11 *The Building Blocks of Control Control Architectures*

The job of the controller is to provide the brains for the robot and so it can be autonomous and achieve goals. We have seen that feedback control is a very good way to write controllers for making a robot perform a single behavior, such as following a wall, avoiding obstacles, and so on. Those behaviors do not require much thinking, though. Most robots have more things to do than just follow a wall or avoid obstacles, ranging from simple survival (not running into things or running out of power) to complex task achievement (whatever it may be). Having to do multiple things at once and deciding what to do at any point in time is not simple, even for people, much less for robots. Therefore, putting together controllers that will get the robot to produce the desired overall behavior is not simple, but it is what robot control is really about.

> *So how would you put multiple feedback controllers together? What if you need more than feedback control? How would you decide what is needed, which part of the control system to use in a given situation and for how long, and what priority to assign to it?*

I hope you had no hard-and-fast answers to the above questions, because there aren't any, in general. These are some of the biggest challenges in robot control, and we are going to learn how to deal with them in this and the next few chapters.

11.1 Who Needs Control Architectures?

Just putting rules or programs together does not result in well-behaved robots, through it may be a lot of fun, as long as the robots are small and not dangerous. While there are numerous different ways in which a robot's controller

can be programmed (and there are infinitely many possible robot control programs), most of them are pretty bad, ranging from completely incorrect to merely inefficient. To find a good (correct, efficient, even optimal) way to control a given robot for a given task, we need to know some guiding principles of robot control and the fundamentally different ways in which robots can be programmed. These are captured in robot control architectures.

CONTROL ARCHITECTURE

A robot *control architecture* provides guiding principles and constraints for organizing a robot's control system (its brain). It helps the designer to program the robot in a way that will produce the desired overall output behavior.

The term *architecture* is used here in the same way as in "computer architecture", where it means the set of principles for designing computers out of a collection of well-understood building blocks. Similarly, in robot architectures, there is a set of building blocks or tools at your disposal to make the job of robot control design easier. Robot architectures, like computer architectures, and of course "real" building architectures, where the term originally comes from, use specific styles, tools, constraints, and rules.

To be completely honest, you don't have to know anything about control architectures to get a robot to do something. So why should we bother with this and the next few chapters? Because there is more to robotics than getting simple robots to do simple things. If you are interested in how you can get a complex robot (or a team of robots) to do something useful and robust in a complicated environment, knowing about control architectures is necessary to help you get you there. Trial and error and intuition go only so far, and we'd like to take robotics very far indeed, far beyond that.

You might be wondering exactly what we mean by robot control, since so far we have seen that it involves hardware, signal processing, and computation. That's true in general: the robot's "brain" can be implemented with a conventional program running on a microprocessor, or it may be embedded in hardware, or it may be a combination of the two. The robot's controller does not need to be a single program on a single processor. In most robots it is far from that, since, as we have seen, there is a lot going on in a robot. Sensors, actuators, and decisions need to interact in an effective way to get the robot to do its job, but there is typically no good reason to have all those elements controlled from a single centralized program, and there are many reasons not to.

Can you think of some of those reasons?

Robustness to failure, for one thing; if the robot is controlled in a centralized fashion, the failure of that one processor makes the whole robot stop working. But before we think about how to spread out the different parts of the robot's brain, let's get back to designing robot control programs, wherever on the robot they may be running.

> *Robot control can take place in hardware and in software, but the more complex the controller is, the more likely it is to be implemented in software. Why?*

Hardware is good for fast and specialized uses, and software is good for flexible, more general programs. This means that complicated robot brains typically involve computer programs of some type or another, running on the robot in real time. The brain should be physically on the robot, but that may be just prejudice based on how biological systems do it. If wireless radio communication is reliable enough, some or all of the processing could reside off the robot. The trouble is that communication is never perfectly reliable, so it is much safer to keep your brain with you at all times (keep your head on your shoulders), for robots and people.

Brains, robotic or natural, use their programs to solve problems that stand in the way of achieving their goals and getting their jobs done. The process of solving a problem using a finite (not endless) step-by-step procedure is called an *algorithm*, and is named for the Iranian mathematician, Al-Khawarizmi (if that does not seem similar at all, it may be because we are not pronouncing his name correctly). The field of computer science devotes a great deal of research to developing and analyzing algorithms for all kinds of uses, from sorting numbers to creating, managing, and sustaining the Internet. Robotics also concerns itself with the development and analysis of algorithms (among many other things, as we will learn) for uses that are relevant to robots, such as navigation (see Chapter 19), manipulation (see Chapter 6), learning (see Chapter 21), and many others (see Chapter 22). You can think of algorithms as the structure on which computer programs are based.

ALGORITHM

11.2 Languages for Programming Robots

So robot brains are computer programs and they are written using programming languages. You may be wondering what the best robot programming language is. Don't waste your time: there is no "best" language. Robot programmers use a variety of languages, depending on what they are trying

to get the robot to do, what they are used to, what hardware comes with the robot, and so on. The important thing is that robot controllers can be implemented in various languages.

Any programming language worth its beans is called "Turing universal," which means that, in theory at least, it can be used to write any program. This concept was named after Alan Turing, a famous computer scientist from England who did a great deal of foundational work in the early days of computer science, around World War II. To be Turing universal, a programming language has to have the following capabilities: sequencing (a then b then c), conditional branching (if a then b else c), and iteration (for a=1 to 10 do something). Amazingly, with just those, any language can compute everything that is computable. Proving that, and explaining what it means for something to be computable, involves some very nice formal computer science theory that you can learn about from the references at the end of the chapter.

The good news from all that theory is that programming languages are just tools; you can use them to program various things, from calculators to airline scheduling to robot behaviors, and everything in between. You can use any programming language to write any program, at least in theory. In practice, of course, you should be smart and picky, and choose the language that is well suited to the programming task. (You have no doubt noticed this repeating theme about finding suitable designs for robot sensors and bodies and controllers, and suitable languages for programming the robots, and so on. It's a basic principle of good engineering of any kind.)

If you have programmed even a little bit, you know that programming languages come in a great variety and are usually specific for particular uses. Some are good for programming Web pages, others for games, and still others for programming robots, which is what we really care about in this book. So although there is no best language for programming robots, some languages are better for it than others. As robotics is growing and maturing as a field, there are more and more specialized programming languages and tools.

Various programming languages have been used for robot control, ranging from general-purpose to specially designed ones. Some languages have been designed specifically to make the job of programming particular robot control architectures easier. Remember, architectures exist to provide guiding principles for good robot control design, so it is particularly convenient if the programming language can make following those principles easy and failing to follow them hard.

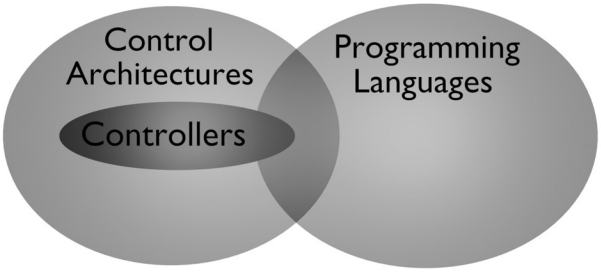

Figure 11.1 A way to visualize the relationship among control architectures, controllers, and programming languages.

Figure 11.1 shows a way in which you can think about the relationship among control architectures, robot controllers, and programming languages. They are all different, yet are needed to make a robot do what you'd like it to do.

11.3 And the Architectures are...

Regardless of which language is used to program a robot, what matters is the control architecture used to implement the controller, because not all architectures are the same. On the contrary, as you will see, architectures impose strong rules and constraints on how robot programs are structured, and the resulting control software ends up looking very different.

We have already agreed that there are numerous ways you can program a robot, and there are fewer, but still many, ways in which you can program a robot well. Conveniently, all those effective robot programs fall into one of the known types of control architectures. Better still, there are very few types of control (that we know of so far). They are:

1. Deliberative control

2. Reactive control
3. Hybrid control
4. Behavior-based control.

In the next few chapters we will study each of these architectures in detail, so you will know what's good and bad about them and which one to choose for a given robot and task.

Because robotics is a field that encompasses both engineering and science, there is a great deal of ongoing research that brings new results and discoveries. At robotics conferences, there are presentations, full of drawings of boxes and arrows, describing architectures that may be particularly well suited for some robot control problem. That's not about to go away, since there are so many things robots could do, and so much yet to be discovered. But even though people may keep discovering new architectures, all of them, and all the robot programs, fit into one of the categories listed above, even if the programmer may not realize it.

In most cases, it is impossible to tell, just by observing a robot's behavior, what control architecture it is using. That is because multiple architectures can get the job done, especially for simple robots. As we have said before, when it comes to more complex robots, the control architecture becomes very important.

Before we launch into the details of the different types of control architectures and what they are good for, let's see what are some important questions to consider that can help us decide which architecture to use. For any robot, task, and environment, many things need to be considered, including these:

- Is there a lot of sensor noise?
- Does the environment change or stay static?
- Can the robot sense all the information it needs? If not, how much can it sense?
- How quickly can the robot sense?
- How quickly can the robot act?
- Is there a lot of actuator noise?
- Does the robot need to remember the past in order to get the job done?

