Chapter 2

Agent Architectures and Hierarchical Control

> *By a hierarchic system, or hierarchy, I mean a system that is composed of interrelated subsystems, each of the latter being in turn hierarchic in structure until we reach some lowest level of elementary subsystem. In most systems of nature it is somewhat arbitrary as to where we leave off the partitioning and what subsystems we take as elementary. Physics makes much use of the concept of "elementary particle," although the particles have a disconcerting tendency not to remain elementary very long ...*
>
> *Empirically a large proportion of the complex systems we observe in nature exhibit hierarchic structure. On theoretical grounds we would expect complex systems to be hierarchies in a world in which complexity had to evolve from simplicity.*
>
> – Herbert A. Simon [1996]

This chapter discusses how an intelligent agent can perceive, reason, and act over time in an environment. In particular, it considers the internal structure of an agent. As Simon points out in the quote above, hierarchical decomposition is an important part of the design of complex systems such as intelligent agents. This chapter presents ways to design agents in terms of hierarchical decompositions and ways that agents can be built, taking into account the knowledge that an agent needs to act intelligently.

2.1 Agents

An **agent** is something that acts in an environment. An agent can, for example, be a person, a robot, a dog, a worm, the wind, gravity, a lamp, or a computer program that buys and sells.

Purposive agents have preferences. They prefer some states of the world to other states, and they act to try to achieve the states they prefer most. The non-purposive agents are grouped together and called **nature**. Whether or not an agent is purposive is a modeling assumption that may, or may not, be appropriate. For example, for some applications it may be appropriate to model a dog as purposive, and for others it may suffice to model a dog as non-purposive.

If an agent does not have preferences, by definition it does not care what world state it ends up in, and so it does not matter what it does. The only reason to design an agent is to instill it with preferences – to make it prefer some world states and try to achieve them. An agent does not have to know its preferences. For example, a thermostat is an agent that senses the world and turns a heater either on or off. There are preferences embedded in the thermostat, such as to keep the occupants of a room at a pleasant temperature, even though the thermostat arguably does not know these are its preferences. The preferences of an agent are often the preferences of the designer of the agent, but sometimes an agent can be given goals and preferences at run time.

Agents interact with the environment with a **body**. An **embodied** agent has a physical body. A **robot** is an artificial purposive embodied agent. Sometimes agents that act only in an information space are called robots, but we just refer to those as agents.

This chapter considers how to build purposive agents. We use robots as a main motivating example, because much of the work has been carried out in the context of robotics and much of the terminology is from robotics. However, the discussion is intended to cover all agents.

Agents receive information through their **sensors**. An agent's actions depend on the information it receives from its sensors. These sensors may, or may not, reflect what is true in the world. Sensors can be noisy, unreliable, or broken, and even when sensors are reliable there is still ambiguity about the world based on sensor readings. An agent must act on the information it has available. Often this information is very weak, for example, "sensor s appears to be producing value v."

Agents act in the world through their **actuators** (also called **effectors**). Actuators can also be noisy, unreliable, slow, or broken. What an agent controls is the message (command) it sends to its actuators. Agents often carry out actions to find more information about the world, such as opening a cupboard door to find out what is in the cupboard or giving students a test to determine their knowledge.

2.2 Agent Systems

Figure 2.1 depicts the general interaction between an agent and its environment. Together the whole system is known as an agent system.

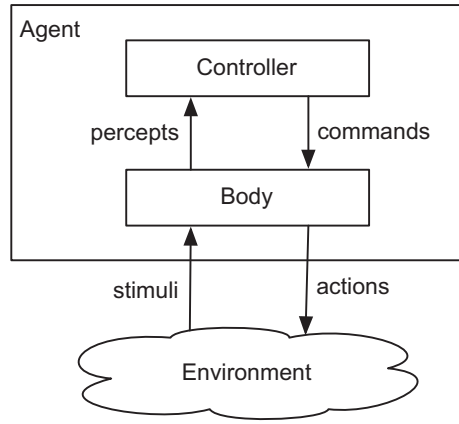

Figure 2.1: An agent system and its components

An **agent system** is made up of an agent and its environment. The agent receives **stimuli** from the environment and carries out **actions** in the environment.

An **agent** is made up of a **body** and a **controller**. The controller receives **percepts** from the body and sends **commands** to the body.

A body includes **sensors** that convert stimuli into percepts and **actuators** that convert commands into actions.

Stimuli include light, sound, words typed on a keyboard, mouse movements, and physical bumps. The stimuli can also include information obtained from a web page or from a database.

Common sensors include touch sensors, cameras, infrared sensors, sonar, microphones, keyboards, mice, and XML readers used to extract information from web pages. As a prototypical sensor, a camera senses light coming into its lens and converts it into a two-dimensional array of intensity values called **pixels**. Sometimes multiple pixel arrays exist for different colors or for multiple cameras. Such pixel arrays could be the percepts for our controller. More often, percepts consist of higher-level features such as lines, edges, and depth information. Often the percepts are more specialized – for example, the positions of bright orange dots, the part of the display a student is looking at, or the hand signals given by a human.

Actions include steering, accelerating wheels, moving links of arms, speaking, displaying information, or sending a post command to a web site. Commands include low-level commands such as to set the voltage of a motor to some particular value, and high-level specifications of the desired motion of a robot, such as "stop" or "travel at 1 meter per second due east" or "go to room 103." Actuators, like sensors, are typically noisy. For example, stopping takes time; a robot is governed by the laws of physics and has momentum, and messages take time to travel. The robot may end up going only approximately

1 meter per second, approximately east, and both speed and direction may fluctuate. Even traveling to a particular room may fail for a number of reasons.

The controller is the brain of the agent. The rest of this chapter is about how to build controllers.

2.2.1 The Agent Function

Agents are situated in time: they receive sensory data in time and do actions in time. The action that an agent does at a particular time is a function of its inputs (page 10). We first consider the notion of time.

Let T be the set of **time** points. Assume that T is totally ordered and has some metric that can be used to measure the temporal distance between any two time points. Basically, we assume that T can be mapped to some subset of the real line.

T is **discrete** if there exist only a finite number of time points between any two time points; for example, there is a time point every hundredth of a second, or every day, or there may be time points whenever interesting events occur. T is **dense** if there is always another time point between any two time points; this implies there must be infinitely many time points between any two points. Discrete time has the property that, for all times, except perhaps a last time, there is always a next time. Dense time does not have a "next time." Initially, we assume that time is discrete and goes on forever. Thus, for each time there is a next time. We write $t + 1$ to be the next time after time t; it does not mean that the time points are equally spaced.

Assume that T has a starting point, which we arbitrarily call 0.

Suppose P is the set of all possible percepts. A **percept trace**, or **percept stream**, is a function from T into P. It specifies what is observed at each time.

Suppose C is the set of all commands. A **command trace** is a function from T into C. It specifies the command for each time point.

Example 2.1 Consider a household trading agent that monitors the price of some commodity (e.g., it checks online for special deals and for price increases for toilet paper) and how much the household has in stock. It must decide whether to buy more and how much to buy. The percepts are the price and the amount in stock. The command is the number of units the agent decides to buy (which is zero if the agent does not buy any). A percept trace specifies for each time point (e.g., each day) the price at that time and the amount in stock at that time. Percept traces are given in Figure 2.2. A command trace specifies how much the agent decides to buy at each time point. An example command trace is given in Figure 2.3.

