Robot Architectures

What is it that distinguishes the software for a reasonably sophisticated robot from most other large and complicated software systems? The answer has to do with its embededness of the robot and the demands of responding to the environment in a timely manner. The relationship between the computational requirements for coming up with an an appropriate response to a given environmental challenge and the time allowed by the circumstances is at the heart of designing robot architectures. In many cases this issue is finessed simply by having robots that have enough computational resources that they don't have to worry about being clever.

Consider the task of driving down an interstate highway. There are the small adjustments you make to stay within your lane. There are larger and more abrupt adjustments you might make to avoid a piece of tire tread or other road hazard. You might plan your trip well in advance to determine which sequence of roads you'll take to get to your desired destination. You'll have to divide your attention between staying in your lane and watching the cars around you and watching for signs and landmarks that tell you of approaching exits. Once you see the sign for your exit, you may have to plan how to maneuver your vehicle across several lanes of traffic in order to make your exit.

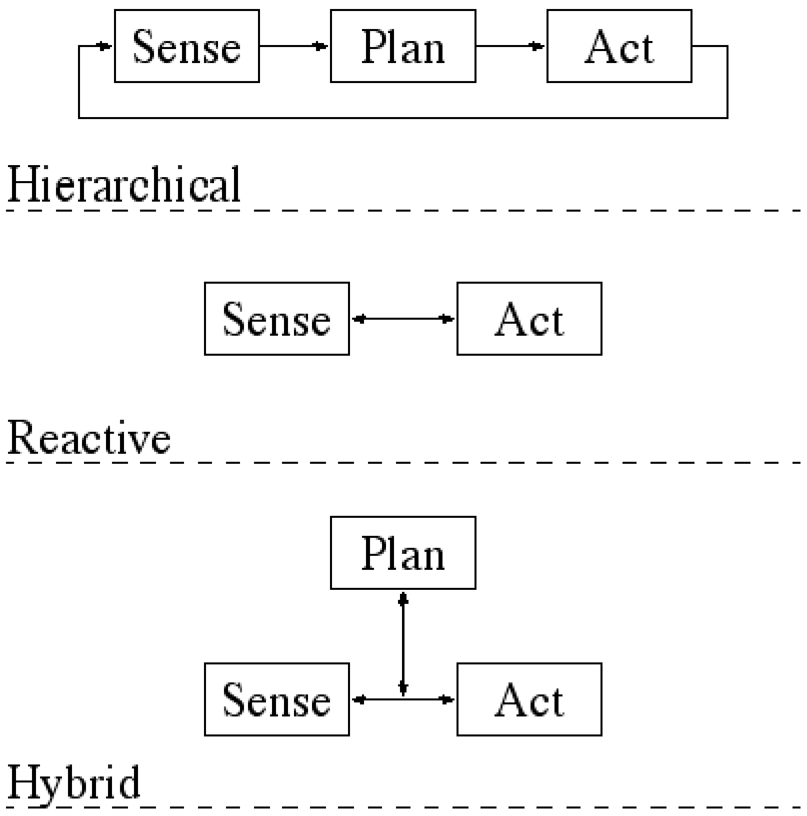
Planning a route could be as difficult as solving a traveling salesman problem or as easy as finding the shortest path in a graph. Certainly thinking about how to maneuver across four lanes of traffic could take longer than figuring out how to swerve to miss a pot hole. What do you do if you're hurtling toward an exit but not sure if it is the best exit to take in getting to your destination? You can't simply stop the world while you figure things out. You can't even focus your attention entirely on the problem because you still have to attend to the road.

There is another issue that often comes to the fore and has its analog in conventional desktop systems and that is the management of resources. Just as two different processes can't be allowed to access a disk drive at the same time, two processes (or *behaviors*) can't be allowed to drive the motors at the same time. Suppose your maneuvering across the highway trying to reach the far right lane to turn onto an approaching exit. At some level all of your attention is on getting the car to move to the right. Then suddenly you notice a car appear on your right and another part of your brain takes control of the wheel and swerves to the left to avoid a collision. How to arbitrate between different goals and behaviors each requiring access to a critical resource?

