2 *The Hierarchical Paradigm*

Chapter Objectives:

- Describe the Hierarchical Paradigm in terms of the three robot paradigms and its organization of sensing.
- Name and evaluate one representative Hierarchical architecture in terms of: support for modularity, niche targetability, ease of portability to other domains, robustness.
- Solve a simple navigation problem using Strips, given a world model, operators, difference table, and difference evaluator. The state of the world model should be shown after each step.
- Understand the meaning of the following terms as applied to robotics: precondition, closed world assumption, open world, frame problem.
- Describe the mission planner, navigator, pilot organization of the Nested Hierarchical Controller.
- List two advantages and disadvantages of the Hierarchical Paradigm.

2.1 Overview

The Hierarchical Paradigm is historically the oldest method of organizing intelligence in mainstream robotics. It dominated robot implementations from 1967, with the inception of the first AI robot, Shakey (Fig. 2.1) at SRI, up until the late 1980's when the Reactive Paradigm burst onto the scene.

This chapter begins with a description of the Hierarchical Paradigm in terms of the SENSE, PLAN, ACT primitives and by its sensing representation. The chapter then covers the use of Strips in Shakey to reason and plan

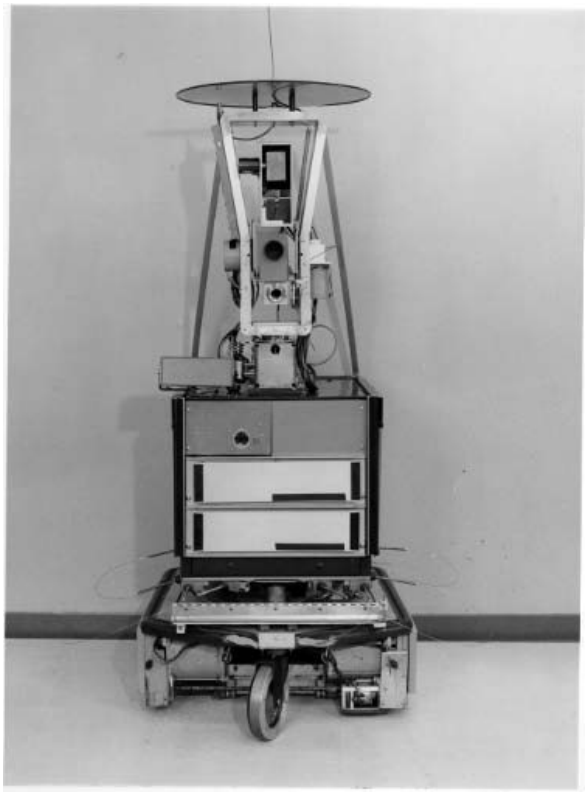


Figure 2.1 Shakey, the first AI robot. It was built by SRI for DARPA 1967–70. (Photograph courtesy of SRI.)

a path. Strips will serve to motivate the reader as to the computer challenges inherent in even as simple a task as walking across a room. However, Strips is not an architecture, per se, just an interesting technique which emerged from trying to build an architecture. Two representative architectures are presented, NHC and RCS, that serve as examples of robot architectures popular at the time. The chapter concludes with programming considerations.

2.2 Attributes of the Hierarchical Paradigm

As noted in Part I, a robotic paradigm is defined by the relationship between the three primitives (SENSE, PLAN, ACT) and by the way sensory data is processed and distributed through the system.

The Hierarchical Paradigm is sequential and orderly, as shown in Figs. 2.2 and 2.3. First the robot senses the world and constructs a global world map. Then "eyes" closed, the robot plans all the directives needed to reach the

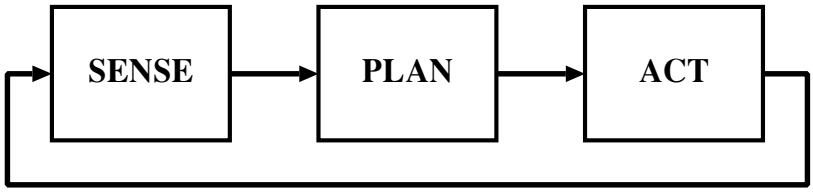


Figure 2.2 S,P,A organization of Hierarchical Paradigm.

ROBOT PRIMITIVES	INPUT	OUTPUT
SENSE	Sensor data	Sensed information
PLAN	Information (sensed and/or cognitive)	Directives
ACT	directives	Actuator commands

Figure 2.3 Alternative description of how the 3 primitives interact in the Hierarchical Paradigm.

goal. Finally, the robot acts to carry out the first directive. After the robot has carried out the SENSE-PLAN-ACT sequence, it begins the cycle again: eyes open, the robot senses the consequence of its action, replans the directives (even though the directives may not have changed), and acts.

As shown in Fig. 2.3, sensing in the Hierarchical Paradigm is monolithic: all the sensor observations are fused into one global data structure, which the planner accesses. The global data structure is generally referred to as a *world model*. The term world model is very broad; "world" means both the outside world, and whatever meaning the robot ascribes to it. In the Hierarchical Paradigm, the world model typically contains

WORLD MODEL

A PRIORI

1. an *a priori* (previously acquired) representation of the environment the robot is operating in (e.g., a map of the building),
2. sensing information (e.g., "I am in a hallway, based on where I've traveled, I must be in the northwest hallway"), plus
3. any additional cognitive knowledge that might be needed to accomplish a task (e.g., all packages received in the mail need to be delivered to Room 118).

Creating a single representation which can store all of this information can be very challenging. Part of the reason for the "sub-turtle" velocity was the lack of computing power during the 1960's. However, as roboticists in the 1980's began to study biological intelligence, a consensus arose that even with increased computing power, the hierarchical, logic-based approach was unsatisfactory for navigational tasks which require a rapid response time to an open world.