- Does the robot need to think into the future and predict in order to get the job done?
- Does the robot need to improve its behavior over time and be able to learn new things?

Control architectures differ fundamentally in the ways they treat the following important issues:

- Time: How fast do things happen? Do all components of the controller run at the same speed?
- Modularity: What are the components of the control system? What can talk to what?
- Representation: What does the robot know and keep in its brain?

Let's talk about each of these briefly. We will spend more time understanding them in detail in the subsequent chapters.

11.3.1 Time

TIME-SCALE

Time, usually called *time-scale*, refers to how quickly the robot has to respond to the environment compared with how quickly it can sense and think. This is a key aspect of control and therefore a major influence on the choice of what architecture should be used.

The four basic architecture types differ significantly in how they treat time.

DELIBERATIVE CONTROL
REACTIVE CONTROL

Deliberative control looks into the future, so it works on a long time-scale (how long depends on how far into the future it looks). In contrast, *reactive control* responds to the immediate, real-time demands of the environment without looking into the past or the future, so it works on a short time-scale.

HYBRID CONTROL

Hybrid control combines the long time-scale of deliberative control and the short time-scale of reactive control, with some cleverness in between. Finally,

BEHAVIOR-BASED CONTROL

behavior-based control works to bring the time-scales together. All of this will make more sense as we spend time getting to know each type of control and the associated architectures.

11.3.2 Modularity

MODULARITY

Modularity refers to the way the control system (the robot's program) is broken into pieces or components, called modules, and how those modules interact with each other to produce the robot's overall behavior. In *deliberative*

control, the control system consists of multiple modules, including sensing (perception), planning, and acting, and the modules do their work in sequence, with the output of one providing the input for the next. Things happen one at a time, not at the same time, as we will see in Chapter 13. In *reactive control*, things happen at the same time, not one at a time. Multiple modules are all active in parallel and can send messages to each other in various ways, as we will learn in Chapter 14. In *hybrid control*, there are three main modules to the system: the deliberative part, the reactive part, and the part in between. The three work in parallel, at the same time, but also talk to each other, as we will learn in Chapter 15. In *behavior-based control*, there are usually more than three main modules that also work in parallel and talk to each other, but in a different way than in hybrid systems, as we will cover in Chapter 16. So as you can see, how many modules there are, what is in each module, whether the modules work sequentially or in parallel, and which modules can talk to which others are distinguishing features of control architectures.

11.3.3 Representation

Finally, *representation*. That's a tough one to summarize, so we'll give it a whole chapter, coming up next, to do it justice.

> *So how do we know which architecture to use for programming the controller of a particular robot?*

Let's learn more about the different architectures so we can answer that question. After talking about representation, in the next few chapters we will study each of the four basic architecture types: deliberative, reactive, hybrid, and behavior-based.

To Summarize

- Robot control can be done in hardware and/or in software. The more complex the robot and its task, the more software control is needed.
- Control architectures provide guiding principles for designing robot programs and control algorithms.
- There is no one best robot programming language. Robots can be programmed with a variety of languages, ranging from general-purpose to special-purpose.

- Special-purpose robot programming languages can be written to facilitate programming robots in particular control architectures.
- Robot control architectures differ substantially in how they handle time, modularity, and representation.
- The main robot control architectures are deliberative (not used), reactive, hybrid, and behavior-based.

Food for Thought

- How important is the programming language? Could it make or break a particular gadget, device, or robot?
- With the constant development of new technologies that use computation, do you think there will be increasingly more or increasingly fewer programming languages?

Looking for More?

- *Introduction to the Theory of Computation* by Michael Sipser is an excellent textbook for learning about the very abstract but very elegant topics only touched on in this chapter.
- One of the most widely used textbooks for learning about programming languages is *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman. It's actually fun to read, although not necessarily easy to understand.

12 *What's in Your Head? Representation*

In many tasks and environments, the robot cannot immediately sense everything it needs to know. It is therefore sometimes useful to remember what happened in the past, or to try to predict what will happen in the future. It is also sometimes useful to store maps of the environment, images of people or places, and various other information that will be useful for getting the job done.

REPRESENTATION

Representation is the form in which information is stored or encoded in the robot.

MEMORY

Representation is more than memory. In computer science and robotics, we think of *memory* as the storage device used to keep information. Just referring to memory does not say anything about what is stored and how it is encoded: is it in the form of numbers, names, probabilities, x,y locations, distances, colors? Representation is what encodes those important features of what is in the memory.

What is represented and how it is represented has a major impact on robot control. This is not surprising, as it is really the same as saying "What is in your brain influences what you can do." In this chapter we will learn about what representation is all about and why it has such an important role in robot control.

> *How does internal state, the information a robot system keeps around (remember Chapter 3), relate to representation? Is it the same thing? If not, what is it?*

In principle, any internal state is a form of representation. In practice, what matters is the form and function of that representation, how it is stored and how it is used. "Internal state" usually refers to the "status" of the system itself, whereas "representation" refers to arbitrary information about the

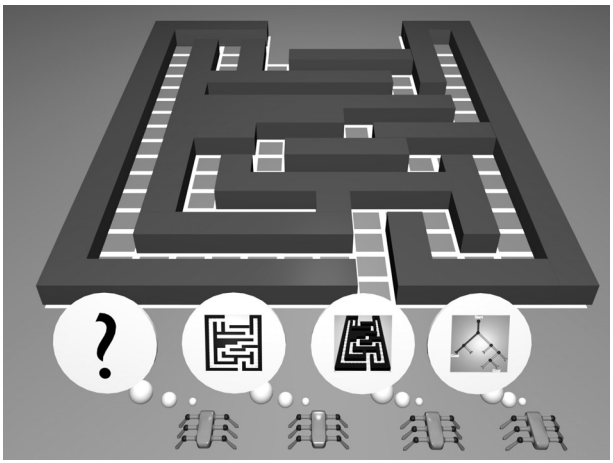

Figure 12.1 A few possible representation options for a maze-navigating robot.

world that the robot stores.

12.1 The Many Ways to Make a Map

WORLD MODEL

Representation of the world is typically called a *world model*. A map (as used in navigation; see Chapter 19) is the most commonly used example of a world model. To give an illustration of how the representation of a particular world, its map, can vary in form, consider the problem of exploring a maze.

What can the robot store/remember to help it navigate a maze?

- The robot may remember the exact path it has taken to get to the end of the maze (e.g., "Go straight 3.4 cm, turn left 90 degrees, go straight 12 cm, turn right 90 degrees."). This remembered path is a type of map for getting through the maze. This is an odometric path.
- The robot may remember a sequence of moves it made at particular landmarks in the environment (e.g., "Left at the first junction, right at the second junction, straight at the third."). This is another way to store a path through the maze. This is a landmark-based path.

- The robot may remember what to do at each landmark in the maze (e.g., "At the green/red junction go left, at the red/blue junction go right', at the blue/orange junction go straight."). This is a landmark-based map; it is more than a path since it tells the robot what to do at each junction, no matter in what order it reaches the junction. A collection of landmarks connected with links is called a *topological map* because it describes the *topology*, the connections, among the landmarks. Topological maps are very useful. (Don't confuse them with *topographic maps*, which are something completely different, having to do with representing the elevation levels of the terrain.)

TOPOLOGICAL MAP

- The robot may remember a map of the maze by "drawing it" using exact lengths of corridors and distances between walls it sees. This is a metric map of the maze, and it is also very useful.

The above are not nearly all the ways in which the robot can construct and store a model of the maze. However, the four types of models above already show you some important differences in how representations can be used. The first model, the odometric path, is very specific and detailed, and that it is useful only if the maze never changes, no junctions become blocked or opened, and if the robot is able to keep track of distances and turns very accurately. The second approach also depends on the map not changing, but it does not require the robot to be as specific about measurements, because it relies on finding landmarks (in this case, junctions). The third model is similar to the second, but connects the various paths into a landmark-based map, a network of stored landmarks and their connections to one another. Finally, the fourth approach is the most complicated because the robot has to measure much more about the environment and store much more information. On the other hand, it is also the most generally useful, since with it the robot can now use its map to think about other possible paths in case any junctions become blocked.

Figure 12.1 illustrates a few more possible representations that could be used by a maze-following robot: no representation at all, a typical 2D map, a visual image of the environment, and a graph of the maze structure.

12.2 What Can the Robot Represent?

Maps are just one of the many things a robot may want to represent or store or model in its "brain." For example, the robot may want to remember how

long its batteries last, and to remind itself to recharge them before it is too late. This is a type of a self-model. Also, the robot may want to remember that traffic is heavy at particular times of day or in particular places in the environment, and to avoid traveling at those times and in those places. Or it may store facts about other robots, such as which ones tend to be slow or fast, and so on.

There are numerous aspects of the world that a robot can represent and model, and numerous ways in which it can do it. The robot can represent information about:

- Self: stored proprioception, self-limitations, goals, intentions, plans
- Environment: navigable spaces, structures
- Objects, people, other robots: detectable things in the world
- Actions: outcomes of specific actions in the environment
- Task: what needs to be done, where, in what order, how fast, etc.