The action of actually buying depends on the command but may be different. For example, the agent could issue a command to buy 12 rolls of toilet paper at a particular price. This does not mean that the agent actually buys 12 rolls because there could be communication problems, the store could have run out of toilet paper, or the price could change between deciding to buy and actually buying.

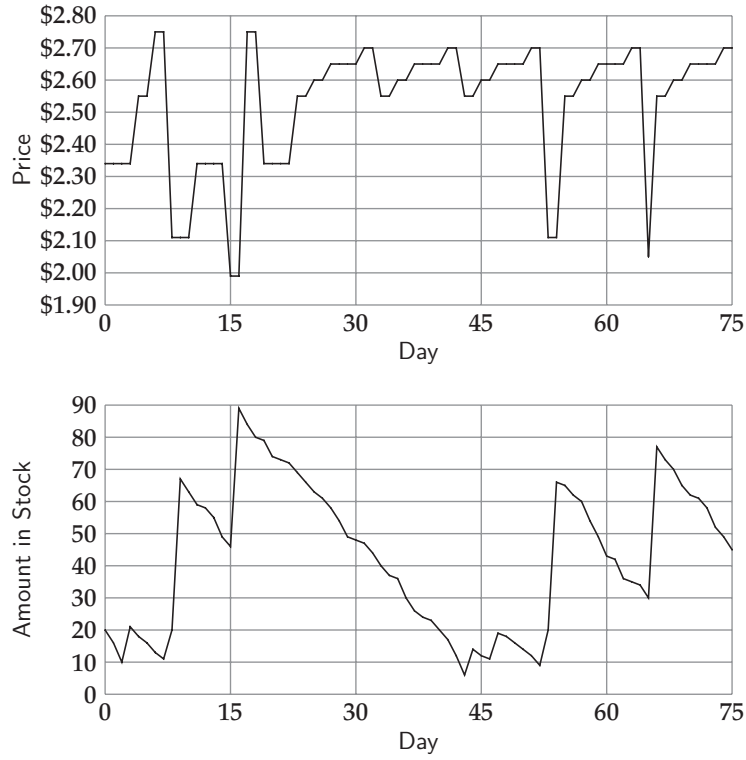

Figure 2.2: Percept traces for Example 2.1

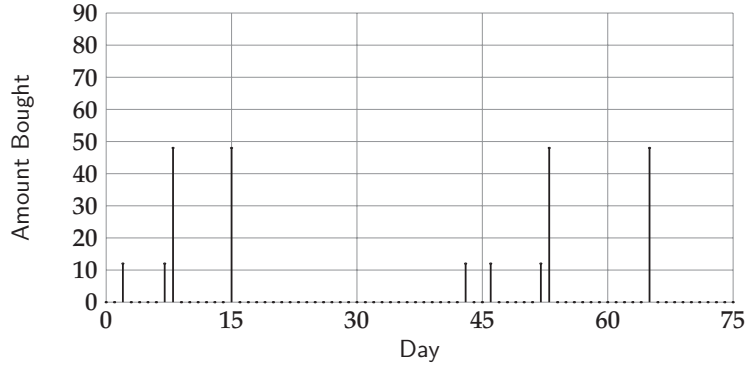

Figure 2.3: Command trace for Example 2.1

A percept trace for an agent is thus the sequence of all past, present, and future percepts received by the controller. A command trace is the sequence of all past, present, and future commands issued by the controller. The commands can be a function of the history of percepts. This gives rise to the concept of a **transduction**, a function that maps percept traces into command traces.

Because all agents are situated in time, an agent cannot actually observe full percept traces; at any time it has only experienced the part of the trace up to *now*. It can only observe the value of the trace at time $t \in T$ when it gets to time t. Its command can only depend on what it has experienced.

A transduction is **causal** if, for all times t, the command at time t depends only on percepts up to and including time t. The causality restriction is needed because agents are situated in time; their command at time t cannot depend on percepts after time t.

A **controller** is an implementation of a causal transduction.

The **history** of an agent at time t is the percept trace of the agent for all times before or at time t and the command trace of the agent before time t.

Thus, a **causal transduction** specifies a function from the agent's history at time t into the command at time t. It can be seen as the most general specification of an agent.

Example 2.2 Continuing Example 2.1 (page 46), a causal transduction specifies, for each time, how much of the commodity the agent should buy depending on the price history, the history of how much of the commodity is in stock (including the current price and amount in stock) and the past history of buying.

An example of a causal transduction is as follows: buy four dozen rolls if there are fewer than five dozen in stock and the price is less than 90% of the average price over the last 20 days; buy a dozen more rolls if there are fewer than a dozen in stock; otherwise, do not buy any.

Although a causal transduction is a function of an agent's history, it cannot be directly implemented because an agent does not have direct access to its entire history. It has access only to its current percepts and what it has remembered.

The **belief state** of an agent at time t is all of the information the agent has remembered from the previous times. An agent has access only to its history that it has encoded in its belief state. Thus, the belief state encapsulates all of the information about its history that the agent can use for current and future commands. At any time, an agent has access to its belief state and its percepts.

The belief state can contain any information, subject to the agent's memory and processing limitations. This is a very general notion of belief; sometimes we use a more specific notion of belief, such as the agent's belief about what is true in the world, the agent's beliefs about the dynamics of the environment, or the agent's belief about what it will do in the future.

Some instances of belief state include the following:

- The belief state for an agent that is following a fixed sequence of instructions may be a program counter that records its current position in the sequence.

- The belief state can contain specific facts that are useful – for example, where the delivery robot left the parcel in order to go and get the key, or where it has already checked for the key. It may be useful for the agent to remember anything that is reasonably stable and that cannot be immediately observed.
- The belief state could encode a model or a partial model of the state of the world. An agent could maintain its best guess about the current state of the world or could have a probability distribution over possible world states; see Section 5.6 (page 199) and Chapter 6.
- The belief state could be a representation of the dynamics of the world and the meaning of its percepts, and the agent could use its perception to determine what is true in the world.
- The belief state could encode what the agent **desires**, the **goals** it still has to achieve, its **beliefs** about the state of the world, and its **intentions**, or the steps it intends to take to achieve its goals. These can be maintained as the agent acts and observes the world, for example, removing achieved goals and replacing intentions when more appropriate steps are found.

A controller must maintain the agent's belief state and determine what command to issue at each time. The information it has available when it must do this includes its belief state and its current percepts.

A **belief state transition function** for discrete time is a function

$$remember : S \times P \to S$$

where S is the set of belief states and P is the set of possible percepts; $s_{t+1} = remember(s_t, p_t)$ means that s_{t+1} is the belief state following belief state s_t when p_t is observed.

A **command function** is a function

$$do : S \times P \to C$$

where S is the set of belief states, P is the set of possible percepts, and C is the set of possible commands; $c_t = do(s_t, p_t)$ means that the controller issues command c_t when the belief state is s_t and when p_t is observed.

The belief-state transition function and the command function together specify a causal transduction for the agent. Note that a causal transduction is a function of the agent's history, which the agent doesn't necessarily have access to, but a command function is a function of the agent's belief state and percepts, which it does have access to.

Example 2.3 To implement the causal transduction of Example 2.2, a controller must keep track of the rolling history of the prices for the previous 20 days. By keeping track of the average (*ave*), it can update the average using

$$ave := ave + \frac{new - old}{20}$$

where *new* is the new price and *old* is the oldest price remembered. It can then discard *old*. It must do something special for the first 20 days.