2.2.1 Strips

ALGORITHM

Shakey, the first AI mobile robot, needed a generalized algorithm for planning how to accomplish goals. (An *algorithm* is a procedure which is correct and terminates.) For example, it would be useful to have the same program allow a human to type in that the robot is in Office 311 and should go to Office 313 or that the robot is in 313 and should the red box.

GENERAL PROBLEM SOLVER (GPS)
STRIPS
MEANS-ENDS ANALYSIS

The method finally selected was a variant of the *General Problem Solver* method, called *Strips*. Strips uses an approach called *means-ends analysis*, where if the robot can't accomplish the task or reach the goal in one "movement", it picks a action which will reduce the difference between what state it was in now (e.g., where it was) versus the goal state (e.g., where it wanted to be). This is inspired by cognitive behavior in humans; if you can't see how to solve a problem, you try to solve a portion of the problem to see if it gets you closer to the complete solution.

GOAL STATE
INITIAL STATE
OPERATOR
DIFFERENCE
DIFFERENCE EVALUATOR

Consider trying to program a robot to figure out how to get to the Stanford AI Lab (SAIL). Unless the robot is at SAIL (represented in Strips as a variable `goal state`), some sort of transportation will have to arranged. Suppose the robot is in Tampa, Florida (`initial state`). The robot may represent the decision process of how to get to a location as function called an `operator` which would consider the Euclidean distance (a variable named `difference`) between the `goal state` and `initial state`. The difference between locations could be computed for comparison purposes, or evaluation, by the square of the hypotenuse (`difference evaluator`). For example using an arbitrary frame of reference that put Tampa at the center of the world with made-up distances to Stanford:

`initial state:`	Tampa, Florida (0,0)
`goal state:`	Stanford, California (1000,2828)
`difference:`	3,000

DIFFERENCE TABLE

This could lead to a data structure called a `difference table` of how to make decisions:

difference	operator
d≥200	fly
100<d < 200	ride_train
d ≤ 100	drive
d< 1	walk

Different modes of transportation are appropriate for different distances. A mode of transportation, `fly, ride_train, drive, walk`, in the table is really a function in the robot's program. It is also called an `operator`, because it reduces the value stored in `difference` as to the distance from being in the `initial state` of Tampa and wanting to be at the `goal state`. A robot following this difference table would begin by flying as close as it could to SAIL.

But suppose the robot flew into the San Francisco airport. It'd be within 100 miles of SAIL, so the robot appears to have made an intelligent decision. But now the robot has a new difference to reduce. It examines the difference table with a new value of `difference`. The table says the robot should `drive`. Drive what? A car? Ooops: if the robot did have a personal car, it would be back in Tampa. The robot needs to be able to distinguish between driving its car and driving a rental car. This is done by listing

PRECONDITIONS

the `preconditions` that have to be true in order to execute that particular operator. The preconditions are a column in the difference table, where a single operator can have multiple preconditions. In practice, the list of preconditions is quite lengthy, but for the purposes of this example, only `drive_rental, drive_personal` will be shown with preconditions.

difference	operator	preconditions
d≤200	fly	
100<d<200	ride_train	
d≤100	drive_rental	at airport
	drive_personal	at home
d<1	walk	

The difference table is now able to handle the issue of deciding to drive a rental car. But this introduces a new issue: how does the robot know where it is at? This is done by monitoring the state of the robot and its world. If it took an airplane from Tampa to the San Francisco airport, its state has changed. Its `initial state` is now at the San Francisco airport, and no

longer Tampa. Therefore, whenever the robot executes an operator, there is almost always something that has to be added to the robot's knowledge of the world state (which is entered into a `add-list`) and something that has to be deleted (`delete-list`). This two lists are stored in the difference table so that when the robot picks an operator based on the difference and operator, it can easily apply the appropriate modifications to the world. The difference table below is expanded to show the add and delete lists.

ADD-LIST
DELETE-LIST

difference	operator	pre-conditions	add-list	delete-list
d≤200	fly		at Y at airport	at X
100<d<200	train		at Y at station	at X
d≤100	drive_rental	at airport		
	drive_personal	at home		
d<1	walk			

Of course, the above difference table is fairly incomplete. Driving a rental car should have a precondition that there is a rental car available. (And that the robot have a waiver from the state highway patrol to drive as an experimental vehicle and a satisfactory method of payment.) The number of facts and preconditions that have to be explicitly represented seem to be growing explosively. Which is Very Bad from a programming standpoint.

The main point is that the difference table appears to be a good data structure for representing what a robot needs in planning a trip. It should be apparent that a recursive function can be written which literally examines each entry in the table for the first operator that reduces the difference. The resulting list of operators is actually the plan: a list of the steps (operators) that the robot has to perform in order to reach a goal. The robot actually constructs the plan before handing it off to another program to execute.

At this point in time, it isn't likely that a robot will get on a plane and then drive. So perhaps the criticisms of Strips is because the example used too complicated a task to be realistic. Let's see if Strips is more streamlined with a simple task of getting from one room to another.

2.2.2 More realistic Strips example

The first step in creating a Strips planner is to construct a Strips-based representation of the world, or `world model`. Everything in the world that is

CONSTRUCTING A WORLD MODEL

AXIOMS
PREDICATES

relevant to the problem is represented by facts, or *axioms*, in predicate logic. *Predicates* are functions that evaluate to `TRUE` or `FALSE`. By years of AI programming convention, predicates are usually written in uppercase.

Consider the problem of a robot named IT in a room, R1, who needs to go to another room, R2, in the house shown in Fig. 2.4. In order to solve this problem using Strips, the robot has to be given a way of representing the world, which will in turn influence the difference table, a difference evaluator, and how the add and delete lists are written. The world model in the previous example was never formally defined.