You can already see that it could take quite a bit of sensing, computation, and memory to acquire and store some types of world models. Moreover, it is not enough simply to get and store a world model; it is also necessary to keep it accurate and updated, or it becomes of little use or, worse yet, becomes misleading to the robot. Keeping a model updated takes sensing, computation, and memory.

Some models are very elaborate; they take a long time to construct and are therefore kept around for the entire lifetime of the robot's task. Detailed metric maps are such models. Other models, on the other hand, may be relatively quickly constructed, briefly used, and soon discarded. A snapshot of the robot's immediate environment showing the path out of the clutter through the nearest door is an example of such a short-term model.

12.3 Costs of Representing

In addition to constructing and updating a representation, using a representation is also not free, in terms of computation and memory cost. Consider maps again: to find a path from a particular location in the map to the goal location, the robot must plan a path. As we will learn in more detail in Chapter 19, this process involves finding all the free/navigable spaces in the map, then searching through those to find any path, or the best path, to the goal.

Because of the processing requirements involved in the construction, maintenance, and use of representation, different architectures have very different properties based on how representation is handled, as we will see in the next few chapters. Some architectures do not facilitate the use of models (or do not allow them at all, such as reactive ones), others utilize multiple types of models (hybrid ones), still others impose constraints on the time and space allowed for the models being used (behavior-based ones).

How much and what type of representation a robot should use depends on its task, its sensors, and its environment. As we will see when we learn more about particular robot control architectures, how representation is handled and how much time is involved in handling it turns out to be a crucial distinguishing feature for deciding what architecture to use for a given robot and task. So, as you would imagine, what is in the robot's head has a very big influence on what that robot can do.

To Summarize

- Representation is the form in which information is stored in the robot.
- Representation of the world around the robot is called a world model.
- Representation can take a great variety of forms and can be used in a great variety of ways by a robot.
- A robot can represent information about itself, other robots, objects, people, the environment, tasks, and actions.
- Representations require constructing and updating, which have computational and memory costs for the robot.
- Different architectures treat representation very differently, from not having any at all, to having centralized world models, to having distributed ones.

Food for Thought

- Do you think animals use internal models? What about insects?
- Why might you not want to store and use internal models?

13 *Think Hard, Act Later Deliberative Control*

Deliberation refers to thinking hard; it is defined as "thoughtfulness in decision and action." Deliberative control grew out of early artificial intelligence (AI). As you remember from the brief history of robotics (Chapter 2), in those days, AI was one of the main influences on how robotics was done.

In AI, deliberative systems were (and sometimes still are) used for solving problems such as playing chess, where thinking hard is exactly the right thing to do. In games, and in some real-world situations, taking time to consider possible outcomes of several actions is both affordable (there is time to do it) and necessary (without strategy, things go bad). In the 1960s and 1970s, AI researchers were so fond of this type of reasoning, they theorized that the human brain works this way, and so should robot control. As you recall from Chapter 2, in the 1960s, early AI-based robots often used vision sensors, which require a great deal of processing, and so it was worth it for a robot to take the time (quite a lot of time back then, with those slow processors) to think hard about how to act given how difficult it was to make sense out of the environment. The robot Shakey, a forerunner of many AI-inspired robotics projects that followed, used the then state of the art in machine vision as input into a planner in order to decide what to do next, how and where to move.

13.1 What Is Planning?

So what is planning?

PLANNING

Planning is the process of looking ahead at the outcomes of the possible actions, and searching for the sequence of actions that will reach the desired

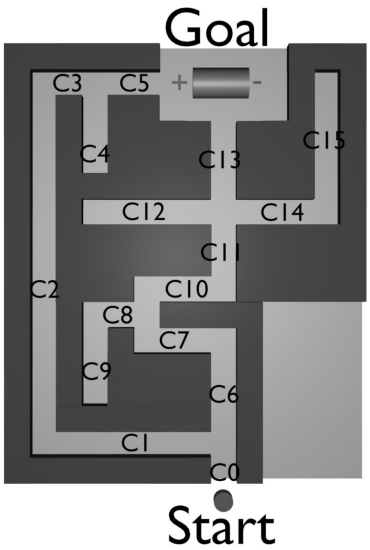

Figure 13.1 A closeup of a maze the robot (marked with a black circle) needs to navigate.

goal.

SEARCH *Search* is an inherent part of planning. It involves looking through the available representation "in search of" the goal state. Sometimes searching the complete representation is necessary (which can be very slow, depending on the size of the representation), while at other times only a partial search is enough, to reach the first found solution.

For example, if a robot has a map of a maze, and knows where it is and where it wants to end up (say at a recharging station at the end of the maze), it can then plan a path from its current position to the goal, such as those shown in figure 13.2. The process of searching through the maze happens in the robot's head, in the representation of the maze (such as the one shown in figure 13.1, not in the physical maze itself. The robot can search from the goal backwards, or from where it is forward. Or it can even search in

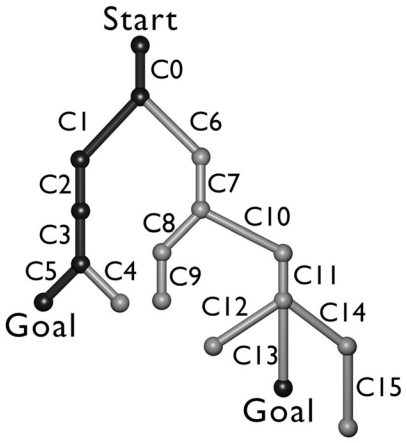

Figure 13.2 A graph of the maze, showing the two possible paths to the charger. One of the paths is shorter/more optimal than the other.

both directions in parallel. That is the nice thing about using internal models or representations: you can do things that you can't do in the real world. Consider the maze in figure 13.1, where the robot is marked with a black circle, and the goal with the recharging battery. Note that at each junction in the maze, the robot has to decide how to turn.

The process of planning involves the robot trying different turns at each junction, until a path leads it to the battery. In this particular maze, there is more than one path to the goal from where the robot is, and by searching the entire maze (in its head, of course) the robot can find both of those paths, and then choose the one it likes better. Usually the shortest path is considered the best, since the robot uses the least time and battery power to reach it. But in some cases other criteria may be used, such as which path is the safest or the least crowded. The process of improving a solution to a problem by finding a better one is called *optimization*. As in the maze example above, various values or properties of the given problem can be optimized, such as the distance of a path; those values are called *optimization criteria*. Usually some optimization criteria conflict (for example, the shortest path may also be the most crowded), so deciding what and how to optimize is not simple. *Optimizing search* looks for multiple solutions (paths, in the case of the maze),

OPTIMIZATION

OPTIMIZATION CRITERIA

OPTIMIZING SEARCH

in some cases all possible paths.

In general, in order to use search and plan a solution to a particular problem, it is necessary to represent the world as a set of states. In the maze example, the states are the corridors, the junctions, the start (where the search starts), the current state (wherever the robot happens to be at the time), and the goal (where the robot wants to end up). Then search is performed to find a path that can take the robot from the current state to the goal state. If the robot wants to find the very best, optimal path, it has to search for *all possible paths* and pick the one that is optimal based on the selected optimization criterion (or criteria, if more than one is used).

13.2 Costs of Planning

For small mazes like the one used here, planning is easy, because the state space is small. But as the number of possible states becomes large (as in chess, for example), planning becomes slow. The longer it takes to plan, the longer it takes to solve the problem. In robotics, this is particularly important since a robot must be able to avoid immediate danger, such as collisions with objects. Therefore, if path planning takes too long, the robot either has to stop and wait for planning to finish before moving on, or it may risk collisions or running into blocked paths if it forges ahead without a finished plan. This is not just a problem in robotics, of course; *whenever a large state space is involved, planning is difficult.* To deal with this fundamental problem, AI researchers have found various ways to speed things up. One popular approach is to use hierarchies of states, where first only a small number of "large," "coarse," or "abstract" states is considered; after that, more refined and detailed states are used in the parts of the state space where it really matters. The graph of the maze shown in figure 13.2 is an example of doing exactly that: it lumps all states within a corridor of the maze together and considers them as a single corridor state in the graph. There are several other clever methods besides this one that can speed up search and planning. These are all optimization methods for planning itself, and they always involve some type of compromise.

Since the early days of AI, computing power has greatly improved and continues to improve (as per Moore's Law; see the end of the chapter for more information). This means larger state spaces can be searched much faster than ever before. However, there is still a limit to what can be done in *real time*, the time in which a physical robot moves around in a dynamic

REAL TIME

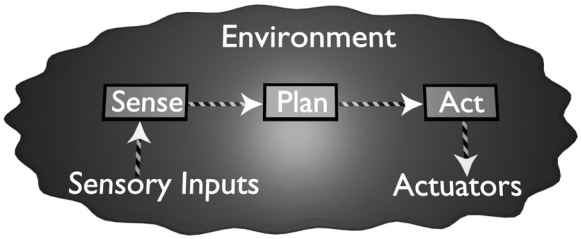

Figure 13.3 A diagram of a classical deliberative architecture, showing the sequence of sense-plan-act (SPA) components.

environment.

In general, a real robot cannot afford to just sit and deliberate. Why?