A simpler controller could, instead of remembering a rolling history in order to maintain the average, remember just the average and use the average as a surrogate for the oldest item. The belief state can then contain one real number (ave). The state transition function to update the average could be

$$ave := ave + \frac{new - ave}{20}$$

This controller is much easier to implement and is not sensitive to what happened 20 time units ago. This way of maintaining estimates of averages is the basis for temporal differences in reinforcement learning (page 467).

If there exists a finite number of possible belief states, the controller is called a **finite state controller** or a **finite state machine**. A **factored representation** is one in which the belief states, percepts, or commands are defined by features (page 21). If there exists a finite number of features, and each feature can only have a finite number of possible values, the controller is a **factored finite state machine**. Richer controllers can be built using an unbounded number of values or an unbounded number of features. A controller that has countably many states can compute anything that is computable by a Turing machine.

2.3 Hierarchical Control

One way that you could imagine building an agent depicted in Figure 2.1 (page 45) is to split the body into the sensors and a complex perception system that feeds a description of the world into a reasoning engine implementing a controller that, in turn, outputs commands to actuators. This turns out to be a bad architecture for intelligent systems. It is too slow, and it is difficult to reconcile the slow reasoning about complex, high-level goals with the fast reaction that an agent needs, for example, to avoid obstacles. It also is not clear that there is a description of a world that is independent of what you do with it (see Exercise 1 (page 66)).

An alternative architecture is a hierarchy of controllers as depicted in Figure 2.4. Each layer sees the layers below it as a **virtual body** from which it gets percepts and to which it sends commands. The lower-level layers are able to run much faster, react to those aspects of the world that need to be reacted to quickly, and deliver a simpler view of the world to the higher layers, hiding inessential information.

In general, there can be multiple features passed from layer to layer and between states at different times.

There are three types of inputs to each layer at each time:

- the features that come from the belief state, which are referred to as the remembered or previous values of these features;
- the features representing the percepts from the layer below in the hierarchy; and
- the features representing the commands from the layer above in the hierarchy.

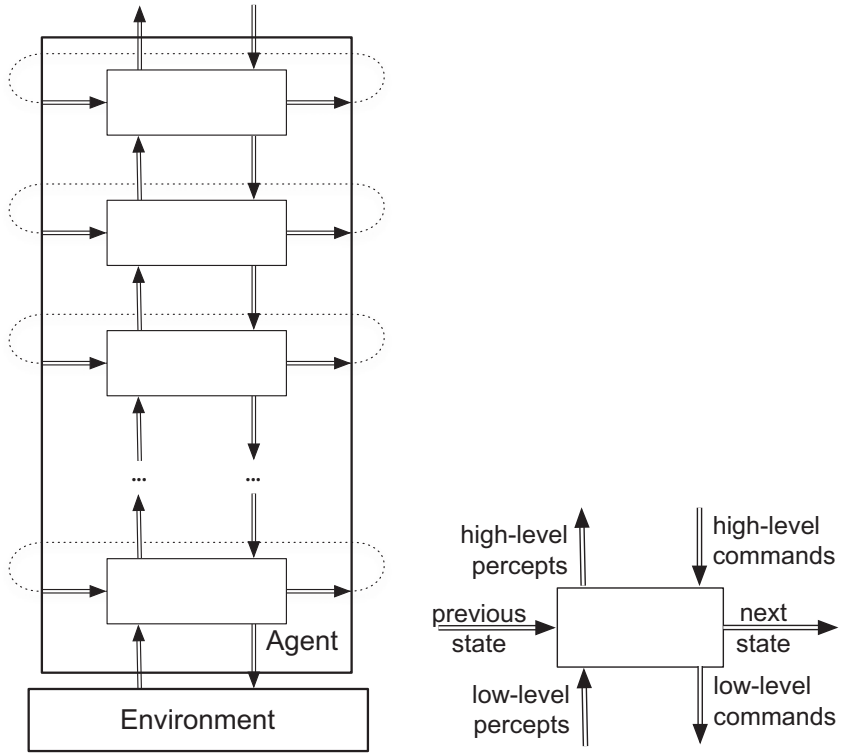

Figure 2.4: An idealized hierarchical agent system architecture. The unlabeled rectangles represent layers, and the double lines represent information flow. The dotted lines show how the output at one time is the input for the next time.

There are three types of outputs from each layer at each time:

- the higher-level percepts for the layer above,
- the lower-level commands for the layer below, and
- the next values for the belief-state features.

An implementation of a layer specifies how the outputs of a layer are a function of its inputs. Computing this function can involve arbitrary computation, but the goal is to keep each layer as simple as possible.

To implement a controller, each input to a layer must get its value from somewhere. Each percept or command input should be connected to an output of some other layer. Other inputs come from the remembered beliefs. The outputs of a layer do not have to be connected to anything, or they could be connected to multiple inputs.

High-level reasoning, as carried out in the higher layers, is often discrete and qualitative, whereas low-level reasoning, as carried out in the lower layers, is often continuous and quantitative (see box on page 52). A controller that reasons in terms of both discrete and continuous values is called a **hybrid system**.

Qualitative Versus Quantitative Representations

Much of science and engineering considers **quantitative reasoning** with numerical quantities, using differential and integral calculus as the main tools. **Qualitative reasoning** is reasoning, often using logic, about qualitative distinctions rather than numerical values for given parameters.

Qualitative reasoning is important for a number of reasons:

- An agent may not know what the exact values are. For example, for the delivery robot to pour coffee, it may not be able to compute the optimal angle that the coffee pot needs to be tilted, but a simple control rule may suffice to fill the cup to a suitable level.
- The reasoning may be applicable regardless of the quantitative values. For example, you may want a strategy for a robot that works regardless of what loads are placed on the robot, how slippery the floors are, or what the actual charge is of the batteries, as long as they are within some normal operating ranges.
- An agent needs to do qualitative reasoning to determine which quantitative laws are applicable. For example, if the delivery robot is filling a coffee cup, different quantitative formulas are appropriate to determine where the coffee goes when the coffee pot is not tilted enough for coffee to come out, when coffee comes out into a non-full cup, and when the coffee cup is full and the coffee is soaking into the carpet.

Qualitative reasoning uses discrete values, which can take a number of forms:

- **Landmarks** are values that make qualitative distinctions in the individual being modeled. In the coffee example, some important qualitative distinctions include whether the coffee cup is empty, partially full, or full. These landmark values are all that is needed to predict what happens if the cup is tipped upside down or if coffee is poured into the cup.
- **Orders-of-magnitude reasoning** involves approximate reasoning that ignores minor distinctions. For example, a partially full coffee cup may be full enough to deliver, half empty, or nearly empty. These **fuzzy terms** have ill-defined borders. Some relationship exists between the actual amount of coffee in the cup and the qualitative description, but there may not be strict numerical divisors.
- **Qualitative derivatives** indicate whether some value is increasing, decreasing, or staying the same.

A flexible agent needs to do qualitative reasoning before it does quantitative reasoning. Sometimes qualitative reasoning is all that is needed. Thus, an agent does not always need to do quantitative reasoning, but sometimes it needs to do both qualitative and quantitative reasoning.