A world model is generally built up from static facts (represented as predicates) from a set of candidates, and things in the world, like the robot. The robot's name is in all capitals because it exists (and therefore is `TRUE`). Lowercase identifiers indicate that the thing is a variable, that a real thing hasn't been assigned to that placeholder yet.

Suppose the robot was limited to knowing only whether a movable object was in a room, next to a door or another movable object, and whether a door was open or closed and what rooms it connected. In a programming sense, there would be only three types of things in the world: `movable_object` (such as the robot, boxes it should pick up), `room`, and `door`. The robot's knowledge could be represented by the following predicates:

`INROOM(x, r)`	where `x` is an object of type `movable_object`, `r` is type `room`
`NEXTTO(x, t)`	where `x` is a `movable_object`, `t` is type `door` or `movable_object`
`STATUS(d, s)`	where `d` is type `door`, `s` is an enumerated type: `OPEN` or `CLOSED`
`CONNECTS(d, rx, ry)`	where `d` is type `door`, `rx, ry` are the `room`

With the above predicates, the world model for the initial state of the world in Fig. 2.4 would be represented by:

```
initial state:
INROOM(IT, R1)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
```

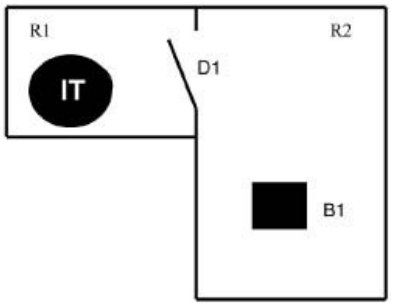


Figure 2.4 An example for Strips of two rooms joined by an open door.

This world model captures that a specific `movable_object` named `IT` is in a room named `R1`, and `B1` is in another room labeled `R2`. A door `D1` connects a room named `R1` to `R2` and it connects `R2` to `R1`. (Two different `CONNECTS` predicates are used to represent that a robot can go through the door from either door.) A door called `D1` has the enumerated value of being `OPEN`. The `NEXTTO` predicate wasn't used, because it wasn't true and there would be nothing to bind the variables to.

Under this style of representation, the world model for the `goal state` would be:

```
goal state:
INROOM(IT,R2)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
```

CONSTRUCTING THE DIFFERENCE TABLE

Once the world model is established, it is possible to construct the difference table. The partial difference table is:

operator	preconditions	add-list	delete-list
OP1: GOTODOOR(IT,dx)	INROOM(IT,rk) CONNECT(dx,rk,rm)	NEXTTO(IT,dx)	
OP2: GOTHRUDOOR(IT,dx)	CONNECT(dx,rk,rm) NEXTTO(IT,dx) STATUS(dx, OPEN) INROOM(IT,rk)	INROOM(IT,rm)	INROOM(IT,rk)

This difference table says the robot is programmed for only two operations: go to a door, and go through a door. The GOTODOOR operator can be applied only if the following two preconditions are true:

- INROOM(IT, rk) The robot is in a room, which will be assigned to the identifier rk.
- CONNECT(dx, rk, rm) There is a door, which will be assigned to the identifier dx, which connects rk to some other room called rm.

The label IT is used to constrain the predicates. Notice that only the variables dx and rk get bound when GOTODOOR is called. rm can be anything. If GOTODOOR is executed, the robot is now next to the door called dx. Nothing gets deleted from the world state because the robot is still in room rk, the door dx still connects the two rooms rk and rm. The only thing that has changed is that the robot is now in a noteworthy position in the room: next to the door.

The difference table specifies that the GOTHRUDOOR operator will only work if the robot is in the room next to the door, the door is open, and the door connects the room the robot is in to another room. In this case, predicates must be added and deleted from the world model when the operator executes. When the robot is in room rk and goes through the door, it is now in room rm (which must be added to the world model) and is no longer in room rk (which must be deleted).

So far, the world model and difference table should seem reasonable, although tedious to construct. But constructing a difference table is pointless without an evaluation function for differences. (Notice that there wasn't a column for the difference in the above table.) The difference evaluator in the travel example was Euclidean distance. In this example, the evaluator is predicate calculus, where the initial state is logically subtracted from the goal state. The logical difference between the initial state goal state is simply:

¬INROOM(IT, R2) or INROOM(IT, R2)=FALSE

REDUCING DIFFERENCES

Reducing differences is a bit like a jigsaw puzzle where Strips tries different substitutions to see if a particular operator will reduce the difference. In order to reduce the difference, Strips looks in the difference table, starting at the top, under the add-list column for a match. It looks in the add-list rather than a separate differences column because the add-list expresses what the result of the operator is. If Strips finds an operator that produces the goal state, then that operator eliminates the existing difference between the initial and goal states.

The add-list in `OP2: GOTHRUDOOR` has a match on form. If `rm=R2`, then the result of `OP2` would be `INROOM(IT, R2)`. This would eliminate the difference, so `OP2` is a candidate operator.

Before the `OP2` can be applied, Strips must check the preconditions. To do this, `rm` must be replaced with `R2` in every predicate in the preconditions. `OP2` has two preconditions, only `CONNECTS(dx, rk, rm)` is affected. It becomes `CONNECTS(dx, rk, R2)`. Until `dx` and `rk` are bound, the predicate doesn't have a true or false value. Essentially `dx, rk` are wildcards, `CONNECTS(*, *, R2)`. To fill in values for these variables, Strips looks at the current state of the world model to find a match. The predicate in the current state of the world `CONNECTS(D1, R1, R2)` matches `CONNECTS(*, *, R2)`. D1 is now bound to `dx` and R1 is bound to `rk`.