Deliberative, planner-based architectures involve three steps that need to be performed in sequence:

1. Sensing (S)
2. Planning (P)
3. Acting (A), executing the plan.

SPA (SENSE-PLAN-ACT) ARCHITECTURES

For this reason, deliberative architectures, which are also called *SPA (sense-plan-act) architectures*, shown in figure 13.3, have serious drawbacks for robotics. Here is what they are.

Drawback 1: Time-Scale

As we have said above, it can take a very long time to search in large state spaces. Robots typically have collections of sensors: some simple digital sensors (e.g., switches, IRs), some more complex ones (e.g., sonars, lasers,

cameras), some analog sensors (e.g., encoders, gauges). The combined inputs from these sensors in themselves constitute a large state space. When combined with internal models or representations (maps, images of locations, previous paths, etc.), the result is a state space that is large and slow to search.

If the planning process is slow compared with the robot's moving speed, it has to stop and wait for the plan to be finished, in order to be safe. So to make progress, it is best to plan as rarely as possible and move as much as possible in between. This encourages open loop control (see Chapter 10), which we know is a bad idea in dynamic environments. If planning is fast, then execution need not be open loop, since replanning can be done at each step; unfortunately, typically this is impossible for real-world problems and robots.

Generating a plan for a real environment can be very slow.

Drawback 2: Space

It may take a great deal of space (i.e., memory storage) to represent and manipulate the robot's state space representation. The representation must contain all information needed for planning and optimization, such as distances, angles, snapshots of landmarks, views, and so on. Computer memory is comparatively cheap, so space is not as much of a problem as time, but all memory is finite, and some algorithms can run out of it.

Generating a plan for a real environment can be very memory-intensive.

Drawback 3: Information

The planner assumes that the representation of the state space is accurate and up to date. This is a reasonable assumption, because if the representation is not accurate and updated, the resulting plan is useless. For example, if a junction point in the maze is blocked, but the robot's internal map of the maze does not show it as blocked, the planned path might go through that junction and would therefore be invalid, but the robot would not discover that until it physically got to the junction. Then it would have to backtrack to a different path. The representation used by the planner must be updated and checked as often as necessary to keep it sufficiently accurate for the task. Thus, the more information the better.

Generating a plan for a real environment requires updating the world model, which takes time.

Drawback 4: Use of Plans

In addition to all of the above, any accurate plan is useful only if:

- The environment does not change during the execution of the plan in a way that affects the plan
- The robot knows what state of the world and of the plan it is in at all times
- The robot's effectors are accurate enough to execute each step of the plan in order to make the next step possible.

Executing a plan, even when one is available, is not a trivial process.

All of the above challenges of deliberative, SPA control became obvious to the early roboticists of the 1960s and 1970s, and they grew increasingly dissatisfied. As we learned in Chapter 2, in the early 1980s they proposed alternatives: reactive, hybrid, and behavior-based control, all of which are in active use today.

What happened to purely deliberative systems?

As a result of robotics work since the 1980s, purely deliberative architectures are no longer used for the majority of physical robots, because the combination of real-world sensors, effectors, and time-scale challenges outlined above renders them impractical. However, there are exceptions, because some applications demand a great deal of advance planning and involve no time pressure, while at the same time presenting a static environment and low uncertainty in execution. Such domains of application are extremely rare, but they do exist. Robot surgery is one such application; a perfect plan is calculated for the robot to follow (for example, in drilling the patient's skull or hip bone), and the environment is kept perfectly static (literally by attaching the patient's body part being operated on to the operating table with bolts), so that the plan remains accurate and its execution precise.

Pure deliberation is alive and well in other uses outside of robotics as well, such as AI for game-playing (chess, Go, etc.). In general, whenever the world is static, there is enough time to plan, and there is no state uncertainty, it is a good thing to do.

What about robotics problems that require planning?

The SPA approach has not been abandoned in robotics; it has been expanded. Given the fundamental problems with purely deliberative approaches, the following improvements have been made:

- Search/planning is slow, so save/cache important and/or urgent decisions
- Open loop plan execution is bad, so use closed loop feedback, and be ready to respond or replan when the plan fails.

In Chapter 15 we will see how the SPA model is currently incorporated into modern robots in a useful way.

To Summarize

- Deliberative architectures are also called SPA architectures, for sense-plan-act.
- They decompose control into functional modules which perform different and independent functions (e.g., sense-world, generate-plan, translate-plan-into-actions).
- They execute the functional modules sequentially, using the outputs of one as inputs of the next.
- They use centralized representation and reasoning.
- They may require extensive, and therefore slow, reasoning computation.
- They encourage open loop execution of the generated plans.

Food for Thought

- Can you use deliberative control without having some internal representation?
- Can animals plan? Which ones, and what do they plan?
- If you had perfect memory, would you still need to plan?

Looking for More?

- The Robotics Primer Workbook exercises for this chapter are found here: http://roboticsprimer.sourceforge.net/workbook/Deliberative_Control
- In 1965, Gordon Moore, the co-founder of Intel, made the following important observation: the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted that this trend would continue for the foreseeable future. In subsequent years, the pace slowed down a bit, to about every eighteen months. So Moore's Law states that the number of transistors per square inch on integrated circuits will double about every eighteen months until at least the year 2020. This law has an important impact on the computer industry, robotics, and even global economics.

14 *Don't Think, React! Reactive Control*

REACTIVE SYSTEMS

Reactive control is one of the most commonly used methods for robot control. It is based on a tight connection between the robot's sensors and effectors. Purely *reactive systems* do not use any internal representations of the environment, and do not look ahead at the possible outcomes of their actions: they operate on a short time-scale and react to the current sensory information.

Reactive systems use a direct mapping between sensors and effectors, and minimal, if any, state information. They consist of collections of rules that couple specific situations to specific actions, as shown in figure 14.1. You can think of reactive rules as being similar to *reflexes*, innate responses that do not involve any thinking, such as jerking back your hand after touching a hot stove. Reflexes are controlled by the neural fibers in the spinal cord, not the brain. This is so that they can be very fast; the time it takes for a neural signal to travel from the potentially burned finger that touched the hot stove to the brain and back, and the computation in between to decide what to do takes too long. To ensure a fast reaction, reflexes don't go all the way to the brain, but only to the spinal cord, which is much more centrally located to most areas of the body. Reactive systems are based on exactly the same principle: complex computation is removed entirely in favor of fast, stored precomputed responses.

Reactive systems consist of a set of situations (stimuli, also called conditions) and a set of actions (responses, also called actions or behaviors). The situations may be based on sensory inputs or on internal state. For example, a robot may turn to avoid an obstacle that is sensed, or because an internal clock indicated it was time to change direction and go to another area. These examples are very simple; reactive rules can be much more complex, involving arbitrary combinations of external inputs and internal state.

The best way to keep a reactive system simple and straightforward is to

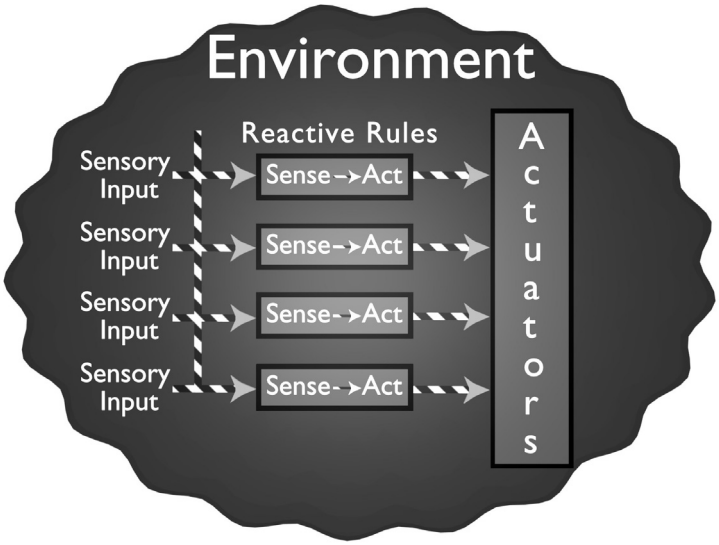

Figure 14.1 A diagram of a reactive architecture, showing the parallel, concurrent task-achieving modules.

have each unique situation (state) that can be detected by the robot's sensors trigger only one unique action of the robot. In such a design, the conditions are said to be *mutually exclusive*, meaning they exclude one another; only one can be true at a time.

MUTUALLY EXCLUSIVE CONDITIONS

However, it is often too difficult to split up all possible situations (world states) in this way, and doing so may require unnecessary encoding. To ensure mutually exclusive conditions, the controller must encode rules for all possible sensory input combinations. Recall from Chapter 3 that all those combinations, when put together, define the robot's sensor space. For a robot with a 1-bit such as a switch, that is total of two possibilities (on and off), for a robot with two switches, the total is four possibilities, and so on. As the sensors grow in complexity and number, the combinatorial space of all possible sensory inputs, i.e., the sensor space, quickly becomes unwieldy. In computer science, AI, and in robotics, the formal term for "unwieldy" is *intractable*. To encode and store such a large sensor space would take a giant lookup table, and searching for entires in such a giant table would be slow,

INTRACTABLE

unless some clever parallel lookup technique is used.