Example 2.4 Consider a delivery robot (page 32) able to carry out high-level navigation tasks while avoiding obstacles. Suppose the delivery robot is required to visit a sequence of named locations in the environment of Figure 1.7 (page 32), avoiding obstacles it may encounter.

Assume the delivery robot has wheels like a car, and at each time can either go straight, turn right, or turn left. It cannot stop. The velocity is constant and the only command is to set the steering angle. Turning the wheels is instantaneous, but adjusting to a certain direction takes time. Thus, the robot can only travel straight ahead or go around in circular arcs with a fixed radius.

The robot has a position sensor that gives its current coordinates and orientation. It has a single whisker sensor that sticks out in front and slightly to the right and detects when it has hit an obstacle. In the example below, the whisker points 30° to the right of the direction the robot is facing. The robot does not have a map, and the environment can change (e.g., obstacles can move).

A layered controller for such a delivery robot is depicted in Figure 2.5 (on the next page). The robot is given a high-level plan to execute. The plan is a sequence of named locations to visit in order. The robot needs to sense the world and to move in the world in order to carry out the plan. The details of the lower layer are not shown in this figure.

The top layer, called *follow plan*, is described in Example 2.6 (page 56). That layer takes in a plan to execute. The plan is a list of named locations to visit in order. The locations are selected in order. Each selected location becomes the current target. This layer determines the x-y coordinates of the target. These coordinates are the target position for the lower level. The upper level knows about the names of locations, but the lower levels only know about coordinates.

The top layer maintains a belief state consisting of a list of names of locations that the robot still needs to visit and the coordinates of the current target. It issues commands to the middle layer in terms of the coordinates of the current target.

The middle layer, which could be called *go to target and avoid obstacles*, tries to keep traveling toward the current target position, avoiding obstacles. The middle layer is described in Example 2.5 (page 55). The target position, *target_pos*, is obtained from the top layer. When the middle layer has arrived at the target position, it signals to the top layer that it has achieved the target by setting *arrived* to be true. This signal can be implemented either as the middle layer issuing an interrupt to the top layer, which was waiting, or as the top layer continually monitoring the middle layer to determine when *arrived* becomes true. When *arrived* becomes true, the top layer then changes the target position to the coordinates of the next location on the plan. Because the top layer changes the current target position, the middle layer must use the *previous* target position to determine whether it has arrived. Thus, the middle layer must get both the current and the previous target positions from the top layer: the previous target position to determine whether it has arrived, and the current target position to travel to.

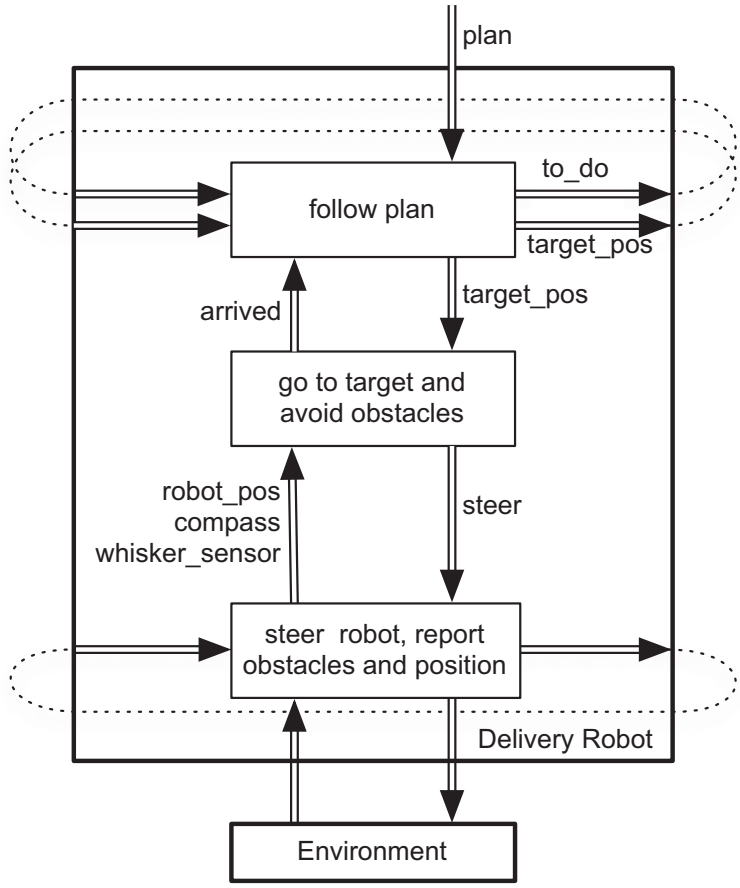

Figure 2.5: A hierarchical decomposition of the delivery robot

The middle layer can access the robot's current position and direction and can determine whether its single whisker sensor is on or off. It can use a simple strategy of trying to head toward the target unless it is blocked, in which case it turns left.

The middle layer is built on a lower layer that provides a simple view of the robot. This lower layer could be called *steer robot and report obstacles and position*. It takes in steering commands and reports the robot's position, orientation, and whether the sensor is on or off.

Inside a layer are features that can be functions of other features and of the inputs to the layers. There is an arc into a feature from the features or inputs on which it is dependent. The graph of how features depend on each other must be acyclic. The acyclicity of the graph allows the controller to be implemented by running a program that assigns the values in order. The features that make up the belief state can be written to and read from memory.

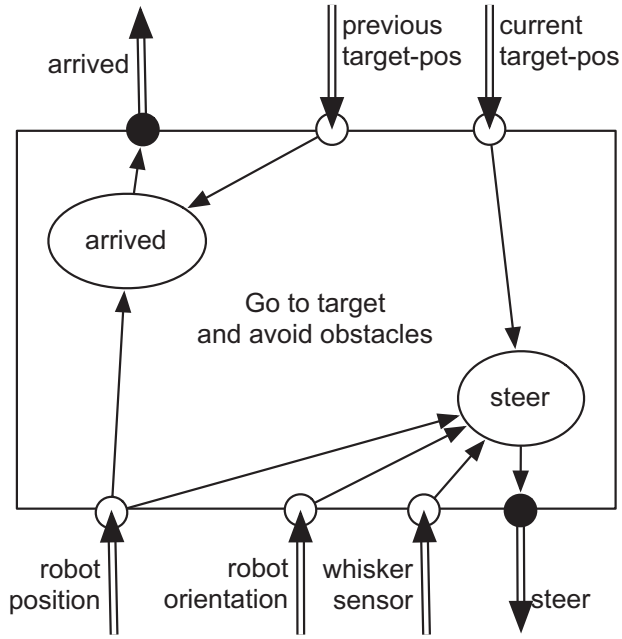

Figure 2.6: The middle layer of the delivery robot

Example 2.5 The middle *go to location and avoid obstacles* layer steers the robot to avoid obstacles. The inputs and outputs of this layer are given in Figure 2.6.

The robot has a single whisker sensor that detects obstacles touching the whisker. The one bit value that specifies whether the whisker sensor has hit an obstacle is provided by the lower layer. The lower layer also provides the robot position and orientation. All the robot can do is steer left by a fixed angle, steer right, or go straight. The aim of this layer is to make the robot head toward its current target position, avoiding obstacles in the process, and to report when it has arrived.

This layer of the controller maintains no internal belief state, so the belief state transition function is vacuous. The command function specifies the robot's steering direction as a function of its inputs and whether the robot has arrived.

