2

The aim of this chapter is to give you an understanding of what agents are, and some of the issues associated with building them. In later chapters, we will see specific approaches to building agents.

An obvious way to open this chapter would be by presenting a definition of the term *agent*. After all, this is a book about multiagent systems - surely we must all agree on what an agent is? Sadly, ~~there~~ is no universally accepted definition of the term agent, and indeed there is much ongoing debate and controversy on this very subject. Essentially, while there is a general consensus that *autonomy* is central to the notion of agency, there is little agreement beyond this. Part of the difficulty is that various attributes associated with agency are of differing importance for different domains. Thus, for some applications, the ability of agents to *learn* from their experiences is of paramount importance; for other applications, learning is not only unimportant, it is ~~undesirable[1]~~.

Nevertheless, some sort of definition is important - otherwise, there is a danger that the term will lose all meaning. The definition presented here is adapted from Wooldridge and Jennings (1995).

> An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.

[1] Michael Georgeff, the main architect of the PRS agent system discussed in later chapters, gives the example of an air-traffic control system he developed; the clients of the system would have been horrified at the prospect of such a system modifying its behaviour at run time...

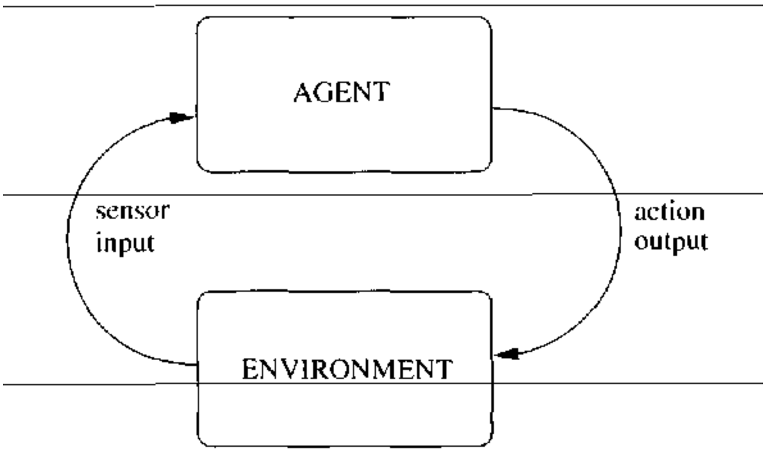

Figure 2.1 An agent in its environment. The agent takes sensory input from the environment, and produces as output actions that affect it. The interaction is usually an ongoing, non-terminating one.

Figure 2.1 gives an abstract view of an agent. In this diagram, we can see the action output generated by the agent in order to affect its environment. In most domains of reasonable complexity, an agent will not have *complete* control over its environment. It will have at best *partial* control, in that it can *influence* it. From the point of view of the agent, this means that the same action performed twice in apparently identical circumstances might appear to have entirely different effects, and in particular, it may *fail* to have the desired effect. Thus agents in all but the most trivial of environments must be prepared for the possibility of *failure.* We can sum this situation up formally by saying that environments are in general assumed to be *non-deterministic.*

Normally, an agent will have a repertoire of actions available to it. This set of possible actions represents the agents *effectoric capability:* its ability to modify its environments. Note that not all actions can be performed in all situations. For example, an action 'lift table' is only applicable in situations where the weight of the table is sufficiently small that the agent *can* lift it. Similarly, the action 'purchase a Ferrari' will fail if insufficient funds are available to do so. Actions therefore have *preconditions* associated with them, which define the possible situations in which they can be applied.

The key problem facing an agent is that of deciding *which* of its actions it should perform in order to best satisfy its design objectives. *Agent architectures,* of which we shall see many examples later in this book, are really software architectures for decision-making systems that are embedded in an environment. At this point, it is worth pausing to consider some examples of agents (though not, as yet, intelligent agents).

Control systems

First, any *control* system can be viewed as an agent. A simple (and overused) example of such a system is a thermostat. Thermostats have a sensor for detecting room temperature. This sensor is directly embedded within the environment (i.e. the room), and it produces as output one of two signals: one that indicates that the temperature is too low, another which indicates that the temperature is OK. The actions available to the thermostat are 'heating on' or 'heating off'. The action 'heating on' will generally have the effect of raising the room temperature, but this cannot be a *guaranteed* effect - if the door to the room is open, for example, switching on the heater may have no effect. The (extremely simple) decision-making component of the thermostat implements (usually in electro-mechanical hardware) the following rules:

too cold —• heating on,
temperature OK —• heating off.

More complex environment control systems, of course, have considerably richer decision structures. Examples include autonomous space probes, fly-by-wire aircraft, nuclear reactor control systems, and so on.

Software demons

Second, most software demons (such as background processes in the Unix operating system), which monitor a software environment and perform actions to modify it, can be viewed as agents. An example is the X Windows program `xbiff`. This utility continually monitors a user's incoming email, and indicates via a GUI icon whether or not they have unread messages. Whereas our thermostat agent in the previous example inhabited a *physical* environment - the physical world - the `xbiff` program inhabits a *software* environment. It obtains information about this environment by carrying out software functions (by executing system programs such as `ls`, for example), and the actions it performs are software actions (changing an icon on the screen, or executing a program). The decision-making component is just as simple as our thermostat example.

To summarize, agents are simply computer systems that are capable of autonomous action in some environment in order to meet their design objectives. An agent will typically sense its environment (by physical sensors in the case of agents situated in part of the real world, or by software sensors in the case of software agents), and will have available a repertoire of actions that can be executed to modify the environment, which may appear to respond non-deterministically to the execution of these actions.

2.1 Environments

Russell and Norvig suggest the following classification of environment properties (Russell and Norvig, 1995, p. 46).

Accessible versus inaccessible. An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state. Most real-world environments (including, for example, the everyday physical world and the Internet) are not accessible in this sense.

Deterministic versus non-deterministic. A deterministic environment is one in which any action has a single guaranteed effect - there is no uncertainty about the state that will result from performing an action.

Static versus dynamic. A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agent. In contrast, a dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control. The physical world is a highly dynamic environment, as is the Internet.

Discrete versus continuous. An environment is discrete if there are a fixed, finite number of actions and percepts in it.