So to do a complete reactive system, the entire state space of the robot (all possible external and internal states) should be uniquely coupled or mapped to appropriate actions, resulting in the complete control space for the robot.

The design of the reactive system, then, is coming up with this complete set of rules. This is done at "design-time," not at "run-time," when the robot is active. This means it takes a lot of thinking by the designer (as any robot design process should), but no thinking by the robot (in contrast to deliberative control, where there is a lot of thinking done by the system).

In general, complete mappings between the entire state space and all possible responses are not used in manually-designed reactive systems. Instead, the designer/programmer identifies the important situations and writes the rules for those; the rest are covered with default responses. Let's see how that is done.

Suppose that you are asked to write a reactive controller that will enable a robot to move around and avoid obstacles. The robot has two simple whiskers, one on the left and one on the right. Each whisker returns 1 bit, "on" or "off"; "on" indicates contact with a surface (i.e., the whisker is bent). A simple reactive controller for wall-following using those sensors would look like this:

```
If left whisker bent, turn right.
If right whisker bent, turn left.
If both whiskers bent, back up and turn to the left.
Otherwise, keep going.
```

In the above example there are only four possible sensory inputs over all, so the robots' sensor space is four, and therefore there are four reactive rules. The last rule is a default, although it covers only one possible remaining case.

> *A robot using the above controller could oscillate if it gets itself into a corner where the two whiskers alternate in touching the walls. How might you get around this rather typical problem?*

There are two popular ways:

1. *Use a little randomness:* When turning, choose a random angle instead of a fixed one. This introduces variety into the controller and prevents it from getting permanently stuck in an oscillation. In general, adding a bit of randomness avoids any permanent stuck situation. However, it could

Figure 14.2 Sonar ring configuration and the navigation zones for a mobile robot.

still take a long time to get out of a corner.

2. *Keep a bit of history:* Remember the direction the robot turned in the previous step (1 bit of memory) and turn in the same direction again, if the situation occurs again soon. This keeps the robot turning in one direction and eventually gets it out of the corner, instead of getting it into an oscillation. However, there are environments in which this may not work.

Now suppose that instead of just two whiskers, your robot has a ring of sonars (twelve of them, to cover the 360-degree span, as you learned in Chapter 9). The sonars are labeled from 1 to 12. Sonars 11, 12, 1 and 2 are at the front of the robot, sonars 3 and 4 are on the right side of the robot, sonars 6 and 7 are in the back, and sonars 1 and 10 are on the left; sonars 5 and 8 also get used, don't worry. Figure 14.2) illustrates where the sonars are on the body of the robot. That is a whole lot more sensory input than two whiskers, which allows you to create a more intelligent robot by writing a whole lot more reactive rules.

> *But just how many rules do you need to have, and which ones are the right ones?*

You could consider each sonar individually, but then you would have a lot

of combinations of sonars to worry about and a lot of possible sonar values (from 0 to 32 feet each, recalling the range of Polaroid sonars from Chapter 9). In reality, your robot does not really care what each of the sonars individually returns; it is more important to focus on any really short sonar values (indicating that an obstacle is near) and specific areas around the robot (e.g., the front, the sides, etc.).

Let's start by defining just two distance areas around the robot:

1. Danger-zone: short readings, things are too close
2. Safe-zone: reasonable readings, good for following edges of objects, but not too far from things

Now let's take those two zones and write a reactive controller for the robot which considers groups of sonars instead of individual sonars:

```
(case
  (if (minimum (sonars 11 12 1 2))     ⇐ danger-zone
        and
       (not stopped)
  then
    stop)
(if ((minimum (sonars 11 12 1 2))     ⇐ danger-zone
         and
       stopped)
then
  move backward)
(otherwise
  move forward))
```

The controller above consists of two reactive rules. As we have learned, it typically takes more than a single rule to get something done.

The controller stops the robot if it is too near an object in front of it, detected as the shortest reading of the front four sonars being in the danger-zone. If the robot is already stopped, and the object is too near (in the danger-zone), it backs up. The result is safe forward movement.

The controller does not check the sides and the back of the robot, however. If we assume that the environment the robot is in is static (furniture, walls) or that it contains rational people who keep their eyes open and do not run into the robot from the side or back, this controller will do just fine. But if the environment includes entities that are not very safe (children, other robots,

hurried people), then it is in fact necessary to check all the sonars, not just the front four.

Moving forward and stopping for obstacles is not all we want from our robot, so let's add another controller, which we can consider another layer or module in a reactive architecture, to make it better able to get around its environment:

```
(case
  (if ((sonar 11 or 12) <= safe-zone
     and
    (sonar 1 or 2) <= safe-zone)
  then
    turn left)
  (if (sonar 3 or 4) <= safe-zone
  then
    turn right))
```

The above controller makes the robot turn away from detected obstacles. Since safe-zone is larger than danger-zone, this allows the robot to turn away gradually before getting too close to an obstacle and having to be forced to stop, as in the previous controller. If obstacles are detected on both sides, the robot consistently turns to the left, to avoid oscillations.

By combining the two controllers above we get a wandering behavior which avoids obstacles at a safe distance while moving smoothly around them, and also avoids collisions with unanticipated nearby obstacles by stopping and backing up.

14.1 Action Selection

The controllers described above use specific, mutually exclusive conditions (and defaults), so that their outputs are never in conflict, because only a single unique situation/condition can be detected at a time. If the rules are not triggered by mutually exclusive conditions, more than one rule can be triggered by the same situation, resulting in two or more different action commands being sent to the effector(s).

ACTION SELECTION

Action selection is the process of deciding among multiple possible actions or behaviors. It may select only one output action or may combine the actions to produce a result. These two approaches are called arbitration and fusion.

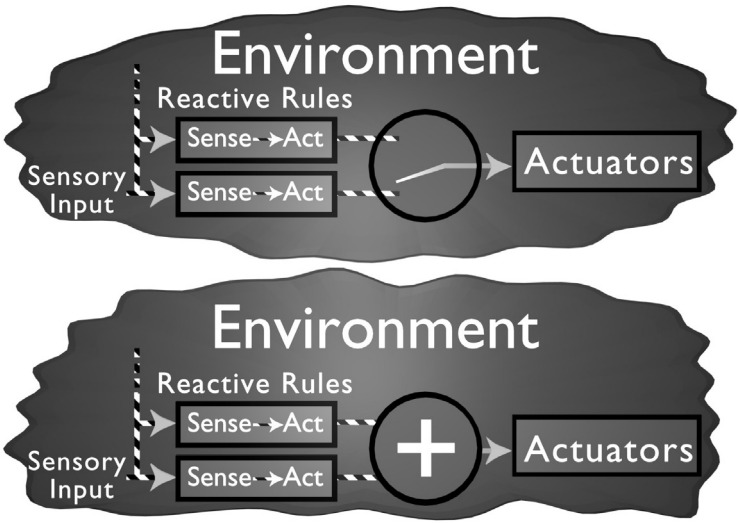

Figure 14.3 Two basic types of action selection: arbitration (top) and fusion (bottom).

COMMAND ARBITRATION

Command arbitration is the process of selecting one action or behavior from multiple candidates.

COMMAND FUSION

Command fusion is the process of combining multiple candidate actions or behaviors into a single output action/behavior for the robot.

The two alternatives are shown in figure 14.3 as part of the generic reactive system.

It turns out that action selection is a major problem in robotics, beyond reactive systems, and that there is a great deal of work (theory and practice) on different methods for command arbitration and fusion. We will learn a great deal more about this interesting problem in Chapter 17.

Although a reactive system may use arbitration to decide which action to execute, and thus execute only one action at a time, it still needs to monitor its rules in parallel, concurrently, in order to be ready to respond to any one or more of them that might be triggered.

Using the example of the wall-following robot, both whiskers have to be monitored all the time, or all sonars have to be monitored all the time, since

an obstacle could appear anywhere, at any time. In the wall-following example, there are very few rules (at least in the whisker case), so given a fast processor, those could be checked sequentially and no real response time would be lost. But some rules may take time to execute. Consider the following controller:

```
If there is no obstacle in front, move forward.
If there is an obstacle in front, stop and turn away.
Start a counter.  After 30 seconds, choose randomly
  between left and right and turn by 30 degrees.
```

What kind of behavior does this controller produce? It produces random wandering with obstacle avoidance (and not very good avoidance at that). Note that the conditions of the three rules above require different amounts of time to compute. The first two check sensory input, while the third uses a timer. If the controller executes the rules sequentially, it would have to wait for 30 seconds in the third rule, before being able to check the first rule again. In that time, the robot could run into an obstacle.

Reactive systems must be able to support parallelism, the ability to monitor and execute multiple rules at once. Practically, this means that the underlying programming language must have the ability to *multitask*, to execute several processes/rules/commands in parallel. The ability to multitask is critical in reactive systems: if a system cannot monitor its sensors in parallel, and instead checks them in sequence, it may miss an event, or at least the onset of an event, and thus fail to react in time.

MULTITASKING

You can begin to see that designing a reactive system for a robot can be quite complicated, since multiple rules, potentially a large number of them, have to be put together in a way that produces effective, reliable, and goal-driven behavior.