The robot has arrived if its current position is close to the previous target position. Thus, *arrived* is assigned a value that is a function of the robot position and previous target position, and a threshold constant:

$$arrived := distance(previous\_target\_pos, robot\_pos) < threshold$$

where $:=$ means assignment, *distance* is the Euclidean distance, and *threshold* is a distance in the appropriate units.

The robot steers left if the whisker sensor is on; otherwise it heads toward the target position. This can be achieved by assigning the appropriate value to

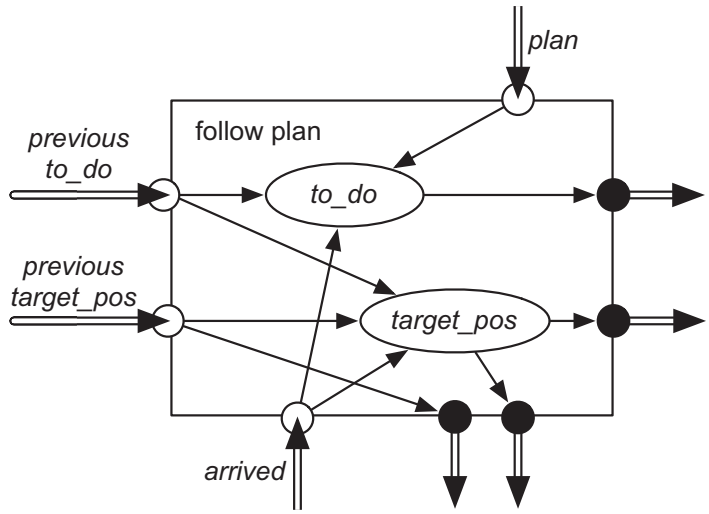

Figure 2.7: The top layer of the delivery robot controller

the *steer* variable:

if $whisker\_sensor = on$
  then $steer := left$
else if $straight\_ahead(robot\_pos, robot\_dir, current\_target\_pos)$
  then $steer := straight$
else if $left\_of(robot\_position, robot\_dir, current\_target\_pos)$
  then $steer := left$
else $steer := right$
end if

where $straight\_ahead(robot\_pos, robot\_dir, current\_target\_pos)$ is true when the robot is at $robot\_pos$, facing the direction $robot\_dir$, and when the current target position, $current\_target\_pos$, is straight ahead of the robot with some threshold (for later examples, this threshold is 11° of straight ahead). The function *left_of* tests if the target is to the left of the robot.

This layer is purely quantitative. It reasons in terms of numerical quantities rather than discrete values.

Example 2.6 The top layer, *follow plan*, is given a plan – a list of named locations to visit in order. These are the kinds of targets that could be produced by a planner, such as those developed in Chapter 8. The top layer is also told when the robot has arrived at the previous target. It must output target coordinates to the middle layer, and remember what it needs to carry out the plan. The layer is shown in Figure 2.7.

This layer maintains an internal belief state. It remembers the current target position and what locations it still has to visit. The *to_do* feature has as its value a

list of all pending locations to visit. The *target_pos* feature maintains the position for the current target.

Once the robot has arrived at its previous target, the next target position is the coordinate of the next location to visit. The top-level plan given to the robot is in terms of named locations, so these must be translated into coordinates for the middle layer to use. The following code shows how the target position and the *to_do* list are changed when the robot has arrived at its previous target position:

if *arrived* and not $empty(to\_do)$
    then
        $target\_pos' := coordinates(head(to\_do))$
        $to\_do' := tail(to\_do)$
end if

where to_do' is the next value for the *to_do* feature, and $target\_pos'$ is the next target position. Here $head(to\_do)$ is the first element of the *to_do* list, $tail(to\_do)$ is the rest of the *to_do* list, and $empty(to\_do)$ is true when the *to_do* list is empty.

In this layer, if the *to_do* list becomes empty, the robot does not change its target position. It keeps going around in circles. See Exercise 2.3 (page 67).

This layer determines the coordinates of the named locations. This could be done by simply having a database that specifies the coordinates of the locations. Using such a database is sensible if the locations do not move and are known a priori. However, if the locations can move, the lower layer must be able to tell the upper layer the current position of a location. The top layer would have to ask the lower layer the coordinates of a given location. See Exercise 2.8 (page 68).

To complete the controller, the belief state variables must be initialized, and the top-level plan must be input. This can be done by initializing the *to_do* list with the tail of the plan and the *target_pos* with the location of the first location.

A simulation of the plan $[goto(o109), goto(storage), goto(o109), goto(o103)]$ with one obstacle is given in Figure 2.8 (on the next page). The robot starts at position $(0, 5)$ facing $90°$ (north), and there is a rectangular obstacle between the positions $(20, 20)$ and $(35, -5)$.

2.3.1 Agents Modeling the World

The definition of a belief state is very general and does not constrain what should be remembered by the agent. Often it is useful for the agent to maintain some model of the world, even if its model is incomplete and inaccurate. A **model** of a world is a representation of the state of the world at a particular time and/or the dynamics of the world.

One method is for the agent to maintain its belief about the world and to update these beliefs based on its commands. This approach requires a model of both the state of the world and the dynamics of the world. Given the state at one time, and the dynamics, the state at the next time can be predicted. This

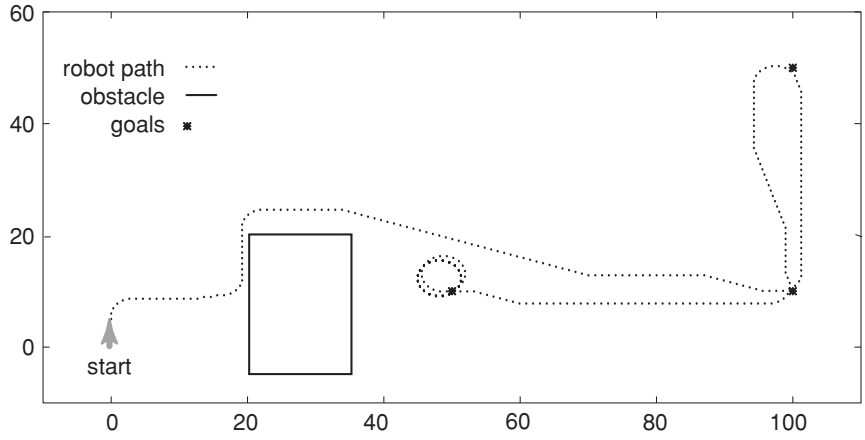

Figure 2.8: A simulation of the robot carrying out the plan of Example 2.6

process is known as **dead reckoning**. For example, a robot could maintain its estimate of its position and update it based on its actions. When the world is dynamic or when there are noisy actuators (e.g., a wheel slips, it is not of exactly the right diameter, or acceleration is not instantaneous), the noise accumulates, so that the estimates of position soon become so inaccurate that they are useless. However, if the model is accurate at some level of abstraction, this may be an appropriate model of that level of abstraction.

An alternative is to use **perception** to build a model of the relevant part of the world. Perception is the use of sensing information to understand the world. This could, for example, involve using vision to detect features of the world and use these features to determine the position of a robot and obstacles or packages to be picked up. Perception tends to be ambiguous and noisy. It is difficult to build a model of a three-dimensional world based on a single image of the world.