We begin our discussion with accessibility. First, note that in extreme cases, the laws of physics prevent many environments from being completely accessible. For example, it may be that as I write, the surface temperature at the North Pole of Mars is -100 °C, but the laws of physics will prevent me from knowing this fact for some ~~time~~. This information is thus *inaccessible* to me. More mundanely, in almost any realistic environment uncertainty is inherently present.

The more accessible an environment is, the simpler it is to build agents that operate effectively within it. The reason for this should be self-evident. Ultimately, a 'good' agent is one that makes the 'right' decisions. The quality of decisions that an agent can make is clearly dependent on the quality of the information available to it. If little, or inaccurate information is available, then the agent's decision is uninformed, and is hence likely to be poor. As more complete and accurate information becomes available, the potential to make a good decision increases.

The next source of complexity we consider is *determinism.* An environment is deterministic if the outcome of any action performed is uniquely defined, and non-deterministic otherwise. Non-determinism can seem an unusual property to attribute to environments. For example, we usually imagine that software environments, governed as they are by precise rules, are paradigms of determinism. Non-determinism captures several important aspects of such environments as follows.

- Non-determinism captures the fact that agents have a limited 'sphere of influence' - they have at best partial control over their environment.

- Similarly, actions are typically performed by agents in order to bring about some desired state of affairs. Non-determinism captures the fact that actions can *fail* to have the desired result.

Clearly, deterministic environments are preferable from the point of view of the agent designer to non-deterministic environments. If there is never any uncertainty about the outcome of some particular action, then an agent need never stop to determine whether or not a particular action had a particular outcome, and thus whether or not it needs to reconsider its course of action. In particular, in a deterministic environment, an agent designer can assume that the actions performed by an agent will always succeed: they will never fail to bring about their intended effect.

Unfortunately, as Russell and Norvig (1995) point out, if an environment is sufficiently complex, then the fact that it is *actually* deterministic is not much help. To all intents and purposes, it may as well be non-deterministic. In practice, almost all realistic environments must be regarded as non-deterministic from an agent's perspective.

Non-determinism is closely related to *dynamism.* Early artificial intelligence research on action selection focused on planning algorithms - algorithms that, given a description of the initial state of the environment, the actions available to an agent and their effects, and a goal state, will generate a plan {i.e. a sequence of actions) such that when executed from the initial environment state, the plan will guarantee the achievement of the goal (Allen *et al.*, 1990). However, such planning algorithms implicitly assumed that the environment in which the plan was being executed was *static* - that it did not change except through the performance of actions by the agent. Clearly, many environments (including software environments such as computer operating systems, as well as physical environments such as the real world), do not enjoy this property - they are dynamic, with many processes operating concurrently to modify the environment in ways that an agent has no control over.

From an agent's point of view, dynamic environments have at least two important properties. The first is that if an agent performs no external action between times t_0 and t_1, then it cannot assume that the environment at t_1 will be the same as it was at time t_0. This means that in order for the agent to select an appropriate action to perform, it must perform *information gathering* actions to determine the stale of the environment (Moore, 1990). In a static environment, there is no need for such actions. The second property is that other processes in the environment can 'interfere' with the actions it attempts to perform. The idea is essentially the concept of interference in concurrent systems theory (Ben-Ari, 1990). Thus if an agent checks that the environment has some property φ and then starts executing some action α on the basis of this information, it cannot in general guarantee that the environment will continue to have property φ while it is executing α.

iese properties suggest that static environments will be inherently simpler to design agents for than dynamic ones. First, in a static environment, an agent need only ever perform information gathering actions *once.* Assuming the information it gathers correctly describes the environment, and that it correctly understands the effects of its actions, then it can accurately *predict* the effects of its actions on the environment, and hence how the state of the environment will evolve. (This is in fact how most artificial intelligence planning algorithms work (Lifschitz, 1986).) Second, in a static environment, an agent never needs to worry about *synchronizing* or *coordinating* its actions with those of other processes in the environment (Bond and Gasser, 1988).

The final distinction made in Russell and Norvig (1995) is between *discrete* and *continuous* environments. A discrete environment is one that can be guaranteed to only ever be in a finite number of discrete states; a continuous one may be in uncountably many states. Thus the game of chess is a discrete environment - there are only a finite (albeit very large) number of states of a chess game. Russell and Norvig (1995) give taxi driving as an example of a continuous environment.

Discrete environments are simpler to design agents for than continuous ones, for several reasons. Most obviously, digital computers are themselves discrete-state systems, and although they can simulate continuous systems to any desired degree of accuracy, there is inevitably a mismatch between the two types of systems. Some information must be lost in the mapping from continuous environment to discrete representation of that environment. Thus the information a discrete-state agent uses in order to select an action in a continuous environment will be made on the basis of information that is inherently approximate. Finally, with finite discrete state environments, it is in principle possible to enumerate all possible states of the environment and the optimal action to perform in each of these states. Such a lookup table approach to agent design is rarely possible in practice, but it is at least *in principle* possible for finite, discrete state environments.

In summary, the most complex general class of environments are those that are inaccessible, non-deterministic, dynamic, and continuous. Environments that have these properties are often referred to as *open* (Hewitt, 1986).

Environmental properties have a role in determining the complexity of the agent design process, but they are by no means the only factors that play a part. The second important property that plays a part is the nature of the *interaction* between agent and environment.

Originally, software engineering concerned itself with what are known as 'functional' systems. A functional system is one that simply takes some input, performs some computation over this input, and eventually produces some output. Such systems may formally be viewed as functions / : / — O from a set I of inputs to a set O of outputs. The classic example of such a system is a compiler, which can be viewed as a mapping from a set / of legal source programs to a set O of corresponding object or machine code programs.

One of the key attributes of such functional systems is that they *terminate*. This means that, formally, their properties can be understood in terms of preconditions and postconditions (Hoare, 1969). The idea is that a precondition *qp* represents what must be true of the program's environment in order for that program to operate correctly. A postcondition ψ represents what will be true of the program's environment after the program terminates, assuming that the precondition was satisfied when execution of the program commenced. A program is said to be completely correct with respect to precondition φ and postcondition ψ if it is guaranteed to terminate when it is executed from a state where the precondition is satisfied, and, upon termination, its postcondition is guaranteed to be satisfied. Crucially, it is assumed that the agent's environment, as characterized by its precondition φ, is *only* modified through the actions of the program itself. As we noted above, this assumption does not hold for many environments.