> *How do we go about organizing a reactive controller in a principled way? By using architectures that have been specifically designed for that purpose.*

The best known architecture for reactive control is Subsumption Architecture, introduced by Prof. Rodney Brooks at MIT in 1985. It's an oldie by now, but it is still a goodie, finding its way into a vast number of reactive, hybrid, and behavior-based systems.

14.2 Subsumption Architecture

The basic idea behind Subsumption Architecture is to build systems incrementally, from the simple parts to the more complex, all the while using the already existing components as much as possible in the new stuff being added. Here is how it works.

Subsumption systems consist of a collection of modules or layers, each of which achieves a task. For example, they might be move-around, avoid-obstacles, find-doors, visit-rooms, pick-up-soda-cans, and so on. All of the task-achieving layers work at the same time, instead of in sequence. This means the rules for each of them are ready to be executed at any time, whenever the right situation presents itself. As you recall, this is the premise of reactive systems.

The modules or layers are designed and added to the robot incrementally. If we number the layers from 0 up, we first design, implement, and debug layer 0. Let's suppose that layer 0 is move-around, which keeps the robot going. Next we add layer 1, avoid-obstacles, which stops and turns or backs away whenever an obstacle is detected. This layer can already take advantage of the existing layer 0, which moves the robot around, so that, together, layers 0 and 1 result in the robot moving around without running into things it can detect. Next we add layer 2, say find-doors, which looks for doors while the robot roams around safely. And so on, until all of the desired tasks can be achieved by the robot through the combination of its layers.

But there is a twist. Higher layers can also temporarily disable one or more of those below them. For example, avoid-obstacles can stop the robot from moving around. What would result is a robot that sits still, but can turn and back away if somebody approaches it. In general, disabling obstacle avoidance is a dangerous thing to do, and so it is almost never done in any real robot system, but having higher layers selectively disable lower layers within a reactive system is one of the principles of Subsumption Architecture. This manipulation is done in one of only two ways, as shown in figure 14.4:

1. The inputs of a layer/module may be suppressed; this way the module receives no sensory inputs, and so computes no reactions and sends no outputs to effectors or other modules.

2. The outputs of a layer/module may be inhibited; this way the module receives sensory inputs, performs its computation, but cannot control any effectors or other modules.

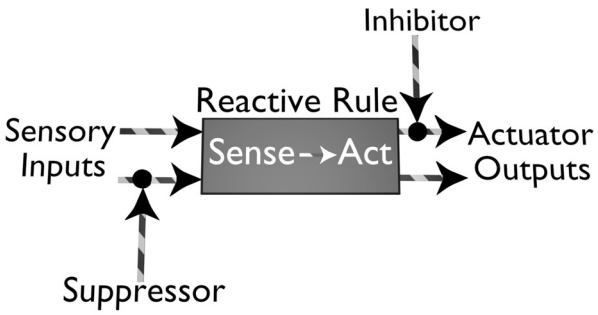

Figure 14.4 Subsumption methods for layer/module interaction: suppression of inputs (left) and inhibition of outputs (right).

SUBSUMPTION ARCHITECTURE

The name *"Subsumption Architecture"* comes from the idea that higher layers can assume the existence of the lower ones and the goals they are achieving, so that the higher layers can use the lower ones to help them in achieving their own goals, either by using them while they are running or by inhibiting them selectively. In this way, higher layers "subsume" lower ones.

There are several benefits to the subsumption style of organizing reactive systems. First, by designing and debugging the system incrementally, we avoid getting bogged down in the complexity of the overall task of the robot. Second, if any higher-level layers or modules of a subsumption robot fail, the lower-level ones will still continue to function unaffected.

BOTTOM-UP

The design of subsumption controllers is called *bottom-up*, because it progresses from the simpler to the more complex, as layers are added incrementally. This is good engineering practice, but its original inspiration came to the inventor of Subsumption Architecture from biology. Brooks was inspired by the evolutionary process, which introduces new abilities based on the existing ones. Genes operate using the process of mixing (crossover) and changing (mutation) of the existing genetic code, so complete creatures are not thrown out and new ones created from scratch; instead, the good stuff that works is saved and used as a foundation for adding more good stuff,

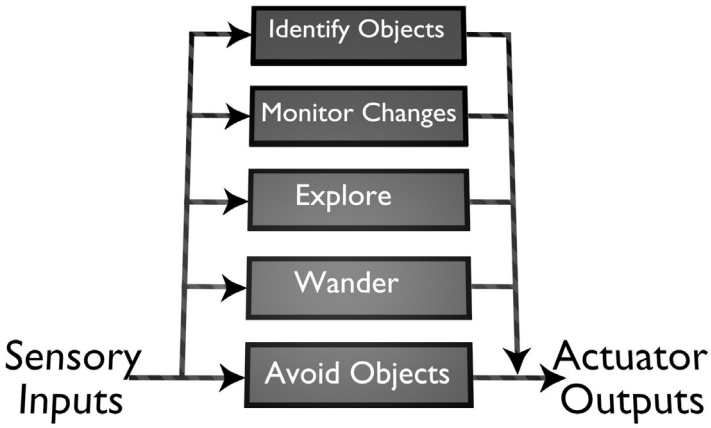

Figure 14.5 A subsumption-based robot control system.

and so complexity grows over time. Figure 14.5 shows an example subsumption control system; it would be constructed bottom-up, adding each layer incrementally.

Building incrementally helps the design and debugging process, and using layers is useful for modularizing the robot controller. This is also good engineering practice. Otherwise, if everything is lumped together, it is hard to design, hard to debug, and hard to change and improve later.

The effectiveness of modularity also depends on not having all modules connected to all others, as that defeats the purpose of dividing them up. So in Subsumption Architecture, the goal is to have very few connections between different layers. The only intended connections are those used for inhibition and suppression. Inside the layers, of course, there are plenty of connections, as multiple rules are put together to produce a task-achieving behavior. Clearly, it takes more than one rule to get the robot to avoid obstacles, to find doors, and so on. But by keeping the rules for the separate tasks apart, the system becomes more manageable to design and maintain.

Therefore, in Subsumption Architecture, we use *strongly coupled* connec-

tions within layers, and *loosely coupled* connections between layers.

> *How do we decide what makes a subsumption layer, and what should go higher or lower?*

Unfortunately, there are no fixed answers to those hard design questions. It all depends on the specifics of the robot, the environment, and the task. There is no strict recipe, but some solutions are better than others, and most of the expertise that robot designers have, is gained through trial and error, basically through sweating it.

14.3 Herbert, or How to Sequence Behaviors Through the World

> *How would you make a reactive robot execute a sequence of behaviors? For example, consider the following task: Search for soda cans, find any empty ones, pick them up, and bring them back to the starting point.*

If you are thinking that you can have one layer or module activate another in a sequence, you are right, but that is not the way to go. Coupling between reactive rules or subsumption layers need not be through the system itself, through explicit communication, but instead through the environment.

A well-known subsumption robot called Herbert, designed by Jonathan Connell (a PhD student of Brooks') in the late 1980s, used this idea to accomplish the task of collecting soda cans. Here is how. Herbert had a layer that moved it around without running into obstacles. (As you will find out, every mobile robot has such a layer/ability/behavior, and it is usually reactive, for the obvious reason that collision-free navigation requires being able to respond quickly.) Herbert's navigation layer used infra red sensors. It also had a layer that used a laser striper and a camera to detect soda cans (by matching them to a particular width in a visual image, quite a clever trick). Herbert also had an arm and a layer of control that could extend the arm, sense if there was a can in its gripper, close the gripper, and pull the arm in. Cleverly, all these arm actions were separate, and were activated not in a sequential, internal way but instead by sensing the environment and the robot directly, like this:

```
If you see something that looks like a soda can
  approach it
If it still looks like a soda can up close
```

```
    then extend the arm
    else turn and go away

  Whenever the arm is extended
  check between the gripper fingers

  Whenever the gripper sensors (IR break beam) detects someth
  close the gripper

  Whenever the gripper is closed and the arm extended
  pull back the arm

  Whenever the arm is pulled back and the gripper closed
go to drop off the can
```

Herbert had a very clever controller; see figure 14.6 for its diagram. It never used any "explicit" sequencing, but instead just checked the different situations in the world (the environment and the robot together) and reacted appropriately. Note, for example, that it was safe for Herbert to close its gripper whenever there was something sensed in it, because the only way it could sense something in the gripper was after it had already looked for cans and extended the arm to reach one. Similarly, it was safe for Herbert to pull the arm back when the gripper was closed, since it was closed only if it was holding a can. And so on.

By the way, in case you are wondering how it knew if the cans were empty or full, it used a simple strain gauge to measure the weight of the can it was holding. It took only the light cans.

Herbert had no internal communication between the layers and rules that achieved can-finding, grabbing, arm-retracting, and moving around, yet it executed everything in the right sequence every time, and was able to perform its can-collecting task. The big idea here is that Herbert was using the sensory inputs as a means of having its rules and layers interact. Brooks called this "interaction through the world."