Now Strips propagates the bindings to the next precondition on the list: `NEXTTO(IT, dx)`. `NEXTTO(IT, D1)` is `FALSE` because the predicate is not in the current world state. `NEXTTO(IT, D1)` is referred to as a *failed precondition*. An informal interpretation is that `GOTHRUDOOR(IT, D1)` will get the robot to the goal state, but before it can do that, IT has to be next to D1.

FAILED PRECONDITIONS

RECURSION TO RESOLVE DIFFERENCES

Rather than give up, STRIPS *recurses* (uses the programming technique of recursion) to repeat the entire procedure. It marks the original goal state as `G0`, pushes it on a stack, then it creates a new sub-goal state, `G1`.

The difference between NEXTTO(IT, D1) and the current world state is:

```
¬NEXTTO(IT, D1)
```

Strips once again searches through the add-list in the difference table to find an operator that would negate this. Indeed, `OP1: GOTODOOR(IT, dx)` has a match in the add-list of `NEXTTO(IT, dx)`. Strips has to start over with reassigning values to the identifiers because the program has entered a new programming scope, so `dx=D1`.

Again, Strips examines the preconditions. This time `rk=R1` and `rm=R2` can be matched with `CONNECTS(dx, rk, rm)`, and all preconditions are satisfied (that is, they evaluate to true). Strips puts the operator `OP1` on the plan stack and applies the operator to the world model, changing the state. (Note that this is the equivalent of a "mental operation"; the robot doesn't actually physically go to the door, it just changes the state to imagine what would happen if it did.)

To recall, the initial state of the world model was:

```
initial state:
INROOM(IT, R1)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
```

Applying the operator `OP1` means making the changes on the add-list and delete-list. There is only a predicate on the add-list and none on the delete-list. After adding `NEXTTO(IT, D1)`, the state of the world is:

```
state after OP1:
INROOM(IT, R1)
INROOM(B1,R2)
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
NEXTTO(IT, D1)
```

Strips then returns control back to the previous call. It resumes where it left off in evaluating the preconditions for `OP2` with `dx=D1, rm=R2` and `rk=R1`, only now the world model has changed. Both `STATUS(D1, OPEN)` and INROOM(IT, R1) are true, so all the preconditions for `OP2` are satisfied. Strips puts `OP2` on its plan stack and changes the world model by applying the add-list and delete-list predicates. This results in what the state of the world will be when the plan executes:

```
state after OP2:
INROOM(IT, R2)
INROOM(B1,R2)
```

```
CONNECTS(D1, R1, R2)
CONNECTS(D1, R2, R1)
STATUS(D1,OPEN)
NEXTTO(IT, D1)
```

Strips exits and the plan for the robot to physically execute (in reverse order on the stack) is: `GOTODOOR(IT, D1), GOTHRUDOOR(IT, D1).`

2.2.3 Strips summary

Strips works recursively; if it can't reach a goal directly, it identifies the problem (a failed precondition), then makes the failed precondition a subgoal. Once the subgoal is reached, Strips puts the operator for reaching the subgoal on a list, then backs up (pops the stack) and resumes trying to reach the previous goal. Strips plans rather than execute: it creates a list of operators to apply; it does not apply the operator as it goes. Strips implementations requires the designer to set up a:

- world model representation
- difference table with operators, preconditions, add, and delete lists
- difference evaluator

The steps in executing Strips are:

1. Compute the difference between the goal state and the initial state using the difference evaluation function. If there is no difference, terminate.
2. If there is a difference, reduce the difference by selecting the first operator from the difference table whose add-list has a predicate which negates the difference.
3. Next, examine the preconditions to see if a set of bindings for the variables can be obtained which are all true. If not, take the first false precondition, make it the new goal and store the original goal by pushing it on a stack. Recursively reduce that difference by repeating step 2 and 3.
4. When all preconditions for an operator match, push the operator onto the plan stack and update a copy of the world model. Then return to the operator with the failed precondition so it can apply its operator or recurse on another failed precondition.

2.3 Closed World Assumption and the Frame Problem

CLOSED WORLD ASSUMPTION
FRAME PROBLEM

Strips sensitized the robotics community to two pervasive issues: the *closed world assumption* and the *frame problem*. As defined earlier, the closed world assumption says that the world model contains everything the robot needs to know: there can be no surprises. If the closed world assumption is violated, the robot may not be able to function correctly. But, on the other hand, it is very easy to forget to put all the necessary details into the world model. As a result, the success of the robot depends on how well the human programmer can think of everything.

But even assuming that the programmer did come up with all the cases, the resulting world model is likely to be huge. Consider how big and cumbersome the world model was just for moving between 2 rooms. And there were no obstacles! People began to realize that the number of facts (or axioms) that the program would have to sort through for each pass through the difference table was going to become intractable for any realistic application. The problem of representing a real-world situation in a way that was computationally tractable became known as the frame problem. The opposite of the closed world assumption is known as the *open world assumption*. When roboticists say that "a robot must function in the open world," they are saying the closed world assumption cannot be applied to that particular domain.

OPEN WORLD ASSUMPTION

The above example, although trivial, shows how tedious Strips is (though computers are good at tedious algorithms). In particular, the need to formally represent the world and then maintain every change about it is non-intuitive. It also illustrates the advantage of a closed-world assumption: imagine how difficult it would be to modify the planning algorithm if the world model could suddenly change. The algorithm could get lost between recursions. The example should also bring home the meaning of the frame problem: imagine what happens to the size of the world model if a third room is added with boxes for the robot to move to and pick up! And this is only for a world of rooms and boxes. Clearly the axioms which frame the world will become too numerous for any realistic domain.