A more promising approach is to combine the agent's prediction of the world state with sensing information. This can take a number of forms:

- If both the noise of forward prediction and sensor noise are modeled, the next belief state can be estimated using Bayes' rule (page 227). This is known as **filtering** (page 267).
- With more complicated sensors such as vision, a model can be used to predict where visual features can be found, and then vision can be used to look for these features close to the predicted location. This makes the vision task much simpler and vision can greatly reduce the errors in position arising from forward prediction alone.

A control problem is **separable** if the best action can be obtained by first finding the best model of the world and then using that model to determine the best action. Unfortunately, most control problems are not separable. This means that the agent should consider multiple models to determine what to

do, and what information it gets from the world depends on what it will do with that information. Usually, there is no best model of the world that is independent of what the agent will do with the model.

2.4 Embedded and Simulated Agents

There are a number of ways an agent's controller can be used:

- An **embedded agent** is one that is run in the real world, where the actions are carried out in a real domain and where the sensing comes from a domain.
- A **simulated agent** is one that is run with a simulated body and environment; that is, where a program takes in the commands and returns appropriate percepts. This is often used to debug a controller before it is deployed.
- A **agent system model** is where there are models of the controller (which may or may not be the actual code), the body, and the environment that can answer questions about how the agent will behave. Such a model can be used to prove properties of agents before they are built, or it can be used to answer hypothetical questions about an agent that may be difficult or dangerous to answer with the real agent.

Each of these is appropriate for different purposes.

- Embedded mode is how the agent must run to be useful.
- A simulated agent is useful to test and debug the controller when many design options must be explored and building the body is expensive or when the environment is dangerous or inaccessible. It also allows us to test the agent under unusual combinations of conditions that may be difficult to arrange in the actual world.

  How good the simulation is depends on how good the model of the environment is. Models always have to abstract some aspect of the world. Appropriate abstraction is important for simulations to be able to tell us whether the agent will work in a real environment.
- A model of the agent, a model of the set of possible environments, and a specification of correct behavior allow us to prove theorems about how the agent will work in such environments. For example, we may want to prove that a robot running a particular controller will always get within a certain distance of the target, that it will never get stuck in mazes, or that it will never crash. Of course, whether what is proved turns out to be true depends on how accurate the models are.
- Given a model of the agent and the environment, some aspects of the agent can be left unspecified and can be adjusted to produce the desired or optimal behavior. This is the general idea behind optimization and planning.
- In reinforcement learning (page 463), the agent improves its performance while interacting with the real world.

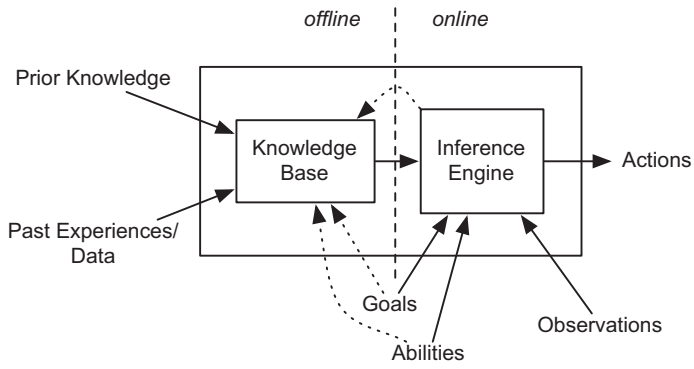

Figure 2.9: Offline and online decomposition of an agent

2.5 Acting with Reasoning

The previous sections assumed that an agent has some belief state that it maintains through time. For an intelligent agent, the belief state can be very complex, even for a single layer.

Experience in studying and building intelligent agents has shown that an intelligent agent requires some internal representation of its belief state. **Knowledge** is the information about a domain that is used for solving problems in that domain. Knowledge can include general knowledge that can be applied to particular situations. Thus, it is more general than the beliefs about a specific state. A **knowledge-based system** is a system that uses knowledge about a domain to act or to solve problems.

Philosophers have defined knowledge as true, justified belief. AI researchers tend to use the terms knowledge and belief more interchangeably. Knowledge tends to mean general information that is taken to be true. Belief tends to mean information that can be revised based on new information. Often beliefs come with measures of how much they should be believed and models of how the beliefs interact. In an AI system, knowledge is typically not necessarily true and is justified only as being useful. This distinction often becomes blurry when one module of an agent may treat some information as true but another module may be able to revise that information.

Figure 2.9 shows a refinement of Figure 1.3 (page 11) for a knowledge-based agent. A **knowledge base** is built offline and is used online to produce actions. This decomposition of an agent is orthogonal to the layered view of an agent; an intelligent agent requires both hierarchical organization and knowledge bases.

Online (page 17), when the agent is acting, the agent uses its knowledge base, its observations of the world, and its goals and abilities to choose what to do and to update its knowledge base. The **knowledge base** is its **long-term**

memory, where it keeps the knowledge that is needed to act in the future. This knowledge comes from prior knowledge and is combined with what is learned from data and past experiences. The **belief state** (page 48) is the **short-term memory** of the agent, which maintains the model of current environment needed between time steps. A clear distinction does not always exist between general knowledge and specific knowledge; for example, an outside delivery robot could learn general knowledge about a particular city. There is feedback from the inference engine to the knowledge base, because observing and acting in the world provide more data from which to learn.

Offline, before the agent has to act, it can build the knowledge base that is useful for it to act online. The role of the offline computation is to make the online computation more efficient or effective. The knowledge base is built from prior knowledge and from data of past experiences (either its own past experiences or data it has been given). Researchers have traditionally considered the case involving lots of data and little prior knowledge in the field of **machine learning**. The case of lots of prior knowledge and little or no data from which to learn has been studied under the umbrella of **expert systems**. However, for most non-trivial domains, the agent must use whatever information is available, and so it requires both rich prior knowledge and lots of data.

The goals and abilities are given offline, online, or both, depending on the agent. For example, a delivery robot could have general goals of keeping the lab clean and not damaging itself or other objects, but it could get other delivery goals at runtime. The online computation can be made more efficient if the knowledge base is tuned for the particular goals and abilities. However, this is often not possible when the goals and abilities are only available at runtime.

Figure 2.10 (on the next page) shows more detail of the interface between the agents and the world.

2.5.1 Design Time and Offline Computation

The knowledge base required for online computation can be built initially at design time and then augmented offline by the agent.

An **ontology** is a specification of the meaning of the symbols used in an information system. It specifies what is being modeled and the vocabulary used in the system. In the simplest case, if the agent is using explicit state-based representation with full observability, the ontology specifies the mapping between the world and the state. Without this mapping, the agent may know it is in, say, state 57, but, without the ontology, this information is just a meaningless number to another agent or person. In other cases, the ontology defines the features or the individuals and relationships. It is what is needed to convert raw sense data into something meaningful for the agent or to get meaningful input from a person or another knowledge source.