Another motto, also by Brooks, recommends that we "use the world as its own best model." This is a key principle of Subsumption Architecture, and of reactive systems in general: If the world can provide the information directly (through sensing), it is best for the robot to get it that way rather than to store it internally in a representation, which may be large, slow, expensive, and outdated.

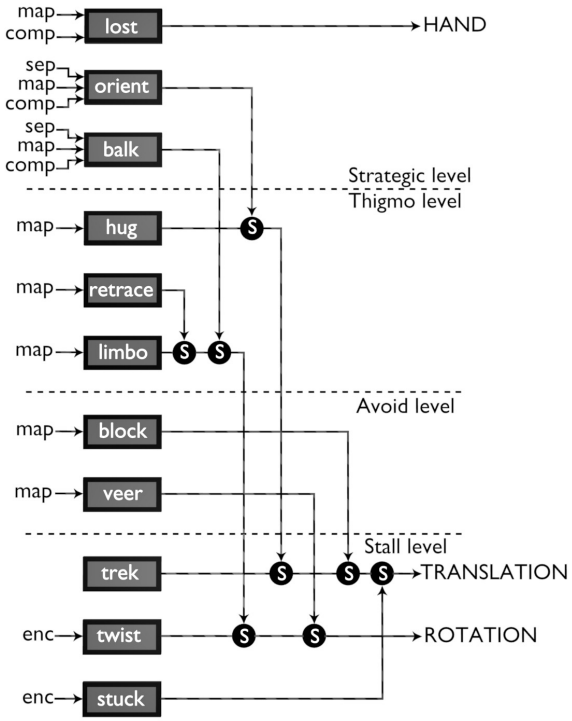

Figure 14.6 Herbert's control architecture.

To summarize, the guiding principles of Subsumption Architecture are:

- Systems are built from the bottom up.
- Components are task-achieving actions/behaviors (not functional modules).
- Components can be executed in parallel (multitasking).
- Components are organized in layers.
- Lowest layers handle the most basic tasks.

- Newly added components and layers exploit the existing ones. Each component provides and does not disrupt a tight coupling between sensing and action.
- There is no use of internal models; "the world is its own best model."

Subsumption Architecture is not the only method for structuring reactive systems, but it is a very popular one, due to its simplicity and robustness. It has been widely used in various robots that successfully interact with uncertain, dynamically changing environments.

How many rules does it take to put together a reactive system?

By now, you know the answer is "That depends on the task, the environment, and the sensors on the robot." But it is important to remember that for any robot, task, and environment that can be specified *in advance*, a complete reactive system can be defined that will achieve the robot's goals. However, that system may be prohibitively large, as it may have a huge number of rules. For example, it is in theory possible to write a reactive system for playing chess by encoding it as a giant lookup table that gives the optimal move for each possible board position. Of course the total number of all possible board positions, and all possible games from all those positions, is too huge for a human to remember or even a machine to compute and store. But for smaller problems, such as tic-tac-toe and backgammon, this approach works very well. For chess, another approach is needed: using deliberative control, as we learned in Chapter 13.

To Summarize

- Reactive control uses tight couplings between perception (sensing) and action to produce timely robotic response in dynamic and unstructured worlds (think of it as "stimulus-response").
- Subsumption Architecture is the best-known reactive architecture, but certainly not the only one.
- Reactive control uses a task-oriented decomposition of the controller. The control system consists of parallel (concurrently executed) modules that achieve specific tasks (avoid-obstacle, follow-wall, etc.).
- Reactive control is part of other types of control, specifically hybrid control and behavior-based control.

- Reactive control is a powerful method; many animals are largely reactive.
- Reactive control has limitations:
  - Minimal (if any) state
  - No memory
  - No learning
  - No internal models / representations of the world.

Food for Thought

- Can you change the goal of a reactive system? If so, how? If not, why not? You will soon learn how other methods for control deal with this problem and what the trade-offs are.
- Can you always avoid using any representation/world model? If so, how? If not, why not, and what could you do instead?
- Can a reactive robot learn a maze?

Looking for More?

- The Robotics Primer Workbook exercises for this chapter are found here: http://roboticsprimer.sourceforge.net/workbook/Reactive_Control
- Genghis, the six-legged robot we mentioned in Chapter 5 was also programmed with Subsumption Architecture. You can learn about how its control system was put together, how many rules it took, and how it gradually got better at walking over rough terrain as more subsumption layers were added, from *Cambrian Intelligence* by Rodney Brooks.

15 *Think and Act Separately, in Parallel Hybrid Control*

As we have seen, reactive control is fast but inflexible, while deliberative control is smart but slow. The basic idea behind hybrid control is to get the best of both worlds: the speed of reactive control and the brains of deliberative control. Obvious, but not easy to do.

HYBRID CONTROL

Hybrid control involves the combination of reactive and deliberative control within a single robot control system. This combination means that fundamentally different controllers, time-scales (short for reactive, long for deliberative), and representations (none for reactive, explicit and elaborate world models for deliberative) must be made to work together effectively. And that, as we will see, is a tall order.

In order to achieve the best of both worlds, a hybrid system typically consists of three components, which we can call layers or modules (though they are not the same as, and should not be confused with, layers/modules used in reactive systems):

- A reactive layer
- A planner
- A layer that links the above two together.

As a result, hybrid architectures are often called *three-layer architectures* and hybrid systems, *three-layer systems*. Figure 15.1 is a diagram of a hybrid architecture and its layers.

We already know about deliberation with planners and about action with reactive systems. What we now need to learn about is the real challenge and "value added" of hybrid systems: the “magic middle.” The middle layer has a hard job, because it has to:

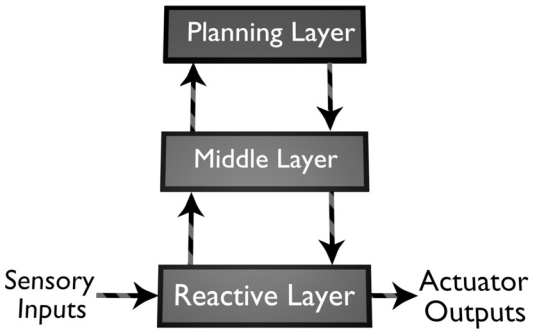

Figure 15.1 A diagram of a hybrid architecture, showing the three layers.

- Compensate for the limitations of both the planner and the reactive system
- Reconcile their different time-scales
- Deal with their different representations
- Reconcile any contradictory commands they may send to the robot.

So the main challenge of hybrid control is achieving the right compromise between the deliberative and reactive parts of the system.

Let's work through an example to see what that means in practice. Suppose we have a robot whose job is to deliver mail and paperwork through the day to various offices within an office building. This was a very popular mobile robot application that researchers liked to think about in the 1970s, 1980s, and 1990s, until email became the preferred means of communication. However, office buildings still have a great many tasks that require "gophering" or running around with deliveries. Also, we can use the same controller to deliver medicines to patients in a hospital, so let's consider that example.

In order to get around a busy office/hospital environment, the robot needs to be able to respond to unexpected obstacles, and fast-moving people and

objects (gurneys, stretchers, rolling lunch carts, etc.). For this it needs a robust reactive controller.

In order to efficiently find specific offices/rooms for making deliveries, the robot needs to use a map and plan short paths to its destinations. For this it needs an internal model and a planner of some kind.

So there you have it, a perfect candidate for a hybrid system. We already know how to do both of the component systems (reactive collision-free navigation and deliberative path planning), so how hard can it be to put the two together?

Here is how hard:

- What happens if the robot needs to deliver a medication to a patient as soon as possible, yet it does not have a plan for a short path to the patient's room? Should it wait for the plan to be computed, or should it move down the corridor (in which direction?) while it is still planning in its head?
- What happens if the robot is headed down the shortest path but suddenly a crew of doctors with a patient on a stretcher starts heading its way? Should it just stop and get out of the way in any direction, and wait as long as it may take, or should it start replanning an alternative path?
- What happens if the plan the robot computed from the map is blocked, because the map is out of date?
- What happens if the patient was moved to another room without the robot knowing?
- What happens if the robot keeps having to go to the same room, and so has to replan that path or parts of it all the time?
- What if, what if, what if.

The methods for handling the above situations and various others are usually implemented in the "magic middle" layer of a hybrid system. Designing the middle layer and its interactions with the other two layers (as shown in figure 15.2) are the main challenges of hybrid control. Here are some commonly used methods and approaches.

15.1 Dealing with Changes in the World/Map/Task

When the reactive system discovers that it cannot do its job (for example, it can't proceed because of an obstacle, a closed door, or some other snag), it can

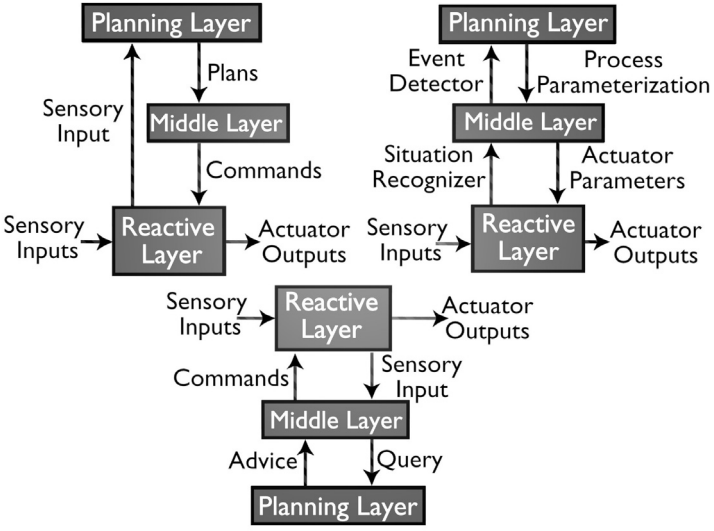

Figure 15.2 Some ways of managing layer interaction in hybrid controllers for robots.

inform the deliberative layer about this new development. The deliberative layer can use this information to update its representation of the world, so that it can now, and in the future, generate more accurate and useful plans.