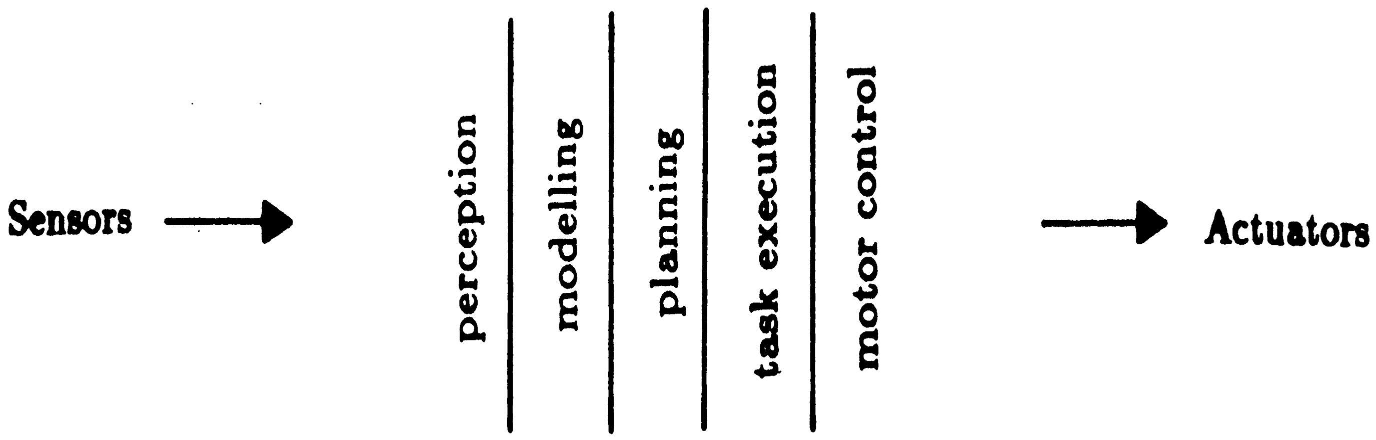
What sort of architecture might allow for timely responses across a wide spectrum of environmental challenges and at the same time provide a framework for arbitrating among competing behaviors?

Look and Lurch

Murphy [2000] describes the range of current architectures (or *paradigms*) in terms of the relationships between three primitives, *sense*, *plan* and *act* and in terms of how sensory data is processed and propagated through the system. The following graphic illustrates the relationships among the primitives in terms of the three dominant paradigms.



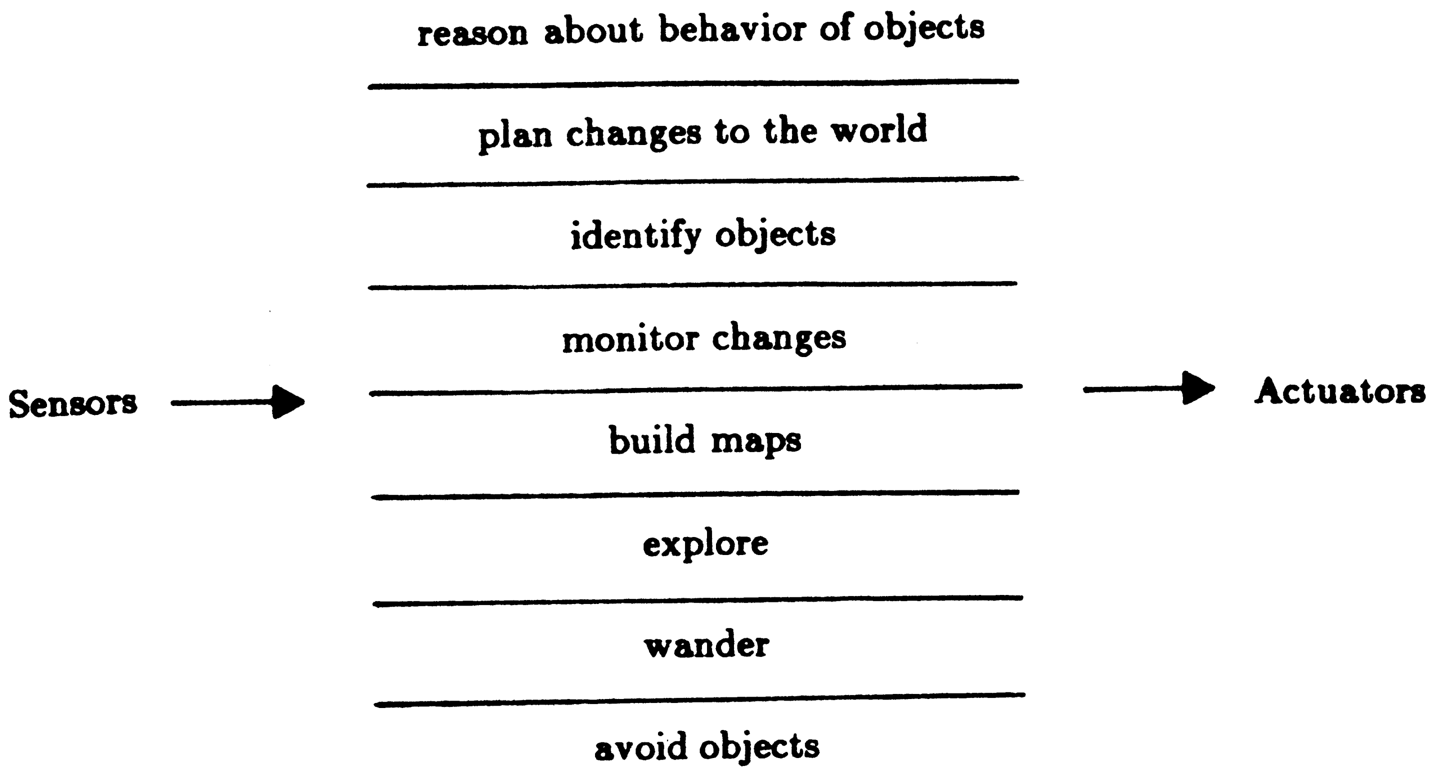
The *hierarchical paradigm* is a bit of a caricature. It was however the dominant paradigm in the early days of AI robotics when much of the focus was on robot planning. The emphasis in these early systems was in constructing a detailed world model and then carefully planning out what steps to take next. The problem was that, while the robot was constructing its model and deliberating about what to do next, the world was likely to change. So these robots exhibited the odd behavior that they would look (acquire data, often in the form of one or more camera images), process and plan, and then (often after a considerable delay) they would lurch into action for a couple of steps before beginning the cycle all over again. Shakey a robot developed at the Stanford Research Institute in the 1970s was largely controlled by a remote computer connected to the robot by a radio link; Shakey exhibited this sort of look-and-lurch behavior as it contemplated moving blocks around to achieve a goal. The characteristic aspects of this paradigm are illustrated by the following figure from [Brooks, 1986]:



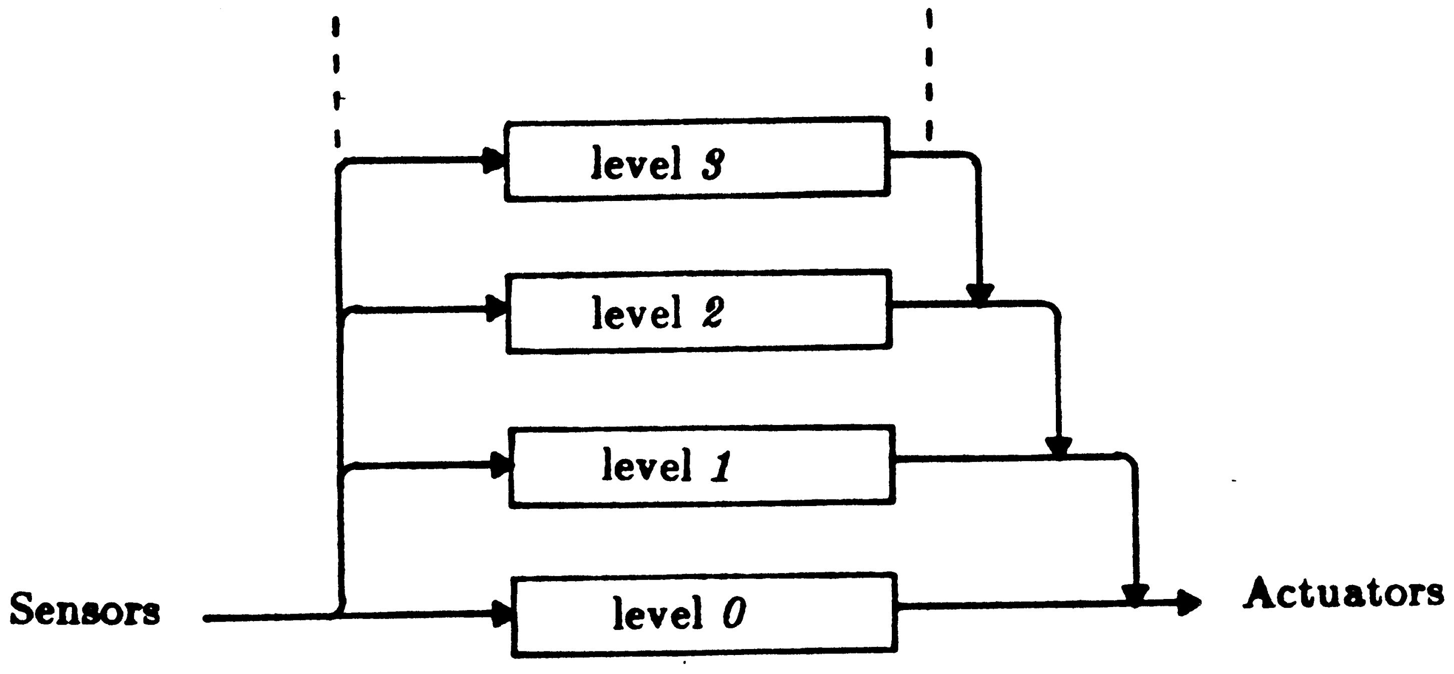
The components of the robot in this case are said to be horizontally organized. Information from the world in the form of sensor data has to filter through several intermediate stages of interpretation before finally becoming available for a response.