Although the internal complexity of a functional system may be great (e.g. in the case of a compiler for a complex programming language such as Ada), functional programs are, in general, comparatively simple to correctly and efficiently engineer. For example, functional systems lend themselves to design methods based on 'divide and conquer'. Top-down stepwise refinement (Jones, 1990) is an example of such a method. Semi-automatic refinement techniques are also available, which allow a designer to refine a high-level (formal) specification of a functional system down to an implementation (Morgan, 1994).

Unfortunately, many computer systems that we desire to build are not functional in this sense. Rather than simply computing a function of some input and then terminating, many computer systems are *reactive,* in the following sense:

> Reactive systems are systems that cannot adequately be described by the *relational* or *functional* view. The relational view regards programs as functions.. .from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) in terms of their on-going behaviour.. .[E]very concurrent system.. .must be studied by behavioural means. This is because each individual module in a concurrent system is a reactive subsystem, interacting with its own environment which consists of the other modules.
>
> (Pnueli, 1986)

There are at least three current usages of the term *reactive system* in computer science. The first, oldest, usage is that by Pnueli and followers (see, for example, Pnueli (1986), and the description above). Second, researchers in AI planning take a reactive system to be one that is capable of responding rapidly to changes in its environment - here the word 'reactive' is taken to be synonymous with 'responsive' (see, for example, Kaelbling, 1986). More recently, the term has been used to denote systems which respond directly to the world, rather than reason explicitly about it (see, for example, Connah and Wavish, 1990).

Reactive systems are harder to engineer than functional ones. Perhaps the most important reason for this is that an agent engaging in a (conceptually) non-terminating relationship with its environment must continually make *local* decisions that have *global* consequences. Consider a simple printer controller agent. The agent continually receives requests to have access to the printer, and is allowed to grant access to any agent that requests it, with the proviso that it is only allowed to grant access to one agent at a time. At some time, the agent reasons that it will give control of the printer to process p_1,rather than p_2, but that it will grant *p?* access at some later time point. This seems like a reasonable decision, when considered in isolation. But if the agent *always* reasons like this, it will *never* grant p_2 access. This issue is known as *fairness* (Francez, 1986). In other words, a decision that seems entirely reasonable in a local context can have undesirable effects when considered in the context of the system's entire history. This is a simple example of a complex problem. In general, the decisions made by an agent have long-term effects, and it is often difficult to understand such long-term effects.

One possible solution is to have the agent explicitly reason about and predict the behaviour of the system, and thus any temporally distant effects, at run-time. But it turns out that such prediction is extremely hard.

Russell and Subramanian (1995) discuss the essentially identical concept of *episodic* environments. In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of the agent in different episodes. An example of an episodic environment would be a mail sorting system (Russell and Subramanian, 1995). As with reactive systems, episodic interactions are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current episode - it does not need to reason about the interactions between this and future episodes.

Another aspect of the interaction between agent and environment is the concept of *real time.* Put at its most abstract, a real-time interaction is simply one in which time plays a part in the evaluation of an agent's performance (Russell and Subramanian, 1995, p. 585). It is possible to identify several different types of real-time interactions:

- those in which a decision must be made about what action to perform within some specified time bound;
- those in which the agent must bring about some state of affairs as quickly as possible;
- those in which an agent is required to repeat some task, with the objective being to repeat the task as often as possible.

If time is not an issue, then an agent can deliberate for as long as required in order to select the 'best' course of action in any given scenario. Selecting the best course of action implies search over the space of all possible courses of action, in order to find the 'best'. Selecting the best action in this way will take time exponential in the number of actions available to the agent[2]. It goes without saying that for any

[2] If the agent has *n* actions available to it, then it has n! courses of action available to it (assuming no duplicate actions).

realistic environment, such deliberation is not viable. Thus any realistic system must be regarded as real-time in some sense.

Some environments are real-time in a much stronger sense than this. For example, the PRS, one of the best-known agent systems, had fault diagnosis on NASA's Space Shuttle as its initial application domain (Georgeff and Lansky, 1987). In order to be of any use, decisions in such a system must be made in milliseconds.

rents

We are not used to thinking of thermostats or Unix demons as agents, and cerlainly not as *intelligent* agents. So, when do we consider an agent to be intelligent? The question, like the question '*whatis intelligence?*' itself, is not an easy one to answer. One way of answering the question is to list the kinds of capabilities that we might expect an intelligent agent to have. The following list was suggested in Wooldridge and Jennings (1995).

Reactivity. Intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives.

Proactiveness. Intelligent agents are able to exhibit goal-directed behaviour by *taking the initiative* in order to satisfy their design objectives.

Social ability. Intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

These properties arc more ~~demanding~~ than they might at first appear. To ~~see~~ why, let us consider them in turn. First, consider *proactiveness:* goal-directed behaviour. It is not hard to build a system that exhibits goal-directed behaviour - we do it every time we write a procedure in Pascal, a function in C, or a method in Java. When we write such a procedure, we describe it in terms of the *assumptions* on which it relies (formally, its *precondition)* and the *effect* it has if the assumptions are valid (its *postcondition).* The effects of the procedure are its *goal:* what the author of the software intends the procedure to achieve. If the precondition holds when the procedure is invoked, then we expect that the procedure will execute *correctly:* that it will terminate, and that upon termination, the postcondition will be true, i.e. the goal will be achieved. This is goal-directed behaviour: the procedure is simply a plan or recipe for achieving the goal. This programming model is fine for many environments. For example, it works well when we consider ~~func~~tional systems, as discussed above.

But for non-functional systems, this simple model of goal-directed programming is not acceptable, as it makes some important limiting assumptions. In particular, it assumes that the environment *does not change* while the procedure is executing. If the environment does change, and, in particular, if the assumptions (precondition) underlying the procedure become false while the procedure is executing, then the behaviour of the procedure may not be defined - often, it will simply crash. Also, it is assumed that the goal, that is, the reason for executing the procedure, remains valid at least until the procedure terminates. If the goal does *not* remain valid, then there is simply no reason to continue executing the procedure.