One early solution was ABStrips which tried to divide the problem into multiple layers of abstraction, i.e., solve the problem on a coarse level first. That had its drawbacks, and soon many people who had started out in robotics found themselves working on an area of AI called planning. The two fields became distinct, and by the 1980's, the planning and robotics researchers had separate conferences and publications. Many roboticists dur-

ing the 1970's and 1980's worked on either computer vision related issues, trying to get the robots to be able to better sense the world, or on path planning, computing the most efficient route around obstacles, etc. to a goal location.

2.4 Representative Architectures

As mentioned in Part I an architecture is a method of implementing a paradigm, of embodying the principles in some concrete way. Ideally, an architecture is generic; like a good object-oriented program design, it should have many reusable pieces for other robot platforms and tasks.

Possibly the two best known architectures of the Hierarchical period are the Nested Hierarchical Controller (NHC) developed by Meystel[93] and the NIST Realtime Control System (RCS) originally developed by Albus,[1] with its teleoperation version for JPL called NASREM.

2.4.1 Nested Hierarchical Controller

As shown in Fig. 2.5, the Nested Hierarchical Controller architecture has components that are easily identified as either SENSE, PLAN, or ACT. The robot begins by gathering observations from its sensors and combining those observations to form the World Model data structure through the SENSE activity. The World Model may also contain *a priori* knowledge about the world, for example, maps of a building, rules saying to stay away from the foyer during the start and finish of business hours, etc. After the World Model has been created or updated, then the robot can PLAN what actions it should take. Planning for navigation has a local procedure consisting of three steps executed by the Mission Planner, Navigator, and Pilot. Each of these modules has access to the World Model in order to compute their portion of planning. The last step in planning is for the Pilot module to generate specific actions for the robot to do (e.g., Turn left, turn right, move straight at a velocity of 0.6 meters per second). These actions are translated into actuator control signals (e.g., Velocity profile for a smooth turn) by the Low-Level Controller. Together, the Low-Level Controller and actuators form the ACT portion of the architecture.

The major contribution of NHC was its clever decomposition of planning into 3 different functions or subsystems aimed at supporting navigation: the *Mission Planner*, the *Navigator*, and the *Pilot*. As shown in Fig. 2.6, the Mission Planner either receives a mission from a human or generates a mission

MISSION PLANNER
NAVIGATOR
PILOT

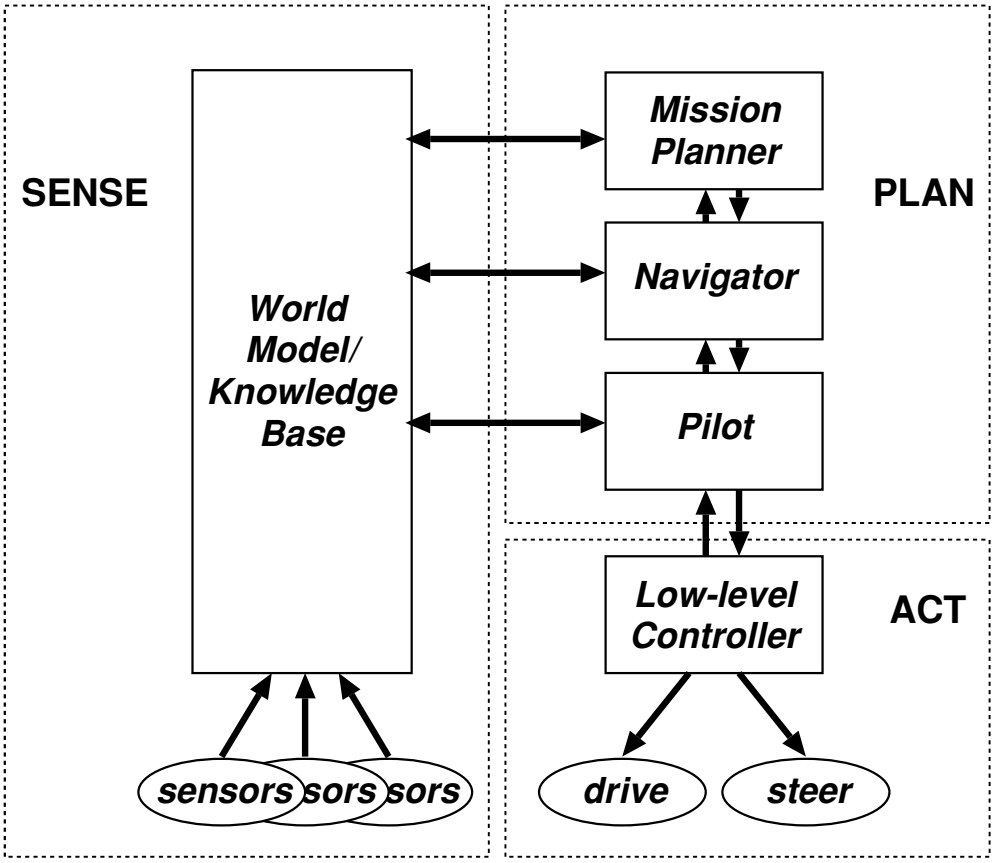


Figure 2.5 Nested Hierarchical Controller.

for itself, for example: pick up the box in the next room. The Mission Planner is responsible for operationalizing, or translating, this mission into terms that other functions can understand: box=B1; rm=ROOM311. The Mission Planner then accesses a map of the building and locates where the robot is and where the goal is. The Navigator takes this information and generates a path from the current location to the goal. It generates a set of waypoints, or straight lines for the robot to follow. The path is passed to the Pilot. The Pilot takes the first straight line or path segment and determines what actions the robot has to do to follow the path segment. For instance, the robot may need to turn around to face the way point before it can start driving forward.