Ontologies are built by communities, often independently of a particular knowledge base or specific application. It is this shared vocabulary that

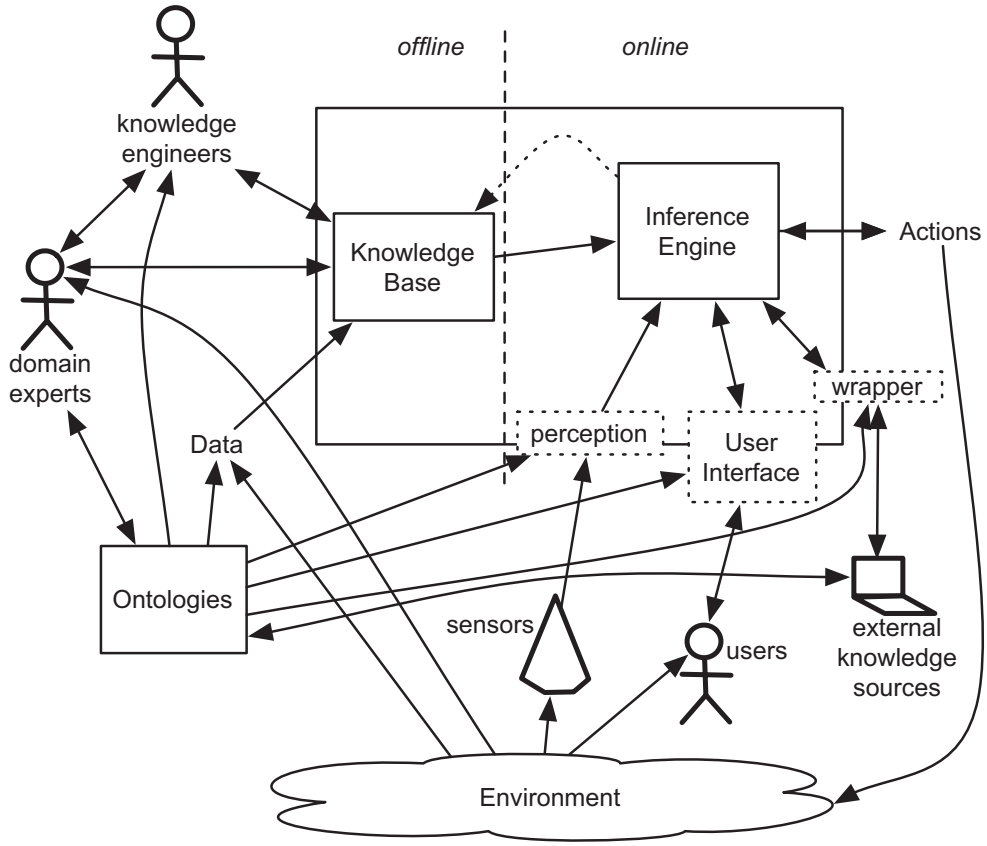

Figure 2.10: Internals of an agent, showing roles

allows for effective communication and interoperation of the data from multiple sources (sensors, humans, and databases). Ontologies for the case of individuals and relationships are discussed in Section 13.3 (page 563).

The ontology logically comes before the data and the prior knowledge: we require an ontology to have data or to have knowledge. Without an ontology, data are just sequences of bits. Without an ontology, a human does not know what to input; it is the ontology that gives the data meaning. Often the ontology evolves as the system is being developed.

The ontology specifies a level or levels of abstraction. If the ontology changes, the data must change. For example, a robot may have an ontology of obstacles (e.g., every physical object is an obstacle to be avoided). If the ontology is expanded to differentiate people, chairs, tables, coffee mugs, and the like, different data about the world are required.

The knowledge base is typically built offline from a combination of expert knowledge and data. It is usually built before the agent knows the particulars of the environment in which it must act. Maintaining and tuning the knowledge base is often part of the online computation.

Offline, there are three major **role**s involved with a knowledge-based system:

- **Software engineers** build the inference engine and user interface. They typically know nothing about the contents of the knowledge base. They need not be experts in the use of the system they implement; however, they must be experts in the use of a programming language like Java, Lisp, or Prolog rather than in the knowledge representation language of the system they are designing.
- **Domain experts** are the people who have the appropriate prior knowledge about the domain. They know about the domain, but typically they know nothing about the particular case that may be under consideration. For example, a medical domain expert would know about diseases, symptoms, and how they interact but would not know the symptoms or the diseases of the particular patient. A delivery robot domain expert may know the sort of individuals that must be recognized, what the battery meter measures, and the costs associated with various actions. Domain experts typically do not know the particulars of the environment the agent would encounter – for example, the details of the patient for the diagnostic assistant or the details of the room a robot is in.

  Domain experts typically do not know about the internal workings of the AI system. Often they have only a semantic view of the knowledge (page 161) and have no notion of the algorithms used by the inference engine. The system should interact with them in terms of the domain, not in terms of the steps of the computation. For example, it is unreasonable to expect that domain experts could debug a knowledge base if they were presented with traces of how an answer was produced. Thus, it is not appropriate to have debugging tools for domain experts that merely trace the execution of a program.
- **Knowledge engineers** design, build, and debug the knowledge base in consultation with domain experts. They know about the details of the system and about the domain through the domain expert. They know nothing about any particular case. They should know about useful inference techniques and how the complete system works.

The same people may fill multiple roles: A domain expert who knows about AI may act as a knowledge engineer; a knowledge engineer may be the same person who writes the system. A large system may have many different software engineers, knowledge engineers, and experts, each of whom may specialize in part of the system. These people may not even know they are part of the system; they may publish information for anyone to use.

Offline, the agent can combine the expert knowledge and the data. At this stage, the system can be tested and debugged. The agent is able to do computation that is not particular to the specific instance. For example, it can compile parts of the knowledge base to allow more efficient inference.

2.5.2 Online Computation

Online, the information about the particular case becomes available, and the agent has to act. The information includes the observations of the domain and often information about the available actions and the preferences or goals. The agent can get observations from sensors, users, and other information sources (such as web sites), but we assume it does not have access to the domain experts or knowledge engineer.

An agent typically has much more time for offline computation than for online computation. However, during online computation it can take advantage of particular goals and particular observations.

For example, a medical diagnosis system only has the details of a particular patient online. Offline, it can acquire knowledge about how diseases and symptoms interact and do some debugging and compilation. It can only do the computation about a particular patient online.

Online the following roles are involved:

- A **user** is a person who has a need for expertise or has information about individual cases. Users typically are not experts in the domain of the knowledge base. They often do not know what information is needed by the system. Thus, it is unreasonable to expect them to volunteer the information about a particular case. A simple and natural interface must be provided because users do not typically understand the internal structure of the system. They often, however, must make an informed decision based on the recommendation of the system; thus, they require an explanation of why the recommendation is appropriate.
- **Sensors** provide information about the environment. For example, a thermometer is a sensor that can provide the current temperature at the location of the thermometer. Sensors may be more sophisticated, such as a vision sensor. At the lowest level, a vision sensor may simply provide an array of 720×480 pixels at 30 frames per second. At a higher level, a vision system may be able to answer specific questions about the location of particular features, whether some type of individual is in the environment, or whether some particular individual is in the scene. An array of microphones can be used at a low level of abstraction to provide detailed vibration information. It can also be used as a component of a higher-level sensor to detect an explosion and to provide the type and the location of the explosion.

  Sensors come in two main varieties. A **passive sensor** continuously feeds information to the agent. Passive sensors include thermometers, cameras, and microphones. The designer can typically choose where the sensors are or where they are pointing, but they just feed the agent information. In contrast, an **active sensor** is controlled or asked for information. Examples of an active sensor include a medical probe able to answer specific questions about a patient or a test given to a student in an intelligent tutoring system. Often sensors that are passive sensors at lower levels of abstraction can be seen as active sensors at higher levels of abstraction. For example, a camera could be asked whether a particular person is in the room. To do this it may need

to zoom in on the faces in the room, looking for distinguishing features of the person.