In many environments, neither of these assumptions are valid. In particular, in domains that are *too complex* for an agent to observe completely, that are *multiagent* (i.e. they are populated with more than one agent that can change the environment), or where there is *uncertainty* in the environment, these assumptions are not reasonable. In such environments, blindly executing a procedure without regard to whether the assumptions underpinning the procedure are valid is a poor strategy. In such dynamic environments, an agent must be *reactive*, in just the way that we described above. That is, it must be responsive to events that occur in its environment, where these events affect either the agent's goals or the assumptions which underpin the procedures that the agent is executing in order to achieve its goals.

As we have seen, building purely goal-directed systems is not hard. As we shall see later, building *purely reactive* systems - ones that *continually* respond to their environment - is also not difficult. However, what turns out to be hard is building a system that achieves an effective *balance* between goal-directed and reactive behaviour. We want agents that will attempt to achieve their goals systematically, perhaps by making use of complex procedure-like patterns of action. But we do not want our agents to continue blindly executing these procedures in an attempt to achieve a goal either when it is clear that the procedure will not work, or when the goal is for some reason no longer valid. In such circumstances, we want our agent to be able to react to the new situation, in time for the reaction to be of some use. However, we do not want our agent to be *continually* reacting, and hence never focusing on a goal long enough to actually achieve it.

On reflection, it should come as little surprise that achieving a good balance between goal-directed and reactive behaviour is hard. After all, it is comparatively rare to find humans that do this very well. This problem - of effectively integrating goal-directed and reactive behaviour - is one of the key problems facing the agent designer. As we shall see, a great many proposals have been made for how to build agents that can do this - but the problem is essentially still open.

Finally, let us say something about *social ability*, the final component of flexible autonomous action as defined here. In one sense, social ability is trivial: every day, millions of computers across the world routinely exchange information with both humans and other computers. But the ability to exchange bit streams is not really social ability. Consider that in the human world, comparatively few of our meaningful goals can be achieved without the *cooperation* of other people, who cannot be assumed to *share* our goals - in other words, they are themselves autonomous, with their own agenda to pursue. To achieve our goals in such situations, we must *negotiate* and *cooperate* with others. We may be required to understand and reason about the goals of others, and to perform actions (such as paying them money) that we would not otherwise choose to perform, in order to get them to cooperate with us, and achieve our goals. This type of social ability is much more complex, and much less well understood, than simply the ability to exchange binary information. Social ability in general (and topics such as negotiation and cooperation in particular) are dealt with elsewhere in this book, and will not therefore be considered here. In this chapter, we will be concerned with the decision making of *individual* intelligent agents in environments which may be dynamic, unpredictable, and uncertain, but do not contain other agents.

2.3 Agents and Objects

Programmers familiar with object-oriented languages such as Java, C++, or Smalltalk sometimes fail to see anything novel in the idea of agents. When one stops to consider the relative properties of agents and objects, this is perhaps not surprising.

> There is a tendency.. .to think of objects as 'actors' and endow them with human-like intentions and abilities. It's tempting to think about objects 'deciding' what to do about a situation, [and] 'asking' other objects for information.... Objects are not passive containers for state and behaviour, but are said to be the agents of a program's activity.
>
> (NeXT Computer Inc., 1993, p. 7)

Objects are defined as computational entities that *encapsulate* some state, are able to perform actions, or *methods* on this state, and communicate by message passing. While there are obvious similarities, there are also significant differences between agents and objects. The first is in the degree to which agents and objects tonomous. Recall that the defining characteristic of object-oriented programming is the principle of encapsulation - the idea that objects can have control over their own internal state. In programming languages like Java, we can declare instance variables (and methods) to be `private`, meaning they are only accessible from within the object. (We can of course also declare them `public`, meaning that they can be accessed from anywhere, and indeed we must do this for methods so that they can be used by other objects. But the use of public instance variables is usually considered poor programming style.) In this way, an object can be thought of as exhibiting autonomy over its state: it has control over it. But an object does not exhibit control over its *behaviour*. That is, if a method `m` is made available for other objects to invoke, then they can do so whenever they wish - once an object has made a method public, then it subsequently has no control over whether or not that method is executed. Of course, an object *vnust* make methods available to other objects, or else we would be unable to build a system out of them. This is not normally an issue, because if we build a system, then we design the objects that go in it, and they can thus be assumed to share a 'common goal'. But in many types of multiagent system (in particular, those that contain agents built by different organizations or individuals), no such common goal can be assumed. It cannot be taken for granted that an agent *i* will execute an action (method) *a* just because another agent *j* wants it to - *a* may not be in the best interests of *i*. We thus do not think of agents as invoking methods upon one another, but rather as *requesting* actions to be performed. If *j* requests *i* to perform *a*, then *i* may perform the action or it may not. The locus of control with respect to the decision about whether to execute an action is thus different in agent and object systems. In the object-oriented case, the decision lies with the object that invokes the method. In the agent case, the decision lies with the agent that receives the request. This distinction between objects and agents has been nicely summarized in the following slogan.

> Objects do it for free; agents do it because they want to.

Of course, there is nothing to stop us implementing agents using object-oriented techniques. For example, we can build some kind of decision making about whether to execute a method into the method itself, and in this way achieve a stronger kind of autonomy for our objects. The point is that autonomy of this kind is not a component of the basic object-oriented model.

The second important distinction between object and agent systems is with respect to the notion of flexible (reactive, proactive, social) autonomous behaviour. The standard object model has nothing whatsoever to say about how to build systems that integrate these types of behaviour. Again, one could object that we can build object-oriented programs that *do* integrate these types of behaviour. But this argument misses the point, which is that the standard object-oriented programming model has nothing to do with these types of behaviour.

The third important distinction between the standard object model and our view of agent systems is that agents are each considered to have their own thread of control - in the standard object model, there is a single thread of control in the system. Of course, a lot of work has recently been devoted to *concurrency* in object-oriented programming. For example, the Java language provides built-in constructs for multi-threaded programming. There are also many programming languages available (most of them admittedly prototypes) that were specifically designed to allow concurrent object-based programming. But such languages do not capture the idea of agents as *autonomous* entities. Perhaps the closest that the object-oriented community comes is in the idea of *active objects*.

> An active object is one that encompasses its own thread of control... Active objects are generally autonomous, meaning that they can exhibit some behaviour without being operated upon by another object. Pas-objects, on the other hand, can only undergo a state change when explicitly acted upon.