This is a good idea not only because it is necessary to update the internal model when things change, but also because we already know that updating internal models and generating plans takes time and computation, so it cannot be afforded continually. The input from the reactive layer provides an indication of a very good time for such an update.

15.2 Planning and Replanning

Whenever the reactive layer discovers that it cannot proceed, this can be used as a signal to the deliberative layer to do some thinking, in order to generate a new plan. This is called *dynamic replanning*.

But not all information flows from the bottom up, from the reactive to the deliberative layer. In fact, the deliberative layer, which provides the path

to the goal (i.e., the room to go to), gives the robot the directions to follow, turns to take, distances to go. If the planner is computing while the robot is moving, it may send a message to the reactive navigation layer to stop, turn around, and head in a different direction because a better way has been discovered in the map.

In general, a complete plan, when finished, has the best answer the deliberator can generate. But sometimes there is not enough time to wait for that full and optimal answer (such as when the patient really needs immediate help). In those cases, it is often best to get the robot going in the generally right direction (for example, toward the correct wing of the hospital) and in the meantime keep generating a more accurate and detailed plan, and updating the navigation layer as needed.

Of course the timing of these two processes, the reactive navigation (running around) and deliberative planning (thinking hard), is obviously not synched up, so the robot may get to the right wing of the hospital and not know where to go next, and thus may have to stop and wait for the planner to get done with its next step. Alternatively, the planner may need to wait for the robot to move around or get out of a crowded area in order to figure out exactly where it is, so that it can generate a useful plan. We will learn more about the various challenges of navigation in Chapter 19.

15.3 Avoiding Replanning

A useful idea researchers in planning had quite a while ago was to remember/save/store plans so that they would not have to be generated again in the future. Of course, every plan is specific to the particular initial state and goal state, but if those are at all likely to recur, it is worth stashing away the plan for future use.

This idea is really popular for situations that happen often and need a fast decision. In our example above, such situations include dealing with crowded junctions of corridors or dynamic obstacles that interfere with a particular direction of movement. Instead of "thinking out" how to deal with this every time, controller designers often preprogram many of these responses ahead of time. But because these are not quite reactive rules, because they involve multiple steps, and yet they are not quite plans, because they do not involve very many steps, they find a perfect home in the middle layer of the three-layer system.

This basic idea of storing and reusing mini plans for repeated situations

has been used in "contingency tables," literally lookup tables that tell the robot what to do (as in a reactive system) and pull out a little plan as a response (as in a deliberative system). This same idea has appeared in the form of fancy terms such as "intermediate-level actions" and "macro operators." In all cases, it is basically the same thing: plans that are computed (either off-line or during the robot's life) and stored for fast lookup in the future.

So who has ultimate priority, the deliberative layer or the reactive layer?

The answer is, as usual, "It depends." What it depends on includes the type of environment, task, sensing, timing, and reaction requirements, and so on. Some hybrid systems use a hierarchical structure, so that one of the systems is always in charge. In some cases, the deliberator's plan is "the law" for the system, while in others the reactive system merely considers it as advice that can be ignored. In the most effective systems, the interaction between thinking and acting is coupled, so that each can inform and interrupt the other. But in order to know who should be in charge when, it is necessary to consider the different modes of the system and specify who gets its say. For example, the planner can interrupt the reactive layer if it (the planner) has a better path than the current being executed, but it should do so only if the new plan is sufficiently better to make it worth the bother. On the other hand, the reactive layer can interrupt the planner if it (the reactive layer) finds a blocked path and cannot proceed, but should only do so after it has tried to get around the barrier for a while.

15.4 On-Line and Off-Line Planning

In everything we've talked about so far, we have assumed that the deliberation and the reaction are happening as the robot is moving around. But as we saw from the role of the middle layer, it is useful to store plans once they are generated. So the next good idea is to preplan for all the situations that might come up, and store those plans ahead of time. This *off-line planning* takes place while the robot is being developed and does not have much to worry about, as compared with *on-line planning* of the kind that a busy robot has to worry about while it is trying to get its job done and its goals achieved.

OFF-LINE PLANNING

ON-LINE PLANNING

> *If we can preplan, why not generate all possible plans ahead of time, store them all, and just look them up, without ever having to do search and deliberation at run-time, while the robot is trying to be fast and efficient?*

This is the idea behind universal plans.

UNIVERSAL PLAN

A *universal plan* is a set of all possible plans for all initial states and all goals within the state space of a particular system.

If for each situation a robot has a preexisting optimal plan it only has to look up, then it can always react optimally, and so have both reactive and deliberative capabilities without deliberating at all. Such a robot is reactive, since the planning is all done off-line and not at run-time.

Another good feature of such precompiled plans is that information can be put into the system in a clean, principled way. Such information about the robot, the task, and the environment is called *domain knowledge*. It is compiled into a reactive controller, so the information does not have to be reasoned about (or planned with) on line, in real time but instead becomes a set of real-time reactive rules that can be looked up.

DOMAIN KNOWLEDGE

This idea was so popular that researchers even developed a way of generating such precompiled plans automatically, by using a special programming language and a compiler. Robot programs written in that language produced a kind of a universal plan. To do so, the program took as input a mathematical specification of the world and of the robot's goals, and produced a control "circuit" (a diagram of what is connected to what) for a reactive "machine." These machines were called *situated automata*; they were not real, physical machines but formal (idealized) ones whose inputs were connected to abstract (again idealized, not real) sensors, and whose outputs were connected to abstract effectors. To be *situated* means to exist in a complex world and to interact with it; *automata* are computing machines with particular mathematical properties.

SITUATED AUTOMATA

Unfortunately, this is too good to be true for real-world robots. Here is why:

- The state space is too large for most realistic problems, so either generating or storing a universal plan is simply not possible.
- The world must not change; if it does, new plans need to be generated for the changed environment.
- The goals must not change; this is the same as with reactive systems, where if the goals change, at least some of the rules need to change as well.

Situated automata could not be situated in the real, physical world after all. Robots exist in the real world but we can't easily write mathematical

specifications that fully describe this world, which is what that the situated automata approach required.

And so we are back to having to do deliberation and reaction in real time, and hybrid systems are a good way to do that. They do have their own drawbacks, of course, including:

- The middle layer is hard to design and implement, and it tends to be very special-purpose, crafted for the specific robot and task, so it has to be reinvented for almost every new robot and task.
- The best of both worlds can end up being the worst of both worlds; if mismanaged, a hybrid system can degenerate into having the planner slow down the reactive system, and the reactive system ignore the planner entirely, minimizing the effectiveness of both.
- An effective hybrid system is not easy to design or debug, but that is true for any robot system.

In spite of the drawbacks of hybrid systems, they are the most popular choice for a great many problems in robotics, especially those involving a single robot that has to perform one or more tasks that involve thinking of some type (such as delivery, mapping, and many others) as well as reacting to a dynamic environment.

To Summarize

- Hybrid control aims to bring together the best aspects of reactive control and deliberative control, allowing the robot to both plan and react.
- Hybrid control involves real-time reactive control in one part of the system (usually the low level) and more time-expensive deliberative control in another part of the system (usually the high level), with an intermediate level in between.
- Hybrid architectures are also called three-layer architectures because of the three distinct layers or components of the control system.
- The main challenge of hybrid systems lies in bringing together the reactive and deliberative components in a way that results in consistent, timely, and robust behavior over all.
- Hybrid systems, unlike reactive systems, are capable of storing representation, planning, and learning.

Food for Thought

Is there an alternative to hybrid systems, or is that all there is in terms of intelligent, real-time robot behavior? Can you think of another way that a robot can be able to both think and react? After you come up with an answer, check out the next chapter.

Looking for More?

- The Robotics Primer Workbook exercises for this chapter are found here: http://roboticsprimer.sourceforge.net/workbook/Hybrid_Control
- Hybrid control is a major area of research in control theory, a field of study usually pursued in electrical engineering, which we described in Chapter 2. (Recall that control theory covers feedback control, which we studied in Chapter 10.) In control theory, the problem of hybrid control deals with the general interplay between continuous and discrete signals and representations, and the control of systems that contain them. Here are a couple of popular textbooks on control theory:
  - *Signals and Systems* by Alan V. Oppenheim, Alan S. Willsky, and with Nawab S. Hamid.
  - *Modern Control Systems*, by Richard C. Dorf and Robert H. Bishop.