What happens if the Pilot gives directions for a long path segment (say 50 meters) or if a person suddenly walks in front of the robot? Unlike Shakey, under NHC, the robot is not necessarily walking around with its eyes closed. After the Pilot gives the Low-Level Controller commands and the controller

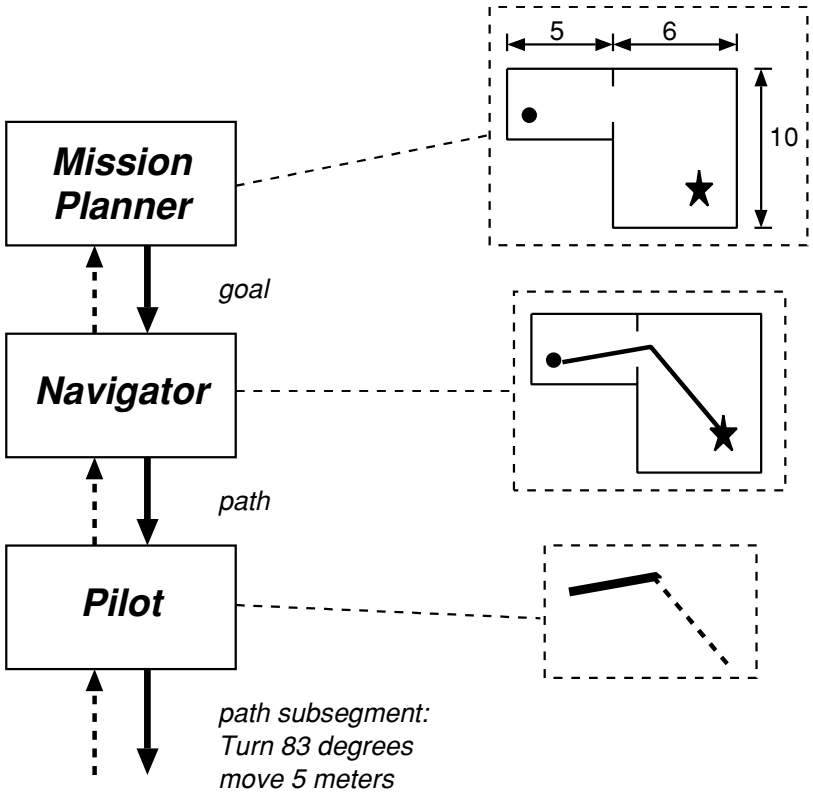


Figure 2.6 Examination of planning components in the NHC architecture.

sends actuator signals, the robot polls its sensors again. The World Model is updated. However, the entire planning cycle does not repeat. Since the robot has a plan, it doesn't need to rerun the Mission Planner or the Navigator. Instead, the Pilot checks the World Model to see if the robot has drifted off the path subsegment (in which case it generates a new control signal), has reached the waypoint, or if an obstacle has appeared. If the robot has reached its waypoint, the Pilot informs the Navigator. If the waypoint isn't the goal, then there is another path subsegment for the robot to follow, and so the Navigator passes the new subsegment to the Pilot. If the waypoint is the goal, then the Navigator informs the Mission Planner that the robot has reached the goal. The Mission Planner may then issue a new goal, e.g., Return to the starting place. If the robot has encountered an obstacle to its path, the Pilot passes control back to the Navigator. The Navigator must compute a new path, and subsegments, based on the updated World Model. Then it gives the updated path subsegment to the Pilot to carry out.

NHC has several advantages. It differs from Strips in that it interleaves planning and acting. The robot comes up with a plan, starts executing it, then changes it if the world is different than it expected. Notice that the decomposition is inherently hierarchical in intelligence and scope. The Mission Planner is "smarter" than the Navigator, who is smarter than the Pilot. The Mission Planner is responsible for a higher level of abstraction then the Navigator, etc. We will see that other architectures, both in the Hierarchical and Hybrid paradigms, will make use of the NHC organization.

One disadvantage of the NHC decomposition of the planning function is that it is appropriate only for navigation tasks. The division of responsibilities seems less helpful, or clear, for tasks such as picking up a box, rather than just moving over to it. The role of a Pilot in controlling end-effectors is not clear. At the time of its initial development, NHC was never implemented and tested on a real mobile robot; hardware costs during the Hierarchical period forced most roboticists to work in simulation.

2.4.2 NIST RCS

Jim Albus at the National Bureau of Standards (later renamed the National Institute of Standards and Technology or NIST) anticipated the need for intelligent industrial manipulators, even as engineering and AI researchers were splitting into two groups. He saw that one of the major obstacles in applying AI to manufacturing robots was that there were no common terms, no common set of design standards. This made industry and equipment manufacturers leery of AI, for fear of buying an expensive robot that would not be compatible with robots purchased in the future. He developed a very detailed architecture called the Real-time Control System (RCS) Architecture to serve as a guide for manufacturers who wanted to add more intelligence to their robots. RCS used NHC in its design, as shown in Fig. 2.7.

SENSE activities are grouped into a set of modules under the heading of sensory perception. The output of the sensors is passed off to the world modeling module which constructs a global map using information in its associated knowledge database about the sensors and any domain knowledge (e.g., the robot is operating underwater). This organization is similar to NHC. The main difference is that the sensory perception module introduces a useful preprocessing step between the sensor and the fusion into a world model. As will be seen in Ch. 6, sensor preprocessing is often referred to as *feature extraction*.