3ooch, 1994, p. 91)

Thus active ~~objects~~ arc essentially ~~agents~~ that do not necessarily ~~have the~~ ability to exhibit flexible autonomous behaviour.

To summarize, the traditional view of an object and our view of an agent have at least three distinctions:

- agents embody a stronger notion of autonomy than objects, and, in particular, they decide for themselves whether or not to perform an action on request from another agent;
- agents are capable of flexible (reactive, proactive, social) behaviour, and the standard object model has nothing to say about such types of behaviour; and
- a multiagent system is inherently multi-threaded, in that each agent is assumed to have at least one thread of control.

2.4 Agents and Expert Systems

Expert systems were the most important AI technology of the 1980s (Hayes-Roth *et al.*, 1983). An expert system is one that is capable of solving problems or giving advice in some knowledge-rich domain (Jackson, 1986). A classic example of an expert system is MYCIN, which was intended to assist physicians in the treatment of blood infections in humans. ~~MYCIN~~ worked by a process of interacting with a user in order to present the system with a number of (symbolically represented) facts, which the system then used to derive some conclusion. MYCIN acted very much as a *consultant:* it did not operate directly on humans, or indeed any other environment. Thus perhaps the most important distinction between agents and expert systems is that expert systems like MYCIN are inherently *disembodied.* By this, I mean that they do not interact directly with any environment: they get their information not via sensors, but through a user acting as middle man. In the same way, they do not *act* on any environment, but rather give feedback or advice to a third party. In addition, expert systems are not generally capable of cooperating with other agents.

In summary, the main differences between agents and expert systems are as follows:

- 'classic' expert systems are disembodied - they are not coupled to any environment in which they act, but rather act through a user as a 'middleman';
- expert systems are not generally capable of reactive, proactive behaviour; and
- expert systems are not generally equipped with social ability, in the sense of cooperation, coordination, and negotiation.

Despite these differences, some expert systems (particularly those that perform real-time control tasks) look very much like agents. A good example is the ARCHON system, discussed in Chapter 9 (Jennings *et al.*, 1996a).

2.5 Agents as Intentional Systems

One common approach adopted when discussing agent systems is the *intentional stance.* With this approach, we 'endow' agents with *mental states:* ~~beliefs,~~ desires, wishes, hope, and so on. The rationale for this approach is as follows. When explaining human activity, it is often useful to make statements such as the following.

> Janine took her umbrella because she *believed* it was going to rain.
> Michael worked hard because he *wanted* to finish his book.

These statements make use of a *folk psychology,* by which human behaviour is predicted and explained through the attribution of *attitudes,* such as believing and wanting (as in the above examples), hoping, fearing, and so on (see, for example, Stich (1983, p. 1) for a discussion of folk psychology). This folk psychology is well established: most people reading the above statements would say they found their meaning entirely clear, and would not give them a second glance.

The attitudes employed in such folk psychological descriptions are called the *intentional* notions[3]. The philosopher Daniel Dennett has coined the term *intentional system* to ~~describe~~ entities 'whose behaviour can be predicted by the method of attributing belief, desires and rational acumen' (Dennett, 1978, 1987, p. 49). Dennett identifies different 'levels' of intentional system as follows.

> A *first-order* intentional system has beliefs and desires (etc.) but no beliefs and desires *about* beliefs and desires. ... A *second-order* intentional system is more sophisticated; it has beliefs and desires (and no doubt other intentional states) about beliefs and desires (and other intentional states) - both those of others and its own.
>
> (Dennett, 1987, p. 243)

One can carry on this hierarchy of intentionality as far as required.

Now we have been using phrases like belief, desire, intention to talk about computer programs. An obvious question is whether it is legitimate or useful to attribute beliefs, desires, and so on to artificial agents. Is this not just anthropomorphism? McCarthy, among others, has argued that there are occasions when the *intentional stance* is appropriate as follows.

[3] Unfortunately, the word 'intention' is used in several different ways in logic and the philosophy of mind. First, there is the BDI-like usage, as in 'I intended to kill him'. Second, an intentional notion is one of the attitudes, as above. Finally, in logic, the word intension (with an 's') means the internal conrent of a concept, as opposed to its extension. In what follows, the intended meaning should always be clear from context.

> To ascribe *beliefs, free will, intentions, consciousness, abilities,* or *wants* to a machine is legitimate when such an ascription expresses the same information about the machine that it expresses about a person. It is useful when the ascription helps us understand the structure of the machine, its past or future behaviour, or how to repair or improve it. It is perhaps never logically required even for humans, but expressing reasonably briefly what is actually known about the state of the machine in a particular situation may require mental qualities or qualities isomorphic to them. Theories of belief, knowledge and wanting can be constructed for machines in a simpler setting than for humans, and later applied to humans. Ascription of mental qualities is most straightforward for machines of known structure such as thermostats and computer operating systems, but is most useful when applied to entities whose structure is incompletely known.
>
> (McCarthy, 1978) (The underlining is from Shoham (1990).)

What objects can be described by the intentional stance? As it turns out, almost any automaton can. For example, consider a light switch as follows.

> It is perfectly coherent to treat a light switch as a (very cooperative) agent with the capability of transmitting current at will, who invariably transmits current when it believes that we want it transmitted and not otherwise; flicking the switch is simply our way of communicating our desires.
>
> (Shoham, 1990, p. 6)

And yet most adults in the modern world would find such a description absurd - perhaps even infantile. Why is this? The answer seems to be that while the intentional stance description is perfectly consistent with the observed behaviour of a light switch, and is internally consistent,

> .. .it does not *buy us anything*, since we essentially understand the mechanism sufficiently to have a simpler, mechanistic description of its behaviour.
>
> (Shoham, 1990, p. 6)

Put crudely, the more we know about a system, the less we need to rely on animistic, intentional explanations of its behaviour - Shoham observes that the move from an intentional stance to a technical description of behaviour correlates well with Piaget's model of child development, and with the scientific development of humankind generally (Shoham, 1990). Children will use animistic explanations of objects - such as light switches - until they grasp the more abstract technical concepts involved. Similarly, the evolution of science has been marked by a gradual move from theological/animistic explanations to mathematical ones. My own experiences of teaching computer programming suggest that, when faced with completely unknown phenomena, it is not only children who adopt animistic explanations. It is often easier to teach some computer concepts by using explanations such as 'the computer does not know...', than to try to teach abstract principles first.

