling claims this pollutes each behavior with too much information about the context in which it runs, for it needs this information to decide how relevant its suggestions are compared to others. Centralizing the mediation decision is an improvement, for it allows the sub-behaviors to be more independent of one another. Mediators in Kaelbling's system can be arbitrarily intelligent, using world information to merge the suggestions of the component behaviors in appropriate ways.

Intelligent mediators facilitate the process of creating behaviors separately and then composing them into more complex behaviors. Specifically, a compound behavior, created by combining several sub-behaviors with a mediator, can be used anywhere a basic behavior is used. The result is not really a hierarchy of behaviors, but rather a hierarchy of mediators. The structure of mediation forms a tree in which all of the leaves are basic behaviors, the internal nodes are intelligent mediators, and the root of the tree controls the vehicle. Figure 1 shows a simple control system from Kaelbling's paper, which can be seen as a tree in which the hashed rectangles, the mediators, form the interior nodes.

The main advantage of hierarchical mediation is that it mirrors the process of incremental construction: to add a new behavior, at worst one need only change the mediators on the path from the new leaf position to the root. This type of encapsulation helps in verification, for a given behavior need only be checked in its interactions with the local context, rather than with the whole control system. Another advantage is that different levels of the hierarchy can use different mechanisms of mediation: weighted addition of commands at one place, fixed priorities at another. However, the hierarchical structure is purely a modularity of construction and *not necessarily* of control. For example, nothing in the model would be violated by collapsing a tree of behaviors to a single super-intelligent mediator that directly mediates the leaves of the tree without intermediate nodes (see 2).