- An **external knowledge source**, such as a web site or a database, can typically be asked questions and can provide the answer for a limited domain. An agent can ask a weather web site for the temperature at a particular location or an airline web site for the arrival time of a particular flight. The knowledge sources have various protocols and efficiency trade-offs. The interface between an agent and an external knowledge source is called a **wrapper**. A wrapper translates between the representation the agent uses and the queries the external knowledge source is prepared to handle. Often wrappers are designed so that the agent can ask the same query of multiple knowledge sources. For example, an agent may want to know about airplane arrivals, but different airlines or airports may require very different protocols to access that information. When web sites and databases adhere to a common ontology, they can be used together because the same symbols have the same meaning. Having the same symbols mean the same thing is called **semantic interoperability**. When they use different ontologies, there must be mappings between the ontologies to allow them to interoperate.

Again, these roles are separate, even though the people in these roles may overlap. The domain expert, for example, may act as a user to test or debug the system. Each of the roles has different requirements for the tools they need. The tools that explain to a user how the system reached a result can be the same tools that the domain experts use to debug the knowledge.

2.6 Review

The main points you should have learned from this chapter are as follows:

- An agent system is composed of an agent and an environment.
- Agents have sensors and actuators to interact with the environment.
- An agent is composed of a body and interacting controllers.
- Agents are situated in time and must make decisions of what to do based on their history of interaction with the environment.
- An agent has direct access not to its history, but to what it has remembered (its belief state) and what it has just observed. At each point in time, an agent decides what to do and what to remember based on its belief state and its current observations.
- Complex agents are built modularly in terms of interacting hierarchical layers.
- An intelligent agent requires knowledge that is acquired at design time, offline or online.

2.7 References and Further Reading

The model of agent systems is based on the constraint nets of Zhang and Mackworth [1995], also on Rosenschein and Kaelbling [1995]. The hierarchcical control is based on Albus [1981] and the subsumption architecture of Brooks [1986]. *Turtle Geometry*, by Abelson and DiSessa [1981], investigates mathematics from the viewpoint of modeling simple reactive agents. Luenberger [1979] is a readable introduction to the classical theory of agents interacting with environments. Simon [1996] argues for the importance of hierarchical control.

For more detail on agent control see Dean and Wellman [1991], Latombe [1991], and Agre [1995].

The methodology for building intelligent agents is discussed by Haugeland [1985], Brooks [1991], Kirsh [1991b], and Mackworth [1993].

Qualitative reasoning is described by Forbus [1996] and Kuipers [2001]. Weld and de Kleer [1990] contains many seminal papers on qualitative reasoning. See also Weld [1992] and related discussion in the same issue. For a recent review see Price, Travé-Massuyàs, Milne, Ironi, Forbus, Bredeweg, Lee, Struss, Snooke, Lucas, Cavazza, and Coghill [2006].

2.8 Exercises

Exercise 2.1 Section 2.3 (page 50) argued that it was impossible to build a representation of a world that is independent of what the agent will do with it. This exercise lets you evaluate this argument.

Choose a particular world, for example, what is on some part of your desk at the current time.

i) Get someone to list all of the things that exist in this world (or try it yourself as a thought experiment).

ii) Try to think of twenty things that they missed. Make these as different from each other as possible. For example, the ball at the tip of the rightmost ballpoint pen on the desk, or the spring in the stapler, or the third word on page 66 of a particular book on the desk.

iii) Try to find a thing that cannot be described using natural language.

iv) Choose a particular task, such as making the desk tidy, and try to write down all of the things in the world at a level of description that is relevant to this task.

Based on this exercise, discuss the following statements:

(a) What exists in a world is a property of the observer.

(b) We need ways to refer to individuals other than expecting each individual to have a separate name.

(c) What individuals exist is a property of the task as well as of the world.

(d) To describe the individuals in a domain, you need what is essentially a dictionary of a huge number of words and ways to combine them to describe

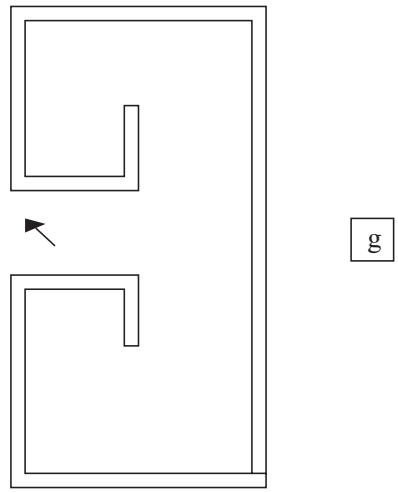

Figure 2.11: A robot trap

individuals, and this should be able to be done independently of any particular domain.

Exercise 2.2 Explain why the middle layer in Example 2.5 (page 55) must have both the previous target position and the current target position as inputs. Suppose it had only one of these as input; which one would it have to be, and what would the problem with this be?

Exercise 2.3 The definition of the target position in Example 2.6 (page 56) means that, when the plan ends, the robot will just keep the last target position as its target position and keep circling forever. Change the definition so that the robot goes back to its home and circles there.

Exercise 2.4 The obstacle avoidance implemented in Example 2.5 (page 55) can easily get stuck.

(a) Show an obstacle and a target for which the robot using the controller of Example 2.5 (page 55) would not be able to get around (and it will crash or loop).

(b) Even without obstacles, the robot may never reach its destination. For example, if it is next to its target position, it may keep circling forever without reaching its target. Design a controller that can detect this situation and find its way to the target.

Exercise 2.5 Consider the "robot trap" in Figure 2.11.

(a) Explain why it is so tricky for a robot to get to location g. You must explain what the current robot does as well as why it is difficult to make a more sophisticated robot (e.g., one that follows the wall using the "right-hand rule": the robot turns left when it hits an obstacle and keeps following a wall, with the wall always on its right) to work.

(b) An intuition of how to escape such a trap is that, when the robot hits a wall, it follows the wall until the number of right turns equals the number of left turns. Show how this can be implemented, explaining the belief state, the belief-state transition function, and the command function.

Exercise 2.6 When the user selects and moves the current target location, the robot described in this chapter travels to the original position of that target and does not try to go to the new position. Change the controller so that the robot will try to head toward the current location of the target at each step.

Exercise 2.7 The current controller visits the locations in the *todo* list sequentially.

(a) Change the controller so that it is opportunistic; when it selects the next location to visit, it selects the location that is closest to its current position. It should still visit all of the locations.

(b) Give one example of an environment in which the new controller visits all of the locations in fewer time steps than the original controller.

(c) Give one example of an environment in which the original controller visits all of the locations in fewer time steps than the modified controller.

(d) Change the controller so that, at every step, the agent heads toward whichever target location is closest to its current position.

(e) Can the controller from part (d) get stuck in a loop and never reach a target in an example where the original controller will work? Either give an example in which it gets stuck in a loop and explain why it cannot find a solution, or explain why it does not get into a loop.

Exercise 2.8 Change the controller so that the robot senses the environment to determine the coordinates of a location. Assume that the body can provide the coordinates of a named location.

Exercise 2.9 Suppose you have a new job and must build a controller for an intelligent robot. You tell your bosses that you just have to implement a command function and a state transition function. They are very skeptical. Why these functions? Why only these? Explain why a controller requires a command function and a state transition function, but not other functions. Use proper English. Be concise.