An obvious question is then, if we have alternative, perhaps less contentious ways of explaining systems: why should we bother with the intentional stance? Consider the alternatives available to us. One possibility is to characterize the behaviour of a complex system by using the *physical stance* (Dennett, 1996, p. 36). The idea of the physical stance is to start with the original configuration of a system, and then use the laws of physics to predict how this system will behave.

> When I predict that a stone released from my hand will fall to the ground, I am using the physical stance. I don't attribute beliefs and desires to the stone; I attribute mass, or weight, to the stone, and rely on the law of gravity to yield my prediction.
>
> (Dennett, 1996, p. 37)

Another alternative is the *design stance.* With the design stance, we use knowledge of what purpose a system is supposed to fulfil in order to predict how it behaves. Dennett gives the example of an alarm clock (see pp. 37-39 of Dennett, 1996). When someone presents us with an alarm clock, we do not need to make use of physical laws in order to understand its behaviour. We can simply make use of the fact that all alarm clocks are designed to wake people up if we set them with a time. No understanding of the clock's mechanism is required to justify such an understanding - we know that *all* alarm clocks have this behaviour.

However, with very complex systems, even if a complete, accurate picture of the system's architecture and working *is* available, a physical or design stance explanation of its behaviour may not be practicable. Consider a computer. Although we might have a complete technical description of a computer available, it is hardly practicable to appeal to such a description when explaining why a menu appears when we click a mouse on an icon. In such situations, it may be more appropriate to adopt an intentional stance description, if that description is consistent, and simpler than the alternatives.

Note that the intentional stance is, in computer science terms, nothing more than an *abstraction tool.* It is a convenient shorthand for talking about complex systems, which allows us to succinctly predict and explain their behaviour without having to understand how they actually work. Now, much of computer science is concerned with looking for good abstraction mechanisms, since these allow system developers to *manage complexity* with greater ease. The history of programming languages illustrates a steady move away from low-level machine-oriented views of programming towards abstractions that are closer to human experience. Procedural abstraction, abstract data types, and, most recently, objects are examples of this progression. So, why not use the intentional stance as an abstraction tool in computing - to explain, understand, and, crucially, *program* complex computer systems?

For many researchers this idea of programming computer systems in terms of mentalistic notions such as belief, desire, and intention is a key component of agent-based systems.

2.6 Abstract Architectures for Intelligent Agents

We can easily formalize the abstract view of agents presented so far. First, let us assume that the environment may be in any of a finite set E of discrete, instantaneous states:

Notice that whether or not the environment 'really is' discrete in this way is not too important for our purposes: it is a (fairly standard) modelling assumption, which we can justify by pointing out that any *continuous* environment can be modelled by a discrete environment to any desired degree of accuracy.

Agents are assumed to have a repertoire of possible actions available to them, which transform the state of the environment. Let

$$Ac = \{\alpha, \mathrm{of}', \ldots\}$$

be the (finite) set of actions.

The basic model of agents interacting with their environments is as follows. The ment starts in some state, and the agent begins by choosing an action to perform on that state. As a result of this action, the environment can respond with a number of possible states. However, only one state will *actually* result - though of course, the agent does not know in advance which it will be. On the basis of this second state, the agent again chooses an action to perform. The environment responds with one of a set of possible states, the agent then chooses another action, and so on.

A *run, r,* of an agent in an environment is thus a sequence of interleaved environment states and actions:

$$r : e_0$$

Let

- $\mathcal{R}$ be the set of all such possible finite sequences (over E and Ac);
- $\mathcal{R}^{Ac}$ be the subset of these that end with an action; and
- $\mathcal{R}^{E}$ be the subset of these that end with an environment state.

We will use $r, r', \ldots$ to stand for members of $\mathcal{R}$.

In order to represent the effect that an agent's actions have on an environment, we introduce a *state transformer* function (cf. Fagin *et al.*, 1995, p. 154):

$$^{Ac} \rightarrow \wp(E).$$

Thus a state transformer function maps a run (assumed to end with the action of an agent) to a set of possible environment stales - those that could result from performing the action.

There are two important points to note about this definition. First, environments are assumed to be *history dependent.* In other words, the next state of an environment is not solely determined by the action performed by the agent and the current state of the environment. The actions made *earlier* by the agent also play a part in determining the current state. Second, note that this definition allows for *non-determinism* in the environment. There is thus *uncertainty* about the result of performing an action in some state.

If T(r) $=$ 0 (where r is assumed to end with an action), then there are no possible successor states to *r.* In this case, we say that the system has *ended* its run. We will also assume that all runs eventually terminate.

~~Formally~~, we say an environment *Env* is a ~~triple~~ *Env* = ~~$\langle E, e_0, \tau\rangle$, where *E*~~ is a set of environment states, e_0 *e E* is an initial state, and T is a state transformer function.

We now need to introduce a model of the agents that inhabit systems. We model agents as functions which map runs (assumed to end with an environment state) to actions (cf. Russell and Subramanian, 1995, pp. 580, 581):

$$Ag : \mathcal{R}^E \to Ac.$$

Thus an agent makes a decision about what action to perform based on the history of the system that it has witnessed to date.

Notice that while environments are implicitly non-deterministic, agents are assumed to be deterministic. Let $\mathcal{AG}$ be the set of all agents.

We say a *system* is a pair containing an agent and an environment. Any system will have associated with it a set of possible runs; we denote the set of runs of agent *Ag* in environment *Env* by ~~$\mathcal{R}(Ag, Env)$~~.For simplicity, we will assume that $\mathcal{R}(Ag,Env)$ contains only *terminated* runs, i.e. runs r such that *r* has no possible successor states: T(r) = 0. (We will thus not consider infinite runs for now.)