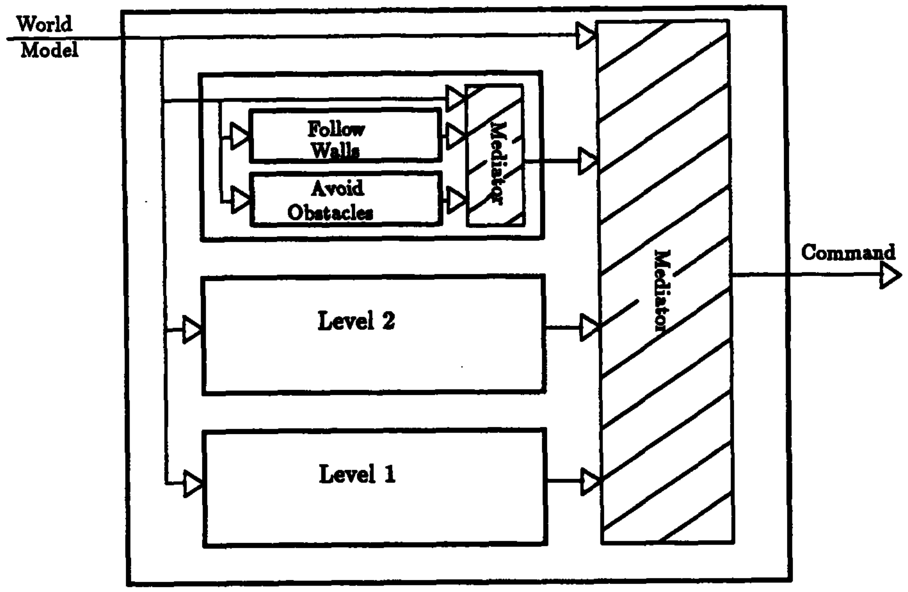

Figure 1: A Control System with Hierarchical Mediators

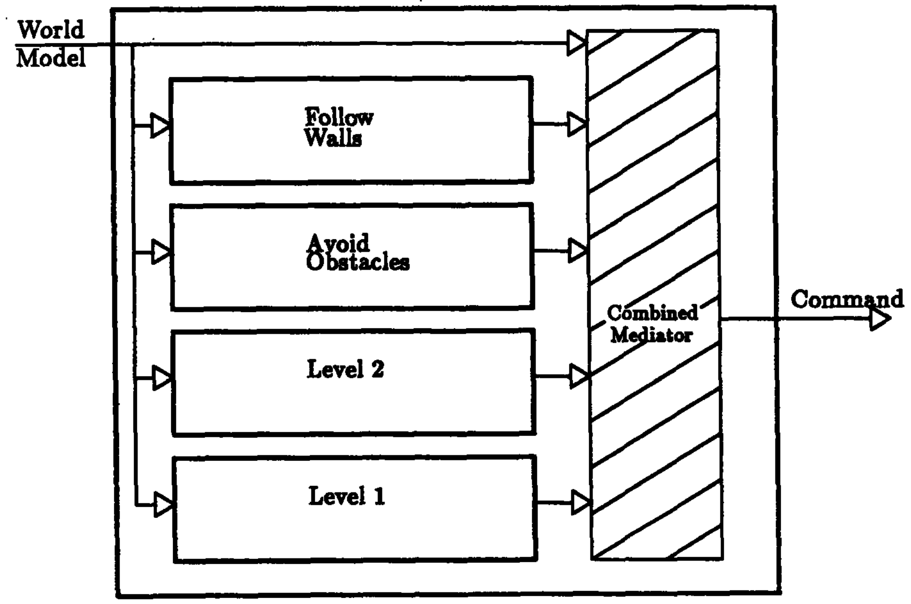

Figure 2: The Same Control System with a Collapsed Mediator

Kaelbling's system does not address the idea of using the output of one level of behaviors to control the operation of another level. Such goal decomposition in her system is buried inside each of the component behaviors. Thus, there is no need for the behavior mechanism to model the hierarchy of control. But the idea of coordinating separate agents to control a resource is too useful to be applied only once in the system, at the level of coordinating the overall behavior of the vehicle. The same issue arises at every point some resource cannot be duplicated and some decision must be made about how to use it. In particular, each way of setting a goal for some level of behavior requires putting aside other goals. A natural way of modeling this would be to have separate behavior-agents propose different ways of setting the goal, with a mediator to choose between them. Kaelbling's insistence that the separate behaviors are completely independent blocks this path.

Kaelbling's system allows some relaxation of the strict division between the perceptual component and the behavioral component. For one thing, the perceptual component can generate outputs at many levels of abstraction, from raw sensor data to complicated world models, and could even generate task-specific models where requested. In addition, Kaelbling's behavioral component can interact significantly with the perceptual component while it is operating, for example by directing its focus of attention. Thus, it is almost unnecessary to draw such a line between the behavioral and perceptual components. Why is the line important then? I believe the answer comes back to Kaelbling's insistence that behaviors be completely separate. It is important to isolate perception precisely so that outputs of perception can be shared between separate behaviors. Were mediation points to be introduced at internal points of the control system, it would be attractive to consider erasing the line between the perception and behavioral components to see if the same mechanisms would do for both.

3.3 University of Massachusetts Schema Approaches

Several projects at the University of Massachusetts at Amherst are basing research on the idea of the tight interconnection between perception and action. The idea of a schema for action, introduced by Arbib in the context of neuromodeling and brain theory [Arbib 1981] [Arbib 1972], has been adapted by other researchers to fields such as vision research, mobile robot path planning, and robot control.

A schema is basically a parameterizable description of a sensing or action task that encodes the static expectations about the task while specifying what parameters might vary. Perceptual schemas, essentially task-specific object models, recognize instances and retrieve the parameters. For example, there might be an active schema for each tracked object in a visual scene. Motor schemas correspond to parameterizable action possibilities; specifying the parameters allows the entire action to proceed. Both perceptual and motor schemas correspond to compiled expectations about what sorts of parameters will be useful to manipulate. However, there is no firm distinction between perceptual and motor schemas. A given schema could conceivably look like either a motor or perceptual schema depending on the observer's point of view. For example, a perceptual schema to track an object and read out its position might have action components to adjust the viewing angle or turn the head.

The schema idea is so general that the various applications of it have little specific in common besides the motivating principles of parameterized descriptions. For review here, I have singled out the work of Damian Lyons on $\mathcal{RS}$ (Robot Schemas) because it seems close to the view of behaviors as distributed active agents [Lyons 1986]. In addition, I briefly survey the mobile robot work of Ronald Arkin [Arkin 1987].

Lyons' goal in his thesis is the development of better ways of describing the connection between sensing and action in robot control programs. What he wants is a computational model appropriate for robotics. Like the SRI REX system, the resulting system includes both a way of building control systems and a method for reasoning about them. Previous robot programming environments have tried to shoehorn existing computational models onto robotics, introducing sensing and action as afterthoughts. His approach is to build his computational model from scratch, taking the connection between sensing and action as the primary construct. The resulting model of task-achieving units bears a great similarity to the idea of programming in behaviors, though his intended application of factory automation in planned environments leads him to downplay the need for real-time responses.

Lyons' system is built on an underlying very-general system for distributed computation. Many aspects of computation that are precomputed in REX must be set up on-the-fly by $\mathcal{RS}$. The overhead of doing this is significant enough that $\mathcal{RS}$ is inappropriate for real-time systems. In $\mathcal{RS}$, the configuration of the machine is dynamic; computing agents are instantiated and deinstantiated as needed by the application. Computing agents are full processes that may take time to compute results. To assure synchronization between agents, neither the sending agent nor the receiving agent of a communication can proceed until the communication is complete (i.e. synchronous communication). A group of interconnected computing agents can be described as an assemblage, which behaves like a single computing agent as far as the rest of the system is concerned.

The communication subsystem must have substantial intelligence to handle dynamic processes with fixed links between processes. Computing agents are set up one at a time by a following a procedural description of the agents and their interconnections. The smart communication system has to deal with dangling links that occur before the acquaintances of a computing agent have been instantiated, and after an acquaintance is deinstantiated. Connections to agents that don't exist yet are allowed by a form of forward reference that has a declarative flavor.

Lyons' uses this model of distributed computation to create *task-units* that he feels are an appropriate basis for robot programming. A task-unit is an assemblage of computing agents with the following pieces:

precondition a computing agent that acts as a guard to instantiate the rest of the task unit when the situation becomes appropriate for it.

perceptual schemas — a set of computing units that together constitute a task-specific object model that can be used to select and parameterize actions.

motor schemas — a set of computing agents that together constitute the basic actions available to this task unit.

connection map — the set of links between perception and action. The map is not intelligent *per se*, but it encodes the decision of which sorts of perception should control which sorts of action.

Lyons does not discuss any specific means of mediation. Any form of mediation could be performed by escaping to the underlying model of computation to introduce computing agents to perform the mediation. But Lyons does not discuss what types of mediation are appropriate to use with task-units. His examples tend to have no choices of this type, which is not surprising in systems designed for a planned environment.

Although Lyons' underlying model of computation has the assemblage mechanism to structure computing agents into groups, Lyons does not describe how to use the assemblage mechanism in structuring task-units into larger task-units. However, he presents an extended example from which we can infer some structuring principles. The main principle observed is to use a low-level task-unit as the representation of actions possible to a higher-level task-unit. What this does is to take the actions proposed by the higher-level task-unit and decompose them further into actions that are closer to the real hardware. Thus, the hierarchy of assemblages mirrors the goal hierarchy.

The idea of task units is really very similar to the idea of programming societies of behaviors. They arose from a concentration on the link between perception and action: that perception only makes sense in the context of some possibility for action. However, the lack of emphasis on real-time constraints makes the specific mechanisms chosen inappropriate for real-time situated systems.

Another project at U. Mass. is applying the idea of linked perception and action to path planning for a mobile robot [Arkin 1987]. The emphasis is not on a computational architecture, but on ways of using separate computing agents to keep track of separate injunctions on the movement of the robot. All the injunctions are encoded as forces on the robot using the potential field method. The forces can then be added together by a simple weighted sum. This project points the leverage that can be gained by having a common reference frame (in this case the potential field) where suggestions from separate behaviors can be combined. The architecture used is similar to Payton's. Schemas are separate processes that are instantiated and deinstantiated dynamically; they communicate using a blackboard architecture. This project is a promising approach to mobile robot programming, but since it introduces no new architectural issues I'll skip over it for now.

3.4 The MIT Approach

Like the SRI approach, the MIT approach proposes both an underlying computational model and a few first steps towards programming in societies of behaviors. Rodney Brooks has developed a computational model, called the Subsumption Architecture, intended for building flexible and robust control systems for robots with insect-level capabilities [Brooks 1986b] [Brooks 1986a]. Extending on this work, [Cudhea and Brooks 1986] proposed the idea of programming in societies of behaviors as a way of capturing common patterns that arise in this sort of control system.

The plan of Brooks' group is to develop mobile robots that can survive long-term autonomy in a complex environment operating under real-time constraints. To survive in such an environment, the robot's control system must be extremely flexible and robust in adapting to changing world conditions. Architectures for flexible and robust systems are currently understood only poorly. The project's methodology for creating such control systems is to build control systems incrementally in an approximation of evolutionary development. To match the great complexity of the world, the tasks set for the robot in the initial stages are simple: it is enough merely to explore the environment while avoiding harm. A robot with the capabilities of an insect is a worthy medium-term goal.

Unlike the other project's described here, Brooks does not aim at making creation of new behaviors *easy*. Rather, the aim is to enforce a set of realistic constraints on computation in order to discover what sorts of control are *appropriate* for a robotic insect. The model of computation uses distributed processors and a fixed-topology low-bandwidth interconnection network. Program control structure within each processor is restricted to act like a finite state machine, where complicated computation is only permitted in computing the state transitions and output values for the machine. Communication between computation agents has several features to facilitate adding new capabilities into an already functioning control system. These features include mechanisms for peeking at messages on existing communication paths and for taking control of existing paths by injecting new messages into them. Because of its simplicity, the architecture can be realized in many technologies.

Two recurring patterns that emerged strongly from current control systems using this architecture were the tight coupling between perception and action and the hierarchical decomposition of high-level goals. We developed these principles into a partial model of programming in behaviors. Each behavior is viewed as acting to achieve or maintain some goal for the robot in its environment. The basic behavior element is a difference engine, shown in Figure 3. Each difference engine consists of a measuring part (perception) that compares the goal with actual world information to decide what needs changing, followed by a decision part that decides which actions to take to change whatever needs changing most. The actions are not carried out inside this behavior, but are presented as goals to other behaviors for further processing. Compound behaviors are amalgamations of simple behaviors that act to achieve a more complicated goal by combining actions of several sub-behaviors.

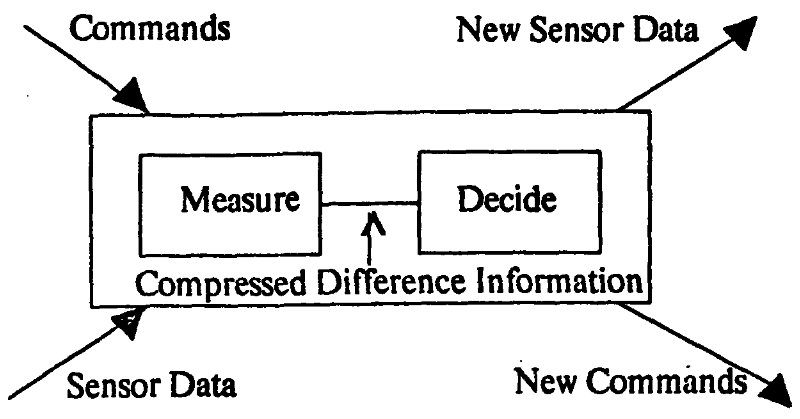

Figure 3: Structure of a Basic Behavior

The behavior system envisaged two levels of mediation, similar to Payton's. The first part selects a set of behaviors that are *a priori* most likely to be useful in the current situation. The second part mediates *a posteriori* between the suggestions they offer. To avoid bottlenecks, the mediation function is assumed to be simple, such as a weighted average or a priority-choose depending on the semantics of the choice-point. The disadvantages of this sort of mediation addressed by Kaelbling are well taken. However, with an emphasis on finding appropriate computation, it is acceptable to go to some effort to remove bottlenecks, including tuning each sub-behavior to the context of other behaviors it must work with. In addition, complicated types of mediation can still be performed by pushing them to the *a priori* selection phase.

The system describes the hierarchy of control, but does not address the hierarchy of construction. Currently, the computational structure is a flat network, with no indications of which pieces fit together to form compound behaviors. The compound behaviors are there; for example, each level of competence in the control system should be viewed as a compound behavior. However, we have not yet built the hierarchical mechanisms to represent this idea of grouping.

As far as actual behaviors are concerned, commands "from above" and sensor data "from below" are interchangeable: both are inputs used in deciding on appropriate actions. In addition, new sensor data and new commands sent on to other behaviors are also interchangeable. The distinction is a helpful convention we use in thinking about control systems but nothing more than that.

4 Issues in Programming Societies of Behaviors

Let us consider the basic picture of behaviors that we have built up. A behavior-agent is a collection of distributed active agents that includes tightly-linked sensory and motor parts, and seems to pursue some goal or task. The behavior must have access to goal and parameter information from other behaviors, world information from perception, and internal status from proprioception. The behavior compares the goal information and internal status to determine what methods for achieving the goal are appropriate. It then generates new goals for lower level behaviors and/or new proprioceptive status information about the internal state.

Behaviors are combined together by several major mechanisms. Mediation is where several independent behaviors propose commensurable responses and a mediator merges them together into a combined response. Goal decomposition is when the actions requested by one agent are actually implemented by manipulating lower-level behaviors. In addition, perceptual behaviors influence other behaviors by reporting perceptual or proprioceptive data to them.

Structural abstractions are useful to group together behaviors that can be treated as a unit. The underlying goal is that real behaviors of the system in the world should be mirrored by internal behavior-agents representing those behaviors. But structural abstractions are somehow arbitrary, and there seems no necessary connection between the abstractions we draw and the behaviors we see.

The rest of this paper considers pieces of this picture to clarify the issues involved. It is divided into sections on the underlying computational models, primitive behaviors, mediation, and abstraction.

4.1 Underlying Computational Models

One of the primary lessons of this paper is that behavior agents are principally a convention for structuring activity in some underlying model of distributed active agents. The behavior mechanism cannot stand on its own as a description of the operation of the system. In particular, the mechanism for behaviors must escape back to the underlying computational medium for many things. Behaviors must be built from primitive sensory, decision, and action units, which must themselves be described by some in some underlying formalism. So a full description of the system must describe the operations of all the components the behaviors. Moreover, the description need not really include the behavior structures at all: once you have described how the component units work together, there is no need to describe how those units are grouped into what we think of as behaviors. The structuring of computing agents into behaviors is not necessary for computation, only for our understanding of the computation.

Programming in societies of behaviors gives little guidance for how to design the underlying computational model. As a structuring mechanism, it makes sense in any computational model that supports the idea of distributed active agents. Guidance in designing the computational model comes not from considering a particular structuring method, but from considering the total requirements on the final system. In particular, we can ask what kind of computational models are appropriate for real-time situated systems such as mobile robots. REX and the Brooks Subsumption Achitecture address this issue in different ways. REX makes strong assumptions on the hardware to make it easy to create systems and reason about them. Brooks makes weak assumptions about the hardware to fuel experimentation on appropriate control structures for simple creatures.

Static topologies of agents and connections seem appropriate for real-time computations, though such systems must pay more attention to modifying the behavior of agents that are in place. The most important advantage of static topologies is that the communications links between all the computing agents can be laid out at compile time. This avoids both the bottleneck of a centralized blackboard and the overhead of checking at run-time for dangling communications links. The use of programs to describe the topology is still allowed, but as compile-time meta-programs rather than as a run-time instantiation mechanism.

Dynamic systems are easier to understand, since many issues of control have been abstracted away into the idea of dynamic instantiation and deinstantiation of behaviors. However, dynamic systems suffer from excess overhead. The full generality of instantiating resources at run-time is simply not needed for most embedded systems. We can assume that the resources and their communication links have been created ahead of time, and isolate precisely those aspects of "turning on" a resource that *must* be done at run-time. Mechanisms can be provided that mimic the important aspects of instantiating a resource, but without the general-purpose overhead.

4.2 Behavior-Agents

The external view of a behavior is that it is the basic unit that can get *turned on* or *exploited* in the control system. Three aspects of turning on a behavior can be identified. First, we need to *activate* the behavior, indicating that its services are required somewhere. Second, we need to *direct* its attention, often by giving it some goal. Third, we must *parameterize* its execution to provide constraints on how it should achieve the goal. For example, we can imagine a simple control system sitting ready to control some actuator. Activating the system corresponds to connecting its outputs to the actuator (by flipping a switch or adjusting a mediator circuit); directing it corresponds to setting the set-point; and parameterizing it corresponds to setting the gains in the feedback circuit. Activating a behavior and setting its goals often occur as the result of the same message or stimulus, but there are many situations where selecting an action and guiding it must come from separate quarters. Studies of animal behavior in particular talk about the need for both a releasing stimulus and a taxis (orientation component) for a behavior.

What about the internal view of a behavior? Difference engines are a specific way of constructing the internal structure of primitive behaviors. There are several reasons why it may be necessary to understand the internal structure of behaviors as well as their external interfaces. In particular, we must understand the internal structure to understand how to modify the control system by evolutionary programming. For example, one way to modify the operation of a control system is to introduce a new way to measure the differences to be reduced in the world, then introduce mediation between the new measuring method and the old. This has the effect of splitting the old behavior into several new behaviors. The lesson is that evolutionary programming does not respect behavior boundaries. If we want to model evolutionary programming, we must have a theory of the internal structure of behaviors. In addition, the view of difference-engines highlights the goal-directed nature of primitive behaviors.

4.3 Mediation

A key reason for the complexity of building real autonomous agents is the need for programming tradeoffs between almost incomparable factors. Many seemingly arbitrary choices must be made that are anything but arbitrary in determining the success of the agent's actions in the world. These tradeoffs share a common core, which is finding a common frame of reference where the suggestions from the different approaches can be compared or combined. The computations are not inherently difficult, but it seems amazingly difficult first to find a suitable common frame of reference and then to get the tradeoffs right. Exploring different methods of mediation requires answers to both a strategy question (what tradeoffs have to be made in each situation) and a mechanism question (what methods of computing the tradeoffs will work well enough in real-time to keep the robot from making fatal mistakes).

A posteriori mediation works well because it takes advantage of parallel computing, to provide a set of available choices at all times. At any instant, the behavior can select the best among the available choices that have already been computed. By contrast, selecting an appropriate behavior and waiting for it to respond will be much slower. Unfortunately, it is impossible to have all your options open at all times. *A priori* selection of strategies is also needed to adjust the set of active strategies as the situation changes. The time-pressures on these selections are much less strict than the time pressures on direct action.

In mediation, there need not always be a winner. Marvin Minsky [Minsky 1987] formulated this idea as the principle of non-compromise, which says that if two ways of satisfying a goal compete strongly, that tends to weaken the goal in its competitions with other goals. For example, in a prolonged conflict between moving left and moving right, the appropriate response may be neither of the above, but a *panic* jump to get out of the situation any way at all.

4.4 Hierarchies and Abstraction

A mechanism for building compound behaviors will have to support the idea of activating all of the relevant pieces of the behavior at the same time. This is similar to Payton's idea of activation-sets: a way to link many units together so that they can be manipulated as one.

Computer science has developed many tools for understanding structural abstractions. The language REX, for example, is an excellent way of describing distributed circuits by using modular meta-programs. However, the functional abstractions for complex control systems are understood very poorly. In order to build complex control systems, we need to understand more clearly what kinds of functional abstractions exist and when to use them.

What makes describing functional abstractions difficult is that the idea of function itself can only be defined with reference to some world context. This idea is familiar to engineers as the truism that design tradeoffs can only be decided with reference to stated requirements. But for a mobile robot, acting in the real world represents the bulk of the requirements; it is precisely the idea of what it means to act in the world that we don't know how to specify yet.

This paper has touched on several important types of functional abstractions that seem important for the domain of flexible mobile robots.

- Behaviors can be exploited by other behaviors by relying on them without explicit interaction, by releasing them to act in appropriate situations, by specifying their goals, or by adjusting their parameters.
- Simple forms of mediation between behaviors for a shared resource can take advantage of parallel processing to pre-compute several promising alternatives for action.
- We need mechanisms to activate groups of agents as a unit, so that combined behaviors can also be exploited by others. This is an open area for research.

4.5 Integrating Perception with Behaviors

Is there a reason to use different computational architectures for exploring the perceptual and behavioral parts of control systems? It is true that early vision and other low-level perceptual processing require data-intensive, data-driven, spatially uniform computations that are very different from the loosely-coupled distributed computations described here. However, even in intermediate processing at the level of Ullman's visual routines [Ullman 1983], the goals of the agent are used to select and guide perceptual processing. Both this sort of Perceptual processing and behaviors are goal-directed yet world-driven; it makes sense to study them together.

5 Conclusions

Evolutionary Programming

The term *evolutionary programming* has been used to describe development of systems by random mutations and natural selection. Our use of the term departs from this in that mutations are not random but purposeful, driven by our (incomplete) expectations of what sort of structures might be appropriate. What both approaches share is the idea of testing the systems against a real world. This methodology is useful whenever the standard of success, i.e. what it takes to survive, is known but how to accomplish this is not. For robust mobile robots, the standard of success is achieving its goals without being trounced by the world. Evolutionary programming helps us find simple mechanisms that work well in the robot's ecological context.

Evolutionary programming is a methodology for creating control systems, not a prescription for what the control systems should look like. In particular, it is orthogonal to the idea of describing the behaviors in the system. Evolutionary systems tend to develop in layers, to implement successive levels of competence. However, the behaviors in these layers can be arbitrarily complex. The behavioral view talks about how the components within a given system work together, and not about how the system was built up over time. The two issues are related but not directly so.

On-line *versus* Off-line

For economic and practical reasons, it is conventional to divide the robot-programming task into two phases: an *off-line* phase where the robot is instructed, taught or programmed, and an *on-line* phase where the robot carries out its instructions without further human intervention. As computing power increases and robot requirements grow more complex, sophisticated functions migrate from the off-line to the on-line phase while maintaining real-time operation [Malcolm and Ambler 1986].

Programming in behaviors is partly an approach to design of on-line control systems that are both purposeful and resilient. For resilience, the key question is how can mediation of multiple strategies be controlled to yield a sensible combined strategy. For purposefulness, the key question is how can higher-levels of the control system manipulate the reflexes of lower levels to carry out goals indirectly. By identifying what sorts of functional abstractions are needed in control systems for complex environments, the idea of programming in behaviors should facilitate development of more complex on-line mobile-robot programs.

This, in turn, advances a second goal, namely developing better off-line environments for mobile-robot programming. The idea is to develop a language for describing robot control systems in which the behavior-agents described by the programmer are closely related to the behaviors exhibited by the system. It remains to be seen how much the idea of programming in behaviors will contribute to this goal.

References

[Arbib 1972] Michael A. Arbib. *The Metaphorical Brain: An introduction to Cybernetics as Artificial Intelligence and Brain Theory.* Wiley-Interscience, New York, 1972.

[Arbib 1981] Michael A. Arbib. Perceptual structures and distributed motor control. In *Handbook of Physiology: The Nervous System, II*, pages 1449–1480, American Physiological Society, Bethesda, MD, 1981.

[Arkin 1987] Ronald C. Arkin. Motor schema based navigation for a mobile robot: an approach to programming by behavior. In *Submitted to 1987 IEEE conference on Robotics and Automation*, 1987.

[Brooks 1986a] Rodney A. Brooks. *Achieving Artificial Intelligence through Building Robots.* AI Memo 899, MIT Artificial Intelligence Laboratory, May 1986.

[Brooks 1986b] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), April 1986.

[Cudhea and Brooks 1986] Peter W. Cudhea and Rodney A. Brooks. Coordinating multiple goals for a mobile robot. In *Intelligent Autonomous Systems*, Elsevier North-Holland, Amsterdam, December 1986. to appear.

[Giralt, *et al* 1984] Georges Giralt, Raja Chatila, and Marc Vaisset. An integrated navigation and motion control system for autonomous multisensory mobile robots. In Brady and Paul, editors, *Robotics Research 1*, pages 191–214, MIT Press, 1984.

[Kaelbling 1986a] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. April 1986. Unpublished note, SRI International and Stanford University.

[Kaelbling 1986b] Leslie Pack Kaelbling. *REX Programmers Manual.* Technical Note 323, Artificial Intelligence Center, SRI International, 1986.

[Lyons 1986] Damian Lyons. *RS: A Formal Model of Distributed Computation for Sensory-Based Robot Control.* COINS Technical Report 86-43, University of Massachusetts at Amherst, September 1986.

[Malcolm and Ambler 1986] C. A. Malcolm and A. P. Ambler. Some architectural implications of the use of sensors. In *Intelligent Autonomous Systems*, Elsevier North-Holland, Amsterdam, December 1986. to appear.

[Minsky 1987] Marvin Minsky. *The Society of Mind.* Simon and Schuster, New York, 1987.

[Moravec 1983] Hans P. Moravec. The Stanford cart and the CMU rover. *Proceedings of the IEEE*, 71:872–884, July 1983.

[Nilsson 1984] Nils J. Nilsson. *Shakey the Robot.* Technical Note 323, SRI AI Center, April 1984.

[Payton 1986] David W. Payton. An architecture for reflexive autonomous vehicle control. In *IEEE Robotics and Automation Conference*, San Francisco, April 1986.

[Rosenschein and Kaelbling 1986] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In *Proceedings of a Workshop on Planning and Sensing for Autonomous Navigation*, Oak Ridge National Laboratory, January 1986. Published as technical report ORNL/TM-9923/CESAR-86/01.

[Ullman 1983] Shimon Ullman. *Visual Routines.* AI Memo 723, MIT Artificial Intelligence Laboratory, June 1983.