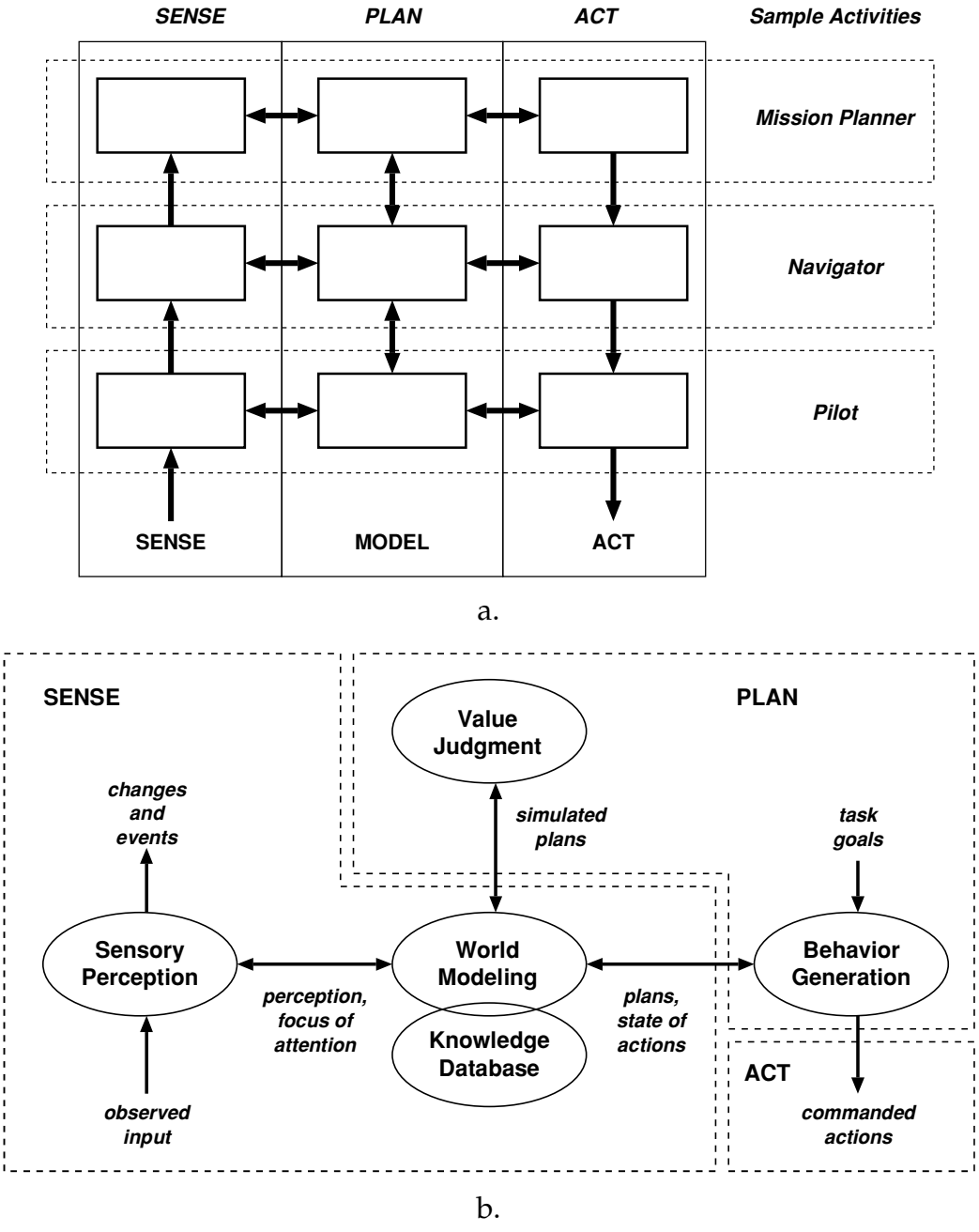


Figure 2.7 Layout of RCS: a.) hierarchical layering of sense-model-act, and b.) functional decomposition.

The Value Judgment module provides most of the functionality associated with the PLAN activity: it plans, then simulates the plans to ensure they will work. Then, as with Shakey, the Planner hands off the plan to another module, Behavior Generation, which converts the plans into actions that the robot can actually perform (ACT). Notice that the Behavior Generation module is similar to the Pilot in NHC, but there appears to be less focus on navigation tasks. The term "behavior" will be used by Reactive and Hybrid Deliberative/Reactive architectures. (This use of "behavior" in RCS is a bit of retrofit, as Albus and his colleagues at NIST have attempted to incorporate new advances. The integration of all sensing into a global world model for planning and acting keeps RCS a Hierarchical architecture.) There is another module, operator interface, which is not shown which allows a human to "observe" and debug what a program constructed with the architecture is doing.

The standard was adapted by many government agencies, such as NASA and the US Bureau of Mines, who were contracting with universities and companies to build robot prototypes. RCS serves as a blueprint for saying: "here's the types of sensors I want, and they'll be fused by this module into a global map, etc." The architecture was considered too detailed and restrictive when it was initially developed by most AI researchers, who continued development of new architectures and paradigms on their own. Fig. 2.8 shows three of the diverse mobile robots that have used RCS.

A close inspection of the NHC and RCS architectures suggests that they are well suited for semi-autonomous control. The human operator could provide the world model (via eyes and brain), decide the mission, decompose it into a plan, and then into actions. The lower level controller (robot) would carry out the actions. As robotics advanced, the robot could replace more functions and "move up" the autonomy hierarchy. For example, taking over the pilot's responsibilities; the human could instruct the robot to stay on the road until the first left turn. As AI advanced, the human would only have to serve as the Mission Planner: "go to the White House." And so on. Albus noted this and worked with JPL to develop a version of RCS for teleoperating a robot arm in space. This is called the *NASREM* architecture and is still in use today.