Formally, a sequence

$$(e_0, \alpha_0, e_1, \alpha_1, e_2, \dots)$$

represents a run of an agent *Ag* in environment *Env* = *(E,* e_0, T) if

(1) e_0 is the initial state of *Env*,

(2) $\alpha_0 = Ag(e_0)$; and

(3) for $u > 0$,

$$e_u \in \tau((e_0, \alpha_0, \dots, \alpha_{u-1})),$$

where

$$\alpha_u = Ag((e_0, \alpha_0, \dots, e_u)).$$

Two agents Ag_1 and Ag_2 are said to be *behaviourally equivalent* with respect to environment *Env* if and only if $\mathcal{R}(Ag_1, Env) = \mathcal{R}(Ag_2, Env)$, and simply behaviourally equivalent if and only if they are behaviourally equivalent with respect to all environments.

Notice that so far, I have said nothing at all about how agents are actually implemented; we will return to this issue later.

Purely reactive agents

Certain types of agents decide what to do without reference to their history. They base their decision making entirely on the present, with no reference at all to the past. We will call such agents *purely reactive*, since they simply respond directly to their environment. (Sometimes they are called *tropistic* agents (Genesereth and Nilsson, 1987): tropism is the tendency of plants or animals to react to certain stimulae.)

Formally, the behaviour of a purely reactive agent can be represented by a function

$$Ag : E \to Ac.$$

It should be easy to see that for every purely reactive agent, there is an equivalent 'standard' agent, as discussed above; the reverse, however, is not generally the case.

Our thermostat agent is an example of a purely reactive agent. Assume, without loss of generality, that the thermostat's environment can be in one of two states - either too cold, or temperature OK. Then the thermostat is simply defined as follows:

$$Ag(e) = \begin{cases} \text{heater off} & \text{if } e = \text{temperature OK}, \\ \text{heater on} & \text{otherwise.} \end{cases}$$

Perception

Viewing agents at this abstract level makes for a pleasantly simple analysis. However, it does not help us to construct them. For this reason, we will now begin to *refine* our abstract model of agents, by breaking it down into sub-systems in exactly the way that one does in standard software engineering. As we refine our view of agents, we find ourselves making *design choices* that mostly relate to the subsystems that go to make up an agent - what data and control structures will be present. An *agent architecture* is essentially a map of the internals of an agent - its data structures, the operations that may be performed on these data structures, and the control flow between these data structures. Later in this book, we will discuss a number of different types of agent architecture, with very different views on the data structures and algorithms that will be present within an agent. ln the remainder of this section, however, we will survey some fairly high-level design decisions. The first of these is the separation of an agent's decision function into *perception* and *action* subsystems: see Figure 2.2.

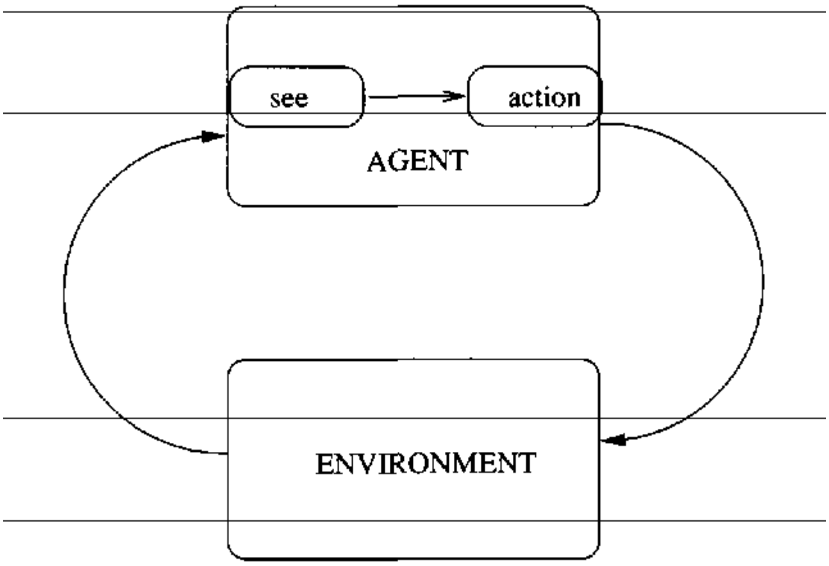

Figure 2.2 Perception and action subsystems.

The idea is that the function *see* captures the agent's ability to observe its environment, whereas the *action* function represents the agent's decision-making process. The *see* function might be implemented in hardware in the case of an agent situated in the physical world: for example, it might be a video camera or an infrared sensor on a mobile robot. For a software agent, the sensors might be system commands that obtain information about the software environment, such as ls, finger, or suchlike. The *output* of the *see* function is a *percept* - a perceptual input. Let *Per* be a (non-empty) set of percepts. Then *see* is a function

$$see : E \to Per$$

which maps environment states to percepts, and *action* is a function

$$action : Per^* \to Ac$$

which maps sequences of percepts to actions. An agent *Ag* is now considered to be a pair $Ag = \langle see, action \rangle$ consisting of a *see* function and an *action* function.

These simple definitions allow us to explore some interesting properties of agents and perception. Suppose that we have two environment states, e_1 e E and e_2 £ E, such that $e_1 \neq e_2$, but *see{e\)* = $see(e_2)$. Then two *different* environment states are mapped to the *same* percept, and hence the agent would receive the same perceptual information from different environment states. As far as the agent is concerned, therefore, e_1 and e_2 are *indistinguishable.* To make this example concrete, let us return to the thermostat example. Let *x* represent the statement

'the room temperature is OK'

and let *y* represent the statement

'John Major is Prime Minister'.

If these are the only two facts about our environment that we are concerned with, then the set *E* of environment states contains exactly four elements:

$$E = \{\underbrace{\{\neg x, \neg y\}}_{e_1}, \underbrace{\{\neg x, y\}}_{e_2}, \underbrace{\{x, \neg y\}}_{e_3}, \underbrace{\{x, y\}}_{e_4}\}.$$

Thus in state *e*, the room temperature is not OK, and John Major is not Prime Minister; in state e_2, the room temperature is not OK, and John Major *is* Prime Minister. Now, our thermostat is sensitive *only* to temperatures in the room. This room temperature is not causally related to whether or not John Major is Prime Minister. Thus the states where John Major is and is not Prime Minister are literally *indistinguishable* to the thermostat. Formally, the *see* function for the thermostat would have two percepts in its range, p_1 and p_2, indicating that the temperature is too cold or OK, respectively. The *see* function for the thermostat would behave as follows:

$$see(e) = \begin{cases} p_1 & \text{if } e = e_1 \text{ or } e = e_2, \\ p_2 & \text{if } e = e_3 \text{ or } e = e_4. \end{cases}$$

Given two environment states *e* ∈ *E* and *e'* e *E*, let us write e ~ *e'* if *see(e)* = *see(e').* It is not hard to see that '~' is an *equivalence relation* over environment states, which partitions *E* into mutually indistinguishable sets of states. Intuitively, the coarser these equivalence classes are, the less effective is the agent's perception. If |~| = |*E*| (i.e. the number of distinct percepts is equal to the number of different environment states), then the agent can distinguish *every* state - the agent has perfect perception in the environment; it is *omniscient.* At the other extreme, if |~| — 1, then the agent's perceptual ability is non-existent - it cannot distinguish between *any* different states. In this case, as far as the agent is concerned, all environment states are identical.

Agents with state

We have so far modelled an agent's decision function as from *sequences* of environment states or percepts to actions. This allows us to represent agents whose decision making is influenced by history. However, this is a somewhat unintuitive representation, and we shall now replace it by an equivalent, but somewhat more natural, scheme. The idea is that we now consider agents that *maintain state* - see Figure 2.3.

These agents have some internal data structure, which is typically used to record information about the environment state and history. Let / be the set of all internal states of the agent. An agent's decision-making process is then based, at least in part, on this information. The perception function *see* for a state-based agent is unchanged, mapping environment states to percepts as before:

$$see : E \to Per.$$

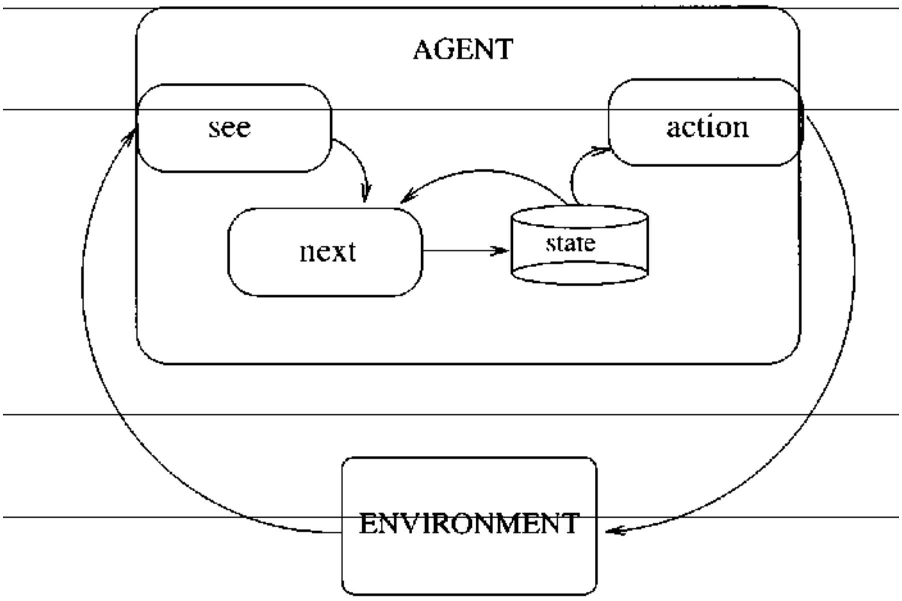

Figure 2.3 Agents that maintain state.

The action-selection function *action* is defined as a mapping

$$action : I \to Ac$$

from internal states to actions. An additional function *next* is introduced, which maps an internal state and percept to an internal state:

$$next\ I \times Per \to I.$$

The behaviour of a state-based agent can be summarized in the following way. The agent starts in some initial internal state i_0. It then observes its environment state *e,* and generates a percept *see(e).* The internal state of the agent is then updated via the *next* function, becoming set to $next(i_0, see(e))$The action selected by the agent is then $action(next(i_0, see(e)$This action is then performed, and the agent enters another cycle, perceiving the world via *see,* updating its state via *next,* and choosing an action to perform via *action.*

It is worth observing that state-based ~~agents~~ as defined here are in fact no more powerful than the standard agents we introduced earlier. In fact, they are *identical* in their expressive power - every state-based agent can be transformed into a standard agent that is behaviourally equivalent.

2.7 How to Tell an Agent What to Do

We do not (usually) build agents for no reason. We build them in order to carry out *tasks* for us. In order to get the agent to do the task, we must somehow communicate the desired task to the agent. This implies that the task to be carried out must be *specified* by us in some way. An obvious question is how to specify these tasks: how to tell the agent what to do. One way to specify the task would be simply to write a program for the agent to execute. The obvious advantage of this approach is that we are left in no uncertainty about what the agent will do; it will do exactly what we told it to, and no more. But the very obvious disadvantage is that we have to think about exactly how the task will be carried out ourselves - if unforeseen circumstances arise, the agent executing the task will be unable to respond accordingly. So, more usually, we want to *tell our agent what to do without telling it how to do it.* One way of doing this is to define tasks *indirectly,* via some kind of *performance measure.* There are several ways in which such a performance measure can be defined. The first is to associate *utilities* with states of the environment.

Utility functions

A utility is a numeric value representing how 'good' the state is: the higher the utility, the better. The task of the agent is then to bring about states that maximize utility - we do not specify to the agent how this is to be done. In this approach, a task specification would simply be a function

$$u : E \to \mathbb{R}$$

which associates a real value with every environment state. Given such a performance measure, we can then define the overall utility of an agent in some particular environment in several different ways. One (pessimistic) way is to define the utility of the agent as the utility of the *worst* state that might be encountered by the agent; another might be to define the overall utility as the average utility of all states encountered. There is no right or wrong way: the measure depends upon the kind of task you want your agent to carry out.

The main disadvantage of this approach is that it assigns utilities to *local* states; it is difficult to specify a *long-term* view when assigning utilities to individual states. To get around this problem, we can specify a task as a function which assigns a utility not to individual states, but to runs themselves:

$$\boldsymbol{u} : \mathcal{R} \to \mathbb{R}.$$

If we are concerned with agents that must operate independently over long periods of time, then this approach appears more appropriate to our purposes. One well-known example of the use of such a utility function is in the Tileworld (Pollack, 1990). The Tileworld was proposed primarily as an experimental environment for evaluating agent architectures. It is a simulated two-dimