SRI International

AN ARCHITECTURE FOR INTELLIGENT REACTIVE SYSTEMS

Technical Note 400

October 8, 1986

By: Leslie Pack Kaelbling
Artificial Intelligence Center
Computer and Information Sciences Division
and
Center for the Study of Language and Information
Stanford University

**APPROVED FOR PUBLIC RELEASE:
DISTRIBUTION UNLIMITED**

This work was supported in part by a gift from the Systems Development Foundation, in part by FMC Corporation under contract 147466 (SRI Project 7390), and in part by General Motors Research Laboratories under contract 50-13 (SRI Project 8662).

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (415) 326-6200 • FAX: (415) 326-5512 • Telex: 334486

Abstract

Any intelligent system that operates in a moderately complex or unpredictable environment must be *reactive* — that is, it must respond dynamically to changes in its environment. A robot that blindly follows a program or plan without verifying that its operations are having their intended effects is not reactive. For simple tasks in carefully engineered domains, non-reactive behavior is acceptable; for more intelligent agents in unconstrained domains, it is not.

This paper presents the outline of an architecture for intelligent reactive systems. Much of the discussion will relate to the problem of designing an autonomous mobile robot, but the ideas are independent of the particular system. The architecture is motivated by the desires for modularity, awareness, and robustness.

Any intelligent system that operates in a moderately complex or unpredictable environment must be *reactive* — that is, it must respond dynamically to changes in its environment. A robot that blindly follows a program or plan without verifying that its operations are having their intended effects is not reactive. For simple tasks in carefully engineered domains, non-reactive behavior is acceptable; for more intelligent agents in unconstrained domains, it is not.

This paper presents the outline of an architecture for intelligent reactive systems. Much of the discussion will relate to the problem of designing an autonomous mobile robot, but the ideas are independent of the particular system. The architecture is motivated by three main desiderata:

Modularity: The system should be built incrementally from small components that are easy to implement and understand.

Awareness: At no time should the system be unaware of what is happening; it should always be able to react to unexepected sensory data.

Robustness: The system should continue to behave plausibly in novel situations and when some of its sensors are inoperative or impaired.

Modularity

It is well-established principle of software engineering that the modular design of programs improves modifiability, understandability and reliability [4]. The ability to combine simple behaviors in different ways will facilitate experimentation in the design of a complex, reactive system.

Brooks [5] has proposed a *horizontal* decomposition of a robot's control system, in which the fundamental units of the program are task accomplishing behaviors. Each behavior consists of both an action and a perception component and may, in a structured manner, depend on other behaviors in the system. This is in contrast with the standard approach, which he refers to as *vertical* decomposition — namely, a division into many subsystems, each of which is essential for even the most elementary behavior. Such a vertical decomposition might include the following components: perception, modeling, planning, execution, and effector control.

Horizontal decomposition is attractive because the system can be built and debugged incrementally, allowing the programmer to test simple behaviors, then build more complex ones on top of them. There are some difficulties involved in having perception distributed throughout multiple behavior components, however. The first is that special-purpose perception mechanisms tend to be weak. Raw sensory data is often noisy and open to a variety of interpretations; to make perception robust, it is necessary to exploit the redundancy of different sensor systems and integrate the information from many sources. The second difficulty stems from the fact that, as behaviors become more sophisticated, they tend to be dependent on conditions in the world, rather than on the particular properties of sensor readings. A general perception mechanism can synthesize information from different sensors into information about the world, which can then be used by many behaviors.

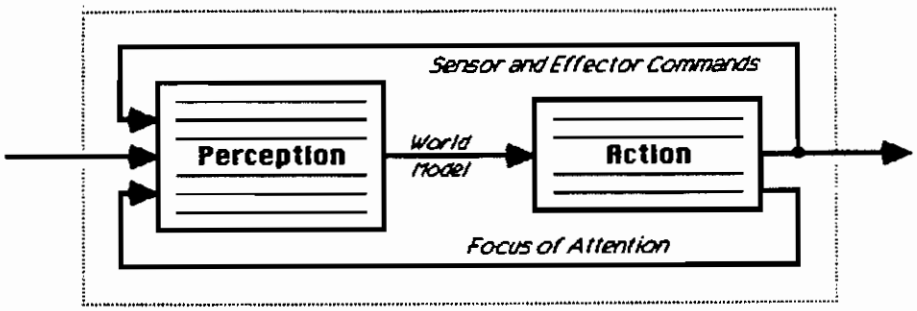

Figure 1: Top-Level Decomposition

We propose a hybrid architecture with one major vertical division between the perception component and the action component. There is to be a horizontal decomposition within each of these components, but any of the action subcomponents may take advantage of any of the perception component's outputs. This component will be decomposed into layers of abstraction, with uninterpreted sensor readings available at the lowest level and sophisticated world models available at the highest level. The action component will consist of a set of behaviors, each of which may undergo some further structural decomposition.

Figure 1 contains a block diagram of this architecture. The system receives raw data from the sensors, and emits commands to the sensors and effectors. The action component takes the output of the perception component as input and, as in the preceding case, generates commands to both the sensors and effectors. This differs from many other systems, in which control of the sensors is the responsibility of the perception component. In many situations the sensors are a scarce resource; consequently, decisions must be made about where to point the camera or which ultrasonic sensor to fire at a given time. What should be done in such cases depends critically on the action strategy that is being followed at the moment: Is the robot following a wall on the left? Is it trying to locate an object in front of it? Since the action component is deciding on the strategy of the effectors, it is in the best position to do so for the sensors as well. For the same reasons, if the perception component is limited in the amount of processing it can do, the action component generates an attention command to the perception component, indicating where its computing power should be directed. It might focus attention on a particular region of the visual field, or on a certain kind of object. The entire sensory data stream goes directly into the perception component, along with the both the attention command of the action component and the commands that were last sent to the effectors.

Awareness

For a robot to be truly aware of its environment, it must be designed in such a way that there is a constant bound on the interval between the time the sensors get a particular reading

and the time the effectors can react to that information. Many robots simply "close their eyes" while a time-consuming system, such as a planner or vision system, is invoked; the penalty for such unawareness is that perceptual inputs are either lost or stacked up for later processing. During this period of dormancy, a truly dynamic world might change to such an extent that the results of the long calculation would no longer be useful. Worse than that, something might happen that requires immediate action on the part of the robot, but the robot would be oblivious of it.

Our approach to this problem is to have a number of processes that work at different rates. We define a tick to be the constant minimum-cycle time for the entire system. During each tick, the inputs are read, some computation is done, and the outputs are set. If a process cannot complete its computation during its portion of a cycle, either because its runtime is inherently non-constant, or has a large constant, it emits a signal indicating that its outputs are not yet available, whereupon its state is saved for resumed execution during the next tick. The Rex language, which is discussed below, allows such a system to be easily constructed.

Robustness

Once again we propose a solution similar to that of Brooks [5]. His system is broken down into *levels of competence* in such a way that, if higher levels break down, the lower levels will still continue to work acceptably. This is especially important for Brooks, since his levels are intended to be built on separate physical devices that can fail independently. Our system, on the other hand, will be implemented on a single piece of hardware, so we will concern ourselves with robustness only in relation to failed sensors or to the possibility of general confusion because of new or unusual situations. We shall refer to these two types of robustness as *perceptual* and *behavioral*.

Perceptual robustness can be achieved by integrating all sensory information into a structure that represents the robot's knowledge or lack of knowledge about the world. If a particular sensor fails and its failure has been detected, the robot's information about the world will be weaker than it would have been if all of the sensors had been working correctly. We say that the information I, carried by an agent is weaker than information I' if and only if the set of possible worlds compatible with I is a superset of the set of possible worlds compatible with I'. Thus, with weaker information, the robot can make fewer discriminations among the states of the world, but it is can still be the case that that information integrated from the remaining sensors will suffice for reasonable but degraded operation. If a particular behavior depended entirely on a single sensor, there would be no room for graceful degradation; it would simply fail. The problem of detecting sensor failure is a difficult one that we shall not be examining here. Eventually, however, work in fault detection mechanisms will have to be integrated into such a system.

Behavioral robustness depends upon the ability to trigger a system's actions in direct accordance with the strength of available information. Consider a behaviorally robust robot with a high-level path-planning module that generates actions based on a strong model of the environment. If the robot's actual information is insufficient for the path planner to

produce a plan — perhaps because the robot had just been switched on or had become lost or confused, that module will simply emit a signal indicating its inability to form a plan. In that case, some less sophisticated module that is capable of operating with weaker information will know what to do; its actions might be directed toward gaining sufficient information to enable the first module to work and avoiding coming to any harm in the process. Another example of behavioral robustness concerns the robot's behavior in case any of the necessary action-computing processes, such as planners or visual matching systems, cannot run in real time. The high-level planning module may not know what to do for several ticks until it has finished computing its plan; during this time, however, lower-level, less competent action modules should be in control, attempting to maintain the status quo and to keep the robot out of danger.

Building Real-time Systems

Rex

Rex is a language designed for the implementation of real-time embedded systems with analyzable information properties [12,8]. It is similar to a hardware description language in that the user declaratively specifies the behavior of a synchronous digital machine. Johnson [7], exploring the idea of using purely functional notation and recursion equations for circuit description, found that it was indeed viable and, moreover, in many ways preferable to standard techniques. He presents techniques for synthesizing digital designs manually from recursion equations. In Rex, the programmer can use both recursion and functional style, as well as having the specifications be translated automatically into hardware descriptions. There are, however, many complex recursion equations that are not automatically translatable into Rex. The declarative nature of Rex makes programs amenable to analysis of semantic and behavioral properties. From a Rex specification, the compiler generates a low-level structural description that can then be simulated by sequential code in C or Lisp.

The resulting machine description can be visualized as a large collection of integer variables and code that updates them once per tick. The variables can be divided into *input*, *state*, and *output*. The input variables, conceptually connected directly to the sensors, contain current sensory values at the beginning of each tick. The state variables are updated during each tick as a combined function of the values of the input variables and the old values of the state variables. The output variables, conceptually connected to the effectors, are updated during each tick as a combined function of the inputs and the old values of the state variables. Rex can be thought of as specifying a function $F : I^{i+s} \rightarrow I^{s+o}$, where i,s, and o are the numbers of input, state, and output variables, respectively, that maps the values of the inputs and the old values of the state variables into new values of the state variables and outputs. For any machine specified in Rex, the function F is guaranteed to be calculable in constant time. This in turn guarantees that the minimum reaction time (minimum time required for the value of an input to affect the value of an output) also has a constant bound, therey making all machines defined in Rex real-time.

Embedding Slow Processes in Fast Systems

As control systems become more sophisticated, they almost always involve planning of some sort. David Chapman has shown that a general planning problem is undecidable and that many restricted planning problems are intractable [6]; we must therefore consider methods for embedding processes that do not operate in constant time in systems with a constant tick rate. The intractability of planning, as well as other time-consuming problems, usually stems from the need for graph search. There are two methods for implementing search procedures in real-time systems. The first is to exploit the power of parallel processing and devote a large amount of dedicated hardware to doing the search in constant time. The second method is to conserve hardware and to search by using a conventional algorithm, such as backtracking, but to guarantee that the searching process will be "swapped out" in such a way that other processes are assured a chance to react to inputs in real time.

Production systems are often used to perform search and inference in problem solving and planning systems. In many of these systems, the rules are fixed and cannot be changed during execution. If this is the case, an inference net [13] can be explicitly implemented in hardware, allowing all search and inference to take place in parallel in constant time that is proportional to the maximum depth of the net. For many problems, the inference net will require less space than would have been needed to encode a rule interpreter and the production rules implicitly embodied by the net.

In general, any computation can trade time for space. Thus, if sufficient computing hardware is not available to implement large searching processes in parallel, they may be serialized and run on general-purpose hardware. Von Neumann computer architecture is an extreme example of this; it allows huge programs to be run on a very small amount of hardware, trading time for space. We propose a middle ground for embedding processes like planners into real-time systems, using general-purpose searching hardware for the processes that involve search, and iterating the searching process over time, while the other processes continue to run in parallel with it.

Planning

There are two problems that arise when a planner is run in a dynamic environment. The first is that, if the planner takes control of the processor, the robot can no longer respond, even at a reflex level, to events in the environment. The second problem is that, during the process of planning, the environment may have changed to such an extent that the newly created plan is no longer executable in the current situation.

A solution to the first problem is for the planner to work incrementally, doing a few computation steps during each state transition, then storing its state until the next tick. Other parts of the system that react more quickly to changes in the environment will be running in parallel with the planner, and will therefore be able to act even if the planner has not finished its computation. This behavior is in contrast to that of a program that "calls" the planner and waits for it to finish executing before doing anything else. When the planner is finished, it issues the plan; until that time, it emits a signal that says it is not

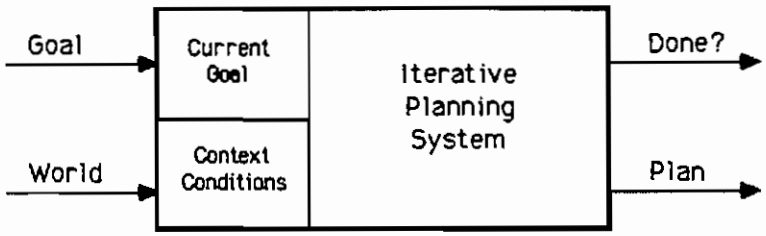

Figure 2: Schematic of an Embedded Planner

ready yet and has no answer. The specification of a planner that works incrementally and saves its state is written easily in Rex. A similar system might also be constructed using an operating system with message passing and a round-robin scheduler. This would make it possible for other processes to respond to external events while the planner is working, although the informational analysis of the Rex version would be much more tractable.

A planner is typically given a description of some initial state and a goal, and then activated. The planner constructs a plan that depends on the truth of some of the conditions in the initial state; we shall call these *context conditions*. The rest of the initial state is either irrelevant to the plan (for instance, the temperature is irrelevant to planning to go down the hall), or can be handled conditionally during plan execution (the robot might assume that it can navigate around local obstacles without planning). A planner embedded in a real-time system must be especially conscious of its context conditions; otherwise it cannot know whether the plan it is working on will be valid when it is done.

In Figure we present a schematic diagram of a planner that works flexibly in dynamic environments. Its inputs are a goal and the output of the world model; its outputs are a plan and a signal as to whether or not the plan is ready. When it is given a new goal, it remembers that goal and the current values of the context conditions in its local state. It begins planning with respect to those values of the goal and context conditions until the plan has been completed or until the goal or context conditions in the world differ from those that are stored in the planner.

If the goal or context conditions change before the completion of the plan, the planner stores their new values and begins planning again. This scheme has the property that the planner will notice at the earliest instant if its plan is no longer valid because of a change in goal or context conditions, and will therefore start working on a new one. The planner might be made more efficient if, when the goal or context is changed, it tried to salvage parts of the plan in progress. It is true that, if the context conditions or goal vary too rapidly, the planner will never succeed in generating a plan. This would happen only if the planner were not written with adequate generality for the environment in which it is embedded. One way to simplify the design of embedded planners, as well as to make the planning

process more efficient, is to use many small planners that are domain-dependent, rather than one large, general, domain-independent planner. Much of the domain knowledge can be "procedurally represented" in a domain-dependent planner, eliminating the need for its runtime manipulation.

The Perception Component

As in the domain of actions, perception can be done at many levels of abstraction. Normally, the higher the level of abstraction, the more processing power is required to integrate new information. Thus, we will break the perception component of this architecture down into several levels of abstraction that can be made to work at different speeds, using the techniques that were applied to the planner in the preceding section. At the lowest level, we might simply store the most recent raw perceptual readings. Since this level requires no interpretation or integration, the data are immediately available to highly time-critical behavior components, such as obstacle-avoidance reflexes. More advanced behaviors will require information that is more robust and abstract. Eventually this will culminate in a representation that integrates data from all of the sensors into a coherent world model. The world model itself might exist at various levels of abstraction, from Cartesian locations of obstacles, to a topological map of interconnections of hallways, doors, and rooms.

It is important to note that these levels of perception may have no direct mapping to the levels of competence in the action component. The highest level of action competence will consist of behaviors at many different levels of abstraction; it thus relies on many or all of the layers of the perception component. It is likely however, that the lower action levels will not make use of the higher perception levels, thereby allowing each of the majore components to be constructed incrementally.

For a system to be behaviorally robust, the representation of perceptual data must explicitly encode the robot's knowledge and lack of knowledge about the world. If we consider the propositional case, the robot can stand stand in three relations to a proposition φ: it can know that φ holds ($K(\varphi)$), it can know that φ doesn't hold ($K(\neg\varphi)$), or it can be unaware as to whether φ holds ($\neg K(\varphi) \wedge \neg K(\neg\varphi)$). If φ were the proposition "I'm about to run into the wall," we might have the following set of action rules:

$$\begin{aligned} K(\varphi) &\rightarrow \mathit{stop} \\ K(\neg\varphi) &\rightarrow \mathit{go} \\ \neg K(\varphi) \wedge \neg K(\neg\varphi) &\rightarrow \mathit{stop} \wedge \mathit{look\_for\_wall} \end{aligned}$$

These rules do something reasonable in each case of the robot's knowledge, or lack thereof, guaranteeing that it won't hit the wall but will go forward if it knows that such a collision is not imminent. It also tries to strengthen its information in case of uncertainty. For many applications, this approach may have to be extended to the probabilistic case, substituting $P(\varphi) > a \rightarrow \alpha$ for $K(\varphi) \rightarrow \alpha$, where a is the necessary degree of belief in the proposition φ to make α an appropriate action. We would similarly substutute $P(\varphi) < b$ for $\neg K(\varphi)$ and $b \leq P(\varphi) \leq a$ for $\neg K(\varphi) \wedge \neg K(\neg\varphi)$.

Framework for Adaptive Hierarchical Control

In this section we present a scheme for the hierarchical decomposition of robot control in terms of compositions of behaviors. We define a *behavior* to be a procedure that maps a set of inputs, which in this case are the outputs of the perception module, into a set of outputs to the effectors of the system. Each behavior is has the same input/output structure as the action module in Figure 1, with possibly some additional outputs that are intended to be used internally. To compose behaviors, we use procedures called *mediators*. A mediator's inputs are outputs of several *subbehaviors* and the the perception module. Since it generates outputs of the same type as a behavior, the complex module consisting of the subbehaviors and a mediator is itself a behavior.

Mediating Behaviors

One scheme for mediation between subbehaviors, described by Kurt Konolige [9], is a "bidding" system in which each behavior outputs in the form of sensor effector commands not only what it wants to do, but also some measure of its "desire" to do it. The mediator decides what to do on the basis of some weighted average of the the outputs of the subbehaviors and their respective degrees of urgency. There are two possible difficulties with such a scheme. One is that, when the behaviors are at a higher level than simple motor control, the mediation will have to be more than a simple average; for example, a robot performing the average of going to office A and office B probably won't get far. A logical response to this difficulty is that the two behaviors (the office A behavior and the office B behavior) would have to know something about each other, and so would only request actions that are compatible. But this seems to require mediation again, albeit internal to the behaviors, and it brings us to the second difficulty. One of the greatest advantages of a compositional methodology is that a particular component can be independently designed and tested, then used in more than one place in a system. In Konolige's approach there is something crucially context dependent about each low-level behavior, since its urgency parameters will have to be tuned for each specific application, depending on what other behaviors it is being combined with.

One approach that appears to overcome these difficulties is to move all the intelligence governing behavior selection into the mediator function itself. In this scheme, the mediator would take the outputs of the subbehaviors, as well as the world model and other perceptual data, as inputs. Then, on the basis of these data, the mediator could output some weighted combination of the input behaviors or, alternatively, simply switch through the output of a particular behavior. If there are very different effectors, it might make sense to perform part of one behavior and part of another; for example, a walking and a talking behavior could be mediated by outputting the speech commands of the talker and the motor commands of the walker. Each behavior can be designed and debugged independently, then used without modification as a building block for other, more complex behaviors. Another advantage of this approach is that proofs of correctness of complex behaviors can be done compositionally. A proof involving a complex behavior need only involve the switching behavior of the

mediator and those properties of the subbehaviors that can be proved independently.

Hierarchically Mediated Behaviors

We will approach the the design of a robot's action component as a top-down decomposition of behaviors into lower-level behaviors and mediators. At the top level, adopting the scheme of Brooks, we have a number of behaviors that represent different levels of competence at executing the main task of the system. Each behaviors, unlike those of Brooks, computes its outputs independently of the outputs of the other modules. Included in the set of possible outputs of each behavior is `no-command`, a signal denoting that that behavior doesn't know what to do in the current situation. We have some intuitive idea of what competence is, and, given two modules, can make subjective judgements about which works "better." We hope to formalize the notion of what makes one action or strategy better than another with respect to some goal; since that has not yet been done, however, the balance of this discussion must be based on our intuitive understanding of "better." If the following four properties hold of a system whose top-level mediation function switches through the entire output of the most competent behavior that knows what to do, the system as a whole will always do the best thing of which it is capable, given the available information.

- The lowest level of competence never outputs `no-command`
- No level emits a command other than `no-command` unless it is a correct command
- Lower levels of competence require weaker information
- If any two levels both emit commands in the same tick, the output of the higher level is better

Thus, if the more competent levels fail or have insufficient information to act, the robot will be controlled by a less competent level that can work with weak information until the more competent components recover and resume control.

Within each of the levels of competence, decomposition is based on abstraction rather than competence. The highest-level behavior is constructed by mediating among medium-level behaviors. Those behaviors are constructed by mediating among low-level behaviors. The structure will typically be a graph rather than a tree, since many high-level behaviors will ultimately be constructed from a few low-level ones. In practice, it will also happen occasionally that the hierarchy will not be strict; that is, a certain behavior might be present at two different levels in the graph.

There has been other work exploring the use of a hierarchy of abstraction for reactive control. James Albus [1,2,3] in the RCS (Real-time Control System), employs an abstraction hierarchy of "multivariant servos" for controlling factory automation systems. Although his approach is similar to ours, it does not allow the simultaneous combining of components of more than one behavior, even if they are potentially compatible. This is equivalent to having the mediating functions always switch through one entire behavior. Nils Nilsson has proposed using triangle tables as a robot programming language[11]. They were originally used in the SRI robot *Shakey* [10] for plan execution monitoring, but the formalism can be extended to hierarchical systems that are very much like the one described by Albus.

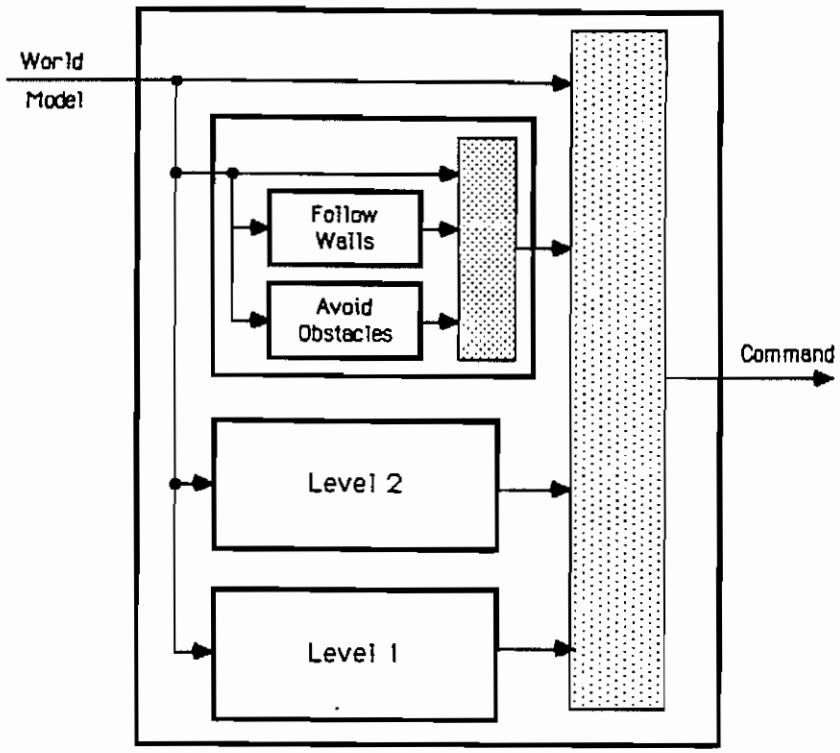

Figure 3: Example of Hierarchies of Competence and Abstraction. (The shaded boxes are mediators).

Example

We shall now present an example that illustrates the methods of hierarchical decomposition discussed in the preceding section. After sketching the top-level decomposition of a complex behavior into levels of competence, we will show how the most competent module is broken down into levels of abstraction. A block diagram of this example is presented in Figure 3.

The task of this robot is to traverse a very long hall without crashing into anything. It is more important to avoid crashes than to get to the end of the hall. The robot's construction is such as to make it highly unlikely that it can roll straight down the hall without veering into the sides unless it corrects its course along the way. The robot has distance sensors pointing forward and to each side. We decompose this problem into three behaviors at different levels of competence, as follows:

Level 1 This behavior looks at accumulated raw sensor data. If any of the measurements in taken from the front of the robot too short, or a significant interval has elapsed since the last measurement was made by the front sensor, it stops; otherwise it moves forward.

Level 2 This behavior also looks at accumulated raw sensor data. As in the preceding

behavior, it stops if the measurements are too short or too old. If it has stopped and cannot move forward, but the sensor data imply that it is safe to turn, the robot turns until the sensor data are no longer too short, then moves. If it isn't save to turn, it emits `no-command`.

Level 3 This behavior looks at data that has been combined at a higher level of abstraction. It can tell whether there is a wall to the front or side, how far away it is, and how tight the bounds on its knowledge of its position are. If it knows that there is no wall too close to it,[1] and knows fairly tight bounds on the locations of the walls on either side, it moves in such a way as to go forward in the middle of the hall, staying parallel to the walls. If it doesn't know this, it emits `no-command`.

This set of behaviors satisfies the rules given above for a correct decomposition. The lowest level always either moves or stops. Each level acts only when it knows the particular action is safe — that is, when executing it will not cause the robot to crash into something. The lowest level requires only sensor readings, which, although weak, are available instantly. The second level requires information about whether it is safe to turn; this information, stronger than that needed by the first level, must be synthesized from the raw sensor readings. The highest level requires very strong wall-location data that must be derived from the aggregation of many sensor readings and knowledge about the world. If each level knows what to do, it is intuitively obvious that the highest-level behavior is "best." It is better to proceed along a hall by staying parallel to the walls than by zig-zagging from side to side (which is what the second behavior is likely to do), or just by going to one side of the hall and stopping when obstructed.

The highest level of competence can be divided into subbehaviors at different levels of abstraction, as shown in Figure 3. The first division is into a behavior that stays parallel to the walls on the sides and one that causes the robot to slow down linearly as a function of its distance to an obstacle in front of it. Each of these behaviors is composed of subbehaviors that cause the robot to move at certain velocities and request certain sensor measurements. Let us assume that each behavior consists of a motor command and sensor command (the robot can poll only one sensor at a time).

Then, in pseudocode, the `follow-walls` behavior is

```
sensor-command := if left-info-weak then left-sensor
                  else if right-info-weak then right-sensor
                  else *noop*
motor-command  := if K-location-of-left-wall and
                     K-location-of-right-wall then servo-to-midline
                  else *no-command*
```

This behavior requests a sensor measurement if it has weak information about one side or the other, and returns `*noop*` if it has no immediate need for sensor information. If it knows the location of the left and right walls to a close enough tolerance, it performs the behavior that servos to the middle line of the hallway; otherwise, it emits `*no-command*`, indicating that it does not know what to do.

[1] A wall is too close to the robot if it will crash into the wall unless it begins its stopping action immediately.

The no-crash behavior is described by

```
sensor-command := if front-info-weak then front-sensor
                  else *noop*
motor-command := if K-location-of-front-obstacle then linear-speed-limit
                 else *no-command*
```

This behavior requests a sensor measurement from the front sensor if it needs it and, if it knows the location of the nearest obstacle in front to sufficient tolerance, it performs the behavior that causes the robot slow down in proportion its distance from the obstacle. If it does not know that location, we emit *no-command*.

Now it remains only to combine these two behaviors. The mediator is

```
sensor-command := if no-crash-sensor-command = *noop*
                  then follow-wall-sensor-command
                  else no-crash-sensor-command
motor-command  := if (follow-wall-motor-command = *no-command*) or
                     (no-crash-motor-command = *no-command*)
                  then *no-command*
                  else rescale (follow-wall-motor-command, no-crash-motor-command)
```

If the no-crash behavior does not request a sensor command, the mediator does what the follow-wall behavior wants to do; otherwise it does what the no-crash behavior wants to do. This gives priority to acquiring information that is relevant to the more important goal of avoiding obstacles. If either motor command is *no-command*, the motor command of the mediator will be the same. If both motor commands are defined, the wall-following motor command defines a set of differential velocities for maintaining the heading of the robot down the center of the hall and the crash-avoidance motor command defines a limit for safe speed, given the knowledge of the distance to obstacles in front of the robot. These values are input to the function rescale which performs a ratiometric scaling of the servo velocities. This is done so that neither velocity will them exceed the speed limit, but their ratio will maintained.

Future Work

This methodology has been applied to simple tasks, such as the one described above, with a large degree of success. As well as expanding the implemented example, we will continue research on the formal specification of goals and the ranking of the "goodness" of behaviors with respect to particular sets of goals. The problem of perceptual organization also requires more attention, with the aim of devising algorithms that use predictions about the environment from old information to facilitate analysis of new information.

Acknowledgments

This research was done in the context of programming the SRI Artificial Intelligence Center's mobile robot to perform hallway navigation tasks. Many of these ideas arose from discussions with Stan Rosenschein, and debugging sessions with Stan Reifel and Sandy Wells.

References

[1] Albus, James S., 1981: *Brains, Behavior, and Robotics* (BYTE Books, Subsidiary of McGraw-Hill, Peterborough, New Hampshire).

[2] Albus, James S., Anthony J. Barbera, and Roger N. Nagel, 1981: "Theory and Practice of Hierarchical Control," *Proc. 23rd IEEE Computer Society International Conference* (September).

[3] Barbera, Anthony J., M. L. Fitzgerald, James S. Albus, and Leonard S. Haynes, 1984: "RCS: The NBS Real-time Control System," *Proc. Robots 8 Conference and Exposition, Detroit, Michigan* (June).

[4] Booch, Grady, 1983: *Software Engineering with Ada* (The Benjamin/Cummings Publishing Company, Menlo Park, California).

[5] Brooks, Rodney, A., 1985: *A Robust Layered Control System for a Mobile Robot* (A. I. Memo 864, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts).

[6] Chapman, David, 1985: *Planning for Conjunctive Goals* (Technical Report 802, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts).

[7] Johnson, Steven D., 1983: *Synthesis of Digital Designs from Recursion Equations* (The MIT Press, Cambridge, Massachusetts).

[8] Kaelbling, Leslie Pack, 1986: *Rex Programmer's Manual* (Technical Note 381, Artificial Intelligence Center, SRI International, Menlo Park, California).

[9] Konolige, Kurt, 1986: personal communication.

[10] Nilsson, Nils J., 1984: *Shakey the Robot* (Technical Note 323, Artificial Intelligence Center, SRI International, Menlo Park, California).

[11] Nilsson, Nils J., 1985: *Triangle Tables: A Proposal For A Robot Programming Language* (Technical Note 347, Artificial Intelligence Center, SRI International, Menlo Park, California).

[12] Rosenschein, Stanley J. and Leslie Pack Kaelbling, 1986: "The Synthesis of Digital Machines with Provable Epistemic Properties," *Proc. Conference on Theoretical Aspects of Reasoning About Knowledge, Asilomar, California*, pp. 83-98.

[13] Winston, Patrick Henry, 1984: *Artificial Intelligence*, second edition (Addison Wesley, Reading, Massachusetts), pp. 181-187.