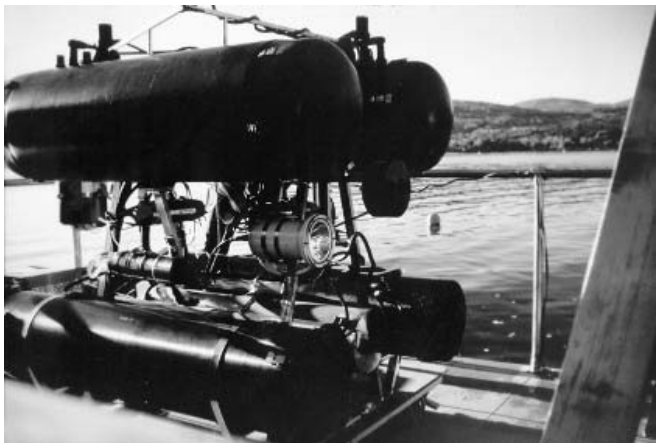
NASREM

2.4.3 Evaluation of hierarchical architectures

Recall from Part I that there are four criteria for evaluating an architecture: support for modularity, niche targetability, ease of portability to other domains, and robustness. NHC and RCS both provide some guidelines in how

a.

b.



c.

Figure 2.8 Three of the diverse mobile robots that have used RCS: a.) a commercial floor cleaning robot, b.) a mining robot, and c.) a submersible or underwater robot. (Photographs courtesy of the National Institute of Standards and Technology.)

to decompose a robot program into intuitive modules. The NHC decomposition of mission planner, navigator, and pilot was focused strictly on navigation, while RCS appears broader. Both have been used recently for successful vehicle guidance, with RCS being used to control mining equipment, submarines, and cars. So both have reasonable niche targetability. The ease of portability to other domains is unclear. The architectures are expressed at a very broad level, akin to "a house should have bedrooms, bathrooms, and a kitchen." Architectures which are more specific, "there should be one bathroom for every two bedrooms," and which have associated techniques simplify portability. It is hard to see how the code written for a mining machine

could be reused for a submarine, especially since RCS is not object-oriented. In terms of robustness, RCS does attempt to provide some explicit mechanisms. In particular, it assumes the Value Judgment module simulates a plan to confirm that it should be successful when deployed. The use of simulation is common for operating equipment in a well-known environment where every piece of equipment is known. The most notable example is a nuclear processing cell. With such detailed information, it is fairly straightforward (although computationally expensive) to simulate whether a particular course for a robot would collide with equipment and cause a spill. This is a very limited form of robustness. The disadvantage is the time delay caused by the robot mentally rehearsing its actions prior to executing them. Simulation may not be appropriate for all actions; if a piece of the ceiling is falling on the robot, it needs to get out of the way immediately or risk coming up with the best place to move too late to avoid being crushed.

2.5 Advantages and Disadvantages

Robots built in the time period before 1990 typically had a Hierarchical style of software organization. They were generally developed for a specific application rather than to serve as a generic architecture for future applications. The robots are interesting because they illustrate the diversity and scope of applications being considered for mobile robots as far back as 15 or 20 years ago.

The primary advantage of the Hierarchical Paradigm was that it provides an ordering of the relationship between sensing, planning, and acting. The primary disadvantage was planning. Every update cycle, the robot had to update a global world model and then do some type of planning. The sensing and planning algorithms of the day were extremely slow (and many still are), so this introduced a significant bottleneck. Notice also that sensing and acting are always disconnected. This effectively eliminated any stimulus-response types of actions ("a rock is crashing down on me, I should move *anywhere*") that are seen in nature.

The dependence on a global world model is related to the frame problem. In Strips, in order to do something as simple as opening a door, the robot had to reason over all sorts of details that were irrelevant (like other rooms, other doors). NHC and RCS represent attempts to divide up the world model into pieces best suited for the type of actions; for example, consider the roles of the Mission Planner, Navigator, and Pilot. Unfortunately, these decomposi-

tions appear to be dependent on a particular application. As a result, robotics gained a reputation as being more of an art than a science.

Another issue that was never really handled by architectures in the Hierarchical Paradigm was uncertainty. Uncertainty comes in many forms, such as semantic (how close does `NEXTTO` mean anyway?), sensor noise, and actuator errors. Another important aspect of uncertainty is action completion: did the robot actually accomplish the action? One robotics researcher said that their manipulator was only able to pick up a cup 60% of the attempts; therefore they had to write a program to check to see if it was holding a cup and then restart the action if it wasn't. Because Shakey essentially closed its eyes during planning and acting, it was vulnerable to uncertainty in action completion.

2.6 Programming Considerations

It is interesting to note that the use of predicate logic and recursion by Strips favors languages like Lisp and PROLOG. These languages were developed by AI researchers specifically for expressing logical operations. These languages do not necessarily have good real-time control properties like C or C++. However, during the 1960's the dominant scientific and engineering language was FORTRAN IV which did not support recursion. Therefore, researchers in AI robotics often chose the lesser of two evils and programmed in Lisp. The use of special AI languages for robotics may have aided the split between the engineering and AI approaches to robotics, as well as slowed down the infusion of ideas from the the two communities. It certainly discouraged non-AI researchers from becoming involved in AI robotics.

The Hierarchical Paradigm tends to encourage monolithic programming, rather than object-oriented styles. Although the NHC decomposes the planning portion of intelligence, the decomposition is strictly functional. In particular, NHC and RCS don't provide much guidance on how to build modular, reusable components.

2.7 Summary

The Hierarchical Paradigm uses a SENSE then PLAN then ACT (S,P,A). It organizes sensing into a global data structure usually called a world model that may have an associated knowledge base to contain *a priori* maps or knowledge relevant to a task. Global data structures often flag that an ar-