Reactive Systems

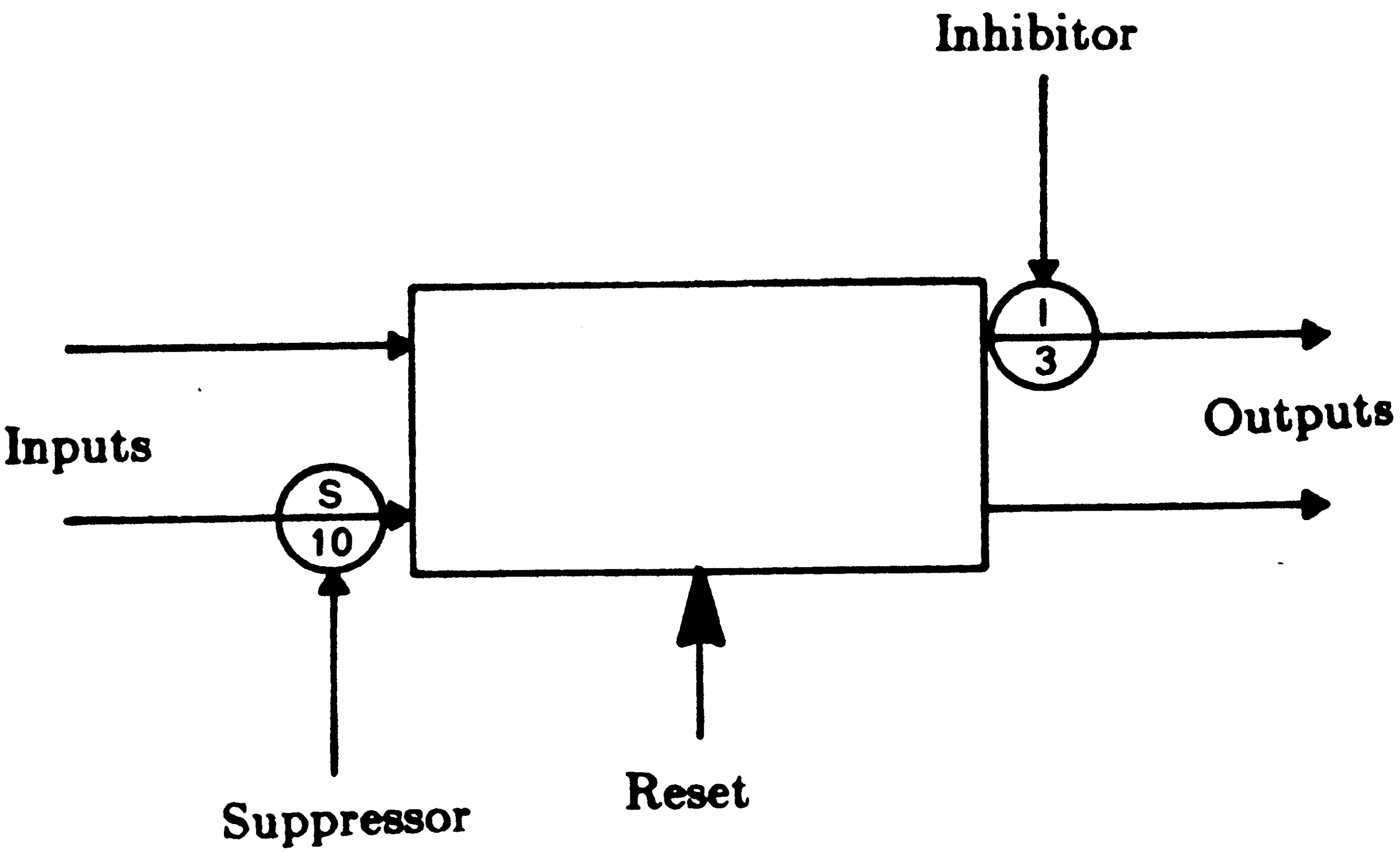
An alternative to the hierarchical paradigm with its horizontally organized architecture is called the *reactive paradigm* and is labeled as such above. Adherents of the reactive paradigm organize the components vertically so that there is a more direct route from sensors to effectors. Schematically Brooks depicts the paradigm as follows:



Note that in this vertical decomposition there is the potential for contention over the effectors. Just as in the example of steering a car to exit from the highway while avoiding an accident with the car in the lane to your right, there is contention among the various components, avoiding, exploring, wandering, planning, for taking control of the robots actuators. Brooks suggests that we solve the problem of contention by allowing components at one level to *subsume* components at a lower level. Thus called this approach as *subsumption architecture*.



In the subsumption architecture, components behaviors are divided into layers with an arbitration scheme whereby behaviors at one level can manipulate what behaviors at a lower level can see or do. Brooks called the most primitive components of his architecture *modules*. Each module has inputs, outputs and a reset. A module at a higher level can suppress the input of a module at a lower level thereby preventing the module from seeing a value at its input. A module can also inhibit the output of a module at a lower level thereby preventing that output from being propagated to other modules.



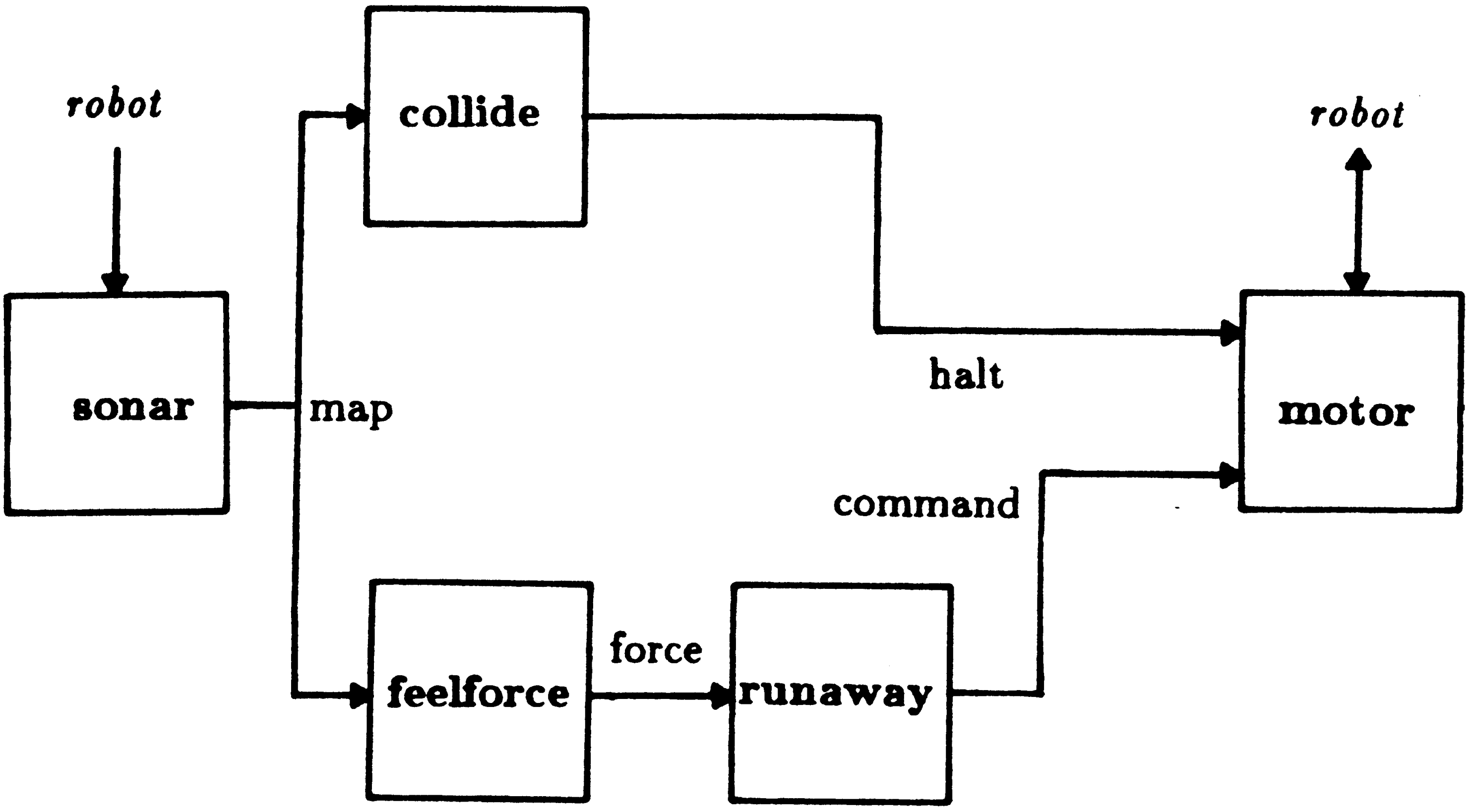
The modules are meant to be simple computationally and so it is reasonable to think of them as circuits or finite state-machines. Brooks assumed that they were augmented finite state controllers. The reset would cause the controller to return to its initial state. Once set in motion the controllers would continuously transition from one state to the next. The transitions can be determined in part by reading from the inputs and some internal state and of course by referring to the present state of the controller. Brooks also allows controllers to have an internal clock or timer and so, for example, they can execute a wait. Here are the basic transition types allowed in specifying the transition function of a finite-state controller.

- **Output** - a transition can compute a value as a function of the module's inputs and internal state and then send the value to one of its outputs before entering a specified state
- **Side effect** - a transition can set one of the module's instance variables (internal state) to some value computed as a function of the module's inputs and internal state; the module then enters a specified state
- **Conditional dispatch** - a predicate on the module's inputs and instance variables is evaluated and depending on the outcome the module enters one of two specified states
- **Event dispatch** - a sequence of conditions and states to branch to is specified; the conditions are then monitored continuously until a condition is met and then the module transitions to the corresponding state

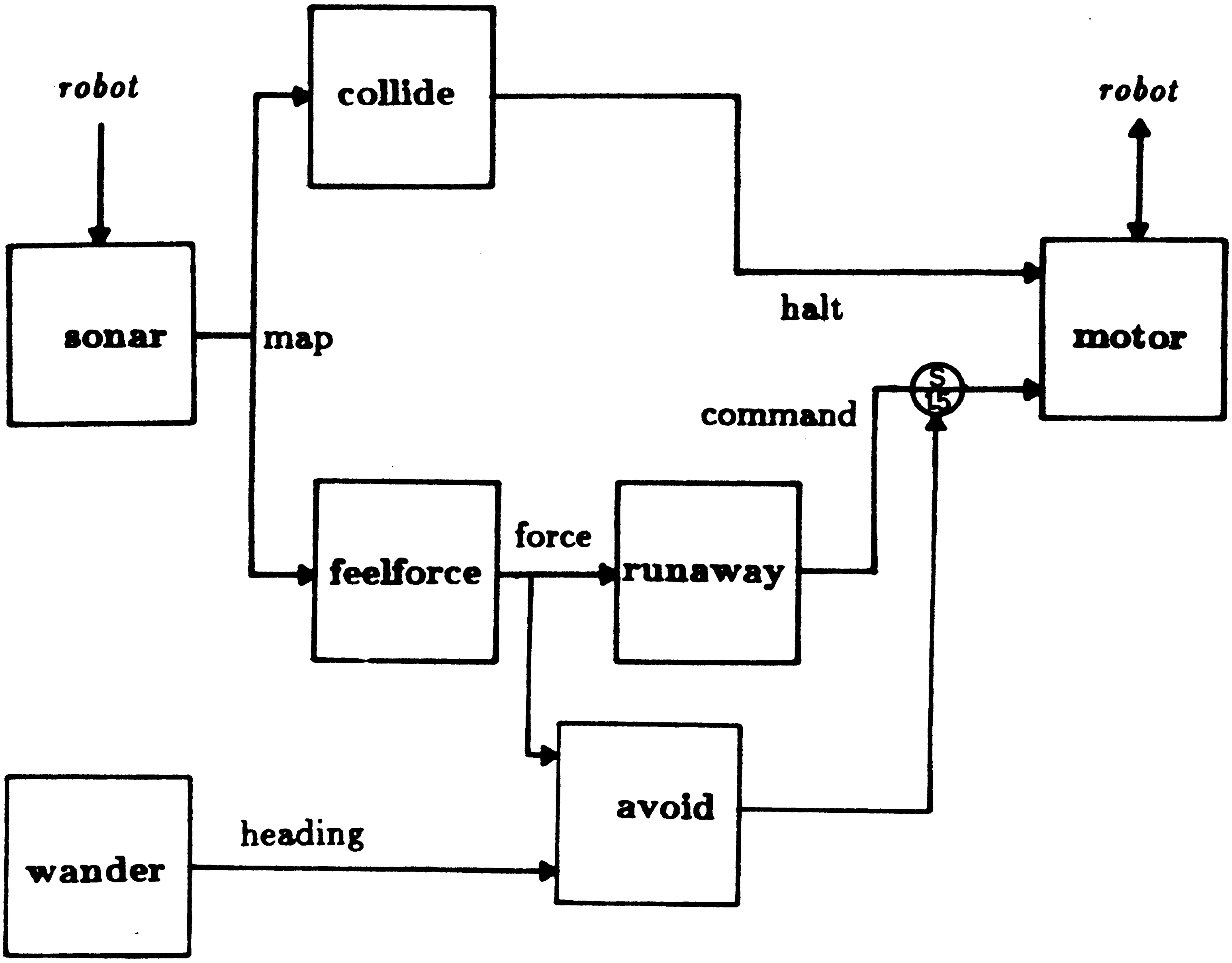
In some applications each module is implemented on a separate procesor with information propagated from inputs to outputs using parallel or serial communications. However, there is nothing preventing us from implementing more than one or all of the modules on a single processor with propagation carried out using shared variables. Here is the specification for a module that is used as one component in a level responsible for avoiding obstacles. The specification language has a lisp-like syntax but the expressions would typically be compiled into assembly code or into an intermediate target language like C.

```
(defmodule avoid
  :inputs (force heading)
  :outputs (command)
  :instance-vars (resultforce)
  :states
    ((nil (event-dispatch (and force heading) plan))
     (plan (setf resultforce (select-direction force heading))
           go)
     (go (conditional-dispatch (significant-force-p resultforce 1.0)
                               start
                               nil))
     (start (output command (follow-force resultforce))
            nil)))
```

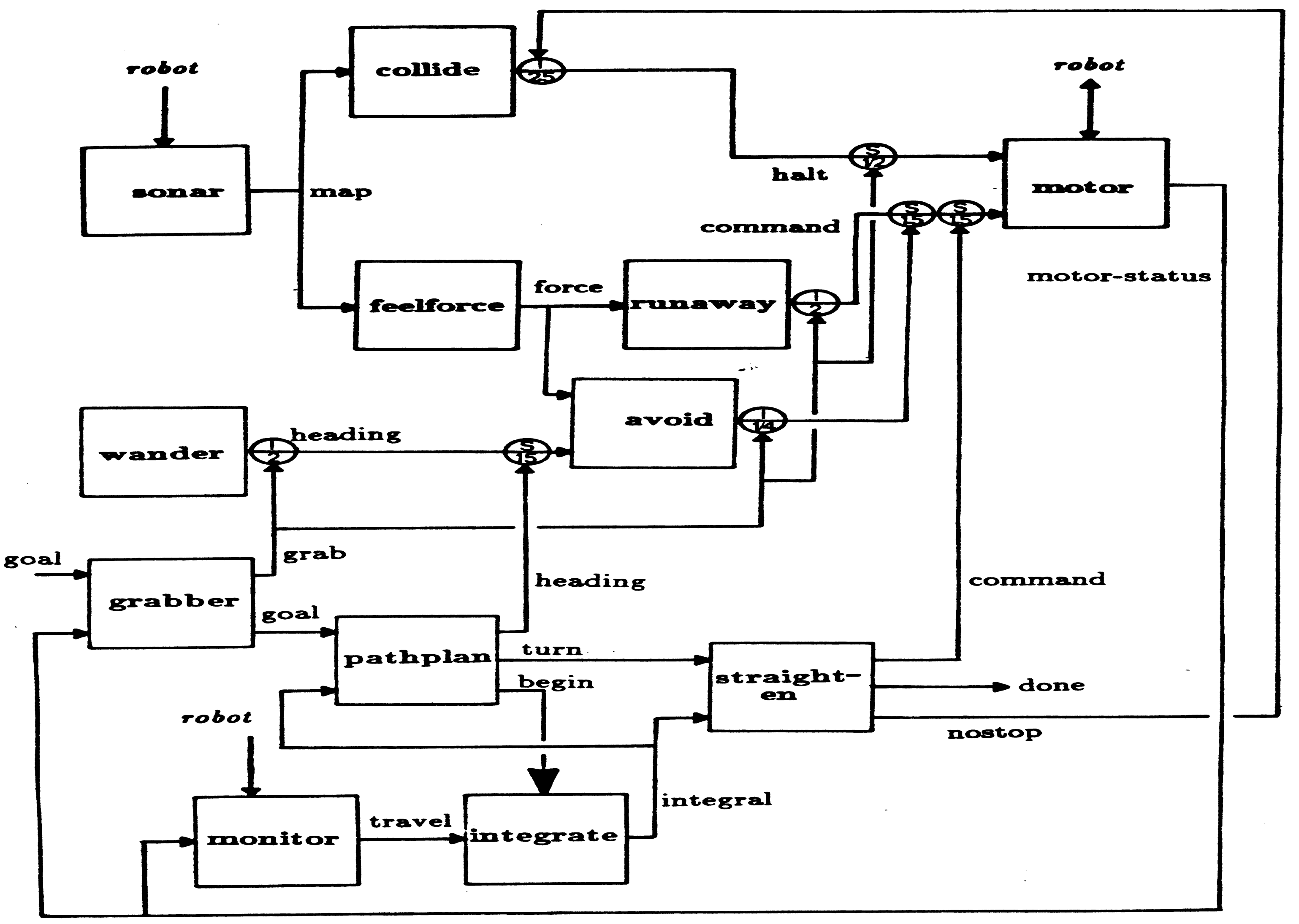
Here is the most primitive level of a mobile robot system consisting of three modules (plus the sonar and motor components which can also be thought of as modules of a sort). We assume a ring of sonars that provides at intervals an updated vector of sonar readings here referred to as a *map*. The collide module looks at the sonar vector and if it determines there is an imminent collision, then it sends a halt command to the motors. The feelforce module treats the sonars as repulsive forces acting on the robot and computes a force vector summarizing these repulsive forces. The feelforce module sends this force vector to the runaway module which computes motor commands to move the robot in accord with the perceived repulsive forces acting on it. The resulting (Level 0) behavior has robot moving away from any obstacles probably ending up in the middle of room. To get more interesting behavior we can add a second level.



In the diagram below we add two Level 1 modules. One of these modules, the wander module, selects an arbitrary heading possibly changing it from time to time using its internal timer and a source of random directions. This heading plus the force vector generated by the feelforce module are provided as input to the avoid module that computes motor commands combining the desired heading with the necessities of avoiding nearby obstacles. In order to prevent the runaway module taking over the control of the motors, the avoid module suppresses the output of the runaway module. The combined Level 0 and Level 1 behavior result in a robot that wanders aimless about but avoid obstacles.



The final diagram below taken from Brooks paper shows an additional level of control that implements a more goal-oriented behavior on top the of more primitive behaviors.



Here are some photographs of some veteran robots from MIT several of which, e.g., Herbert, used variations on the subsumption architecture. Flakey is a mobile robot from SRI that combines aspects of reactive and hierarchical control in a hybrid architecture.

Implementing Subsumption

Last year I developed a compiler in scheme that converted Brooks subsumption language syntax to NQC code. Here is a very simple subsumption code fragment and the resulting NQC output. Because of its limited numbers of processes (tasks) and variables NQC never proved practical as a target language. Eventually I had a compiler that handle most of the subsumption language constructs but would only work for cases involving five or six modules. I thought of converting my compiler to legOS but it didn't seem worth all the work given that the basic ideas of subsumption are pretty easy to incorporate in code and the rest is just syntax. Well, some might say that the syntax enforces a discipline but others just find the discipline confining. In any case, if you want to use subsumption, you can either write your own compiler or you can distill out the basic ideas and implement them in your own syntax.

What are the basic principles?

1. Divide your problem into basic competencies ordered simple to more complex. Designate a level for each basic competency.

2. Further subdivide each level into multiple simple components which interact through shared variables. Limit the sharing of variables among levels to avoid incomprehensible code.
3. Implement each module as a separate light-weight thread. You might think of setting the priorities for these threads so that modules in a given level have the same priority.
4. Implement suppression and inhibition as one or more separate "arbitration" processes that serve to control access to shared variables. You might want to control access using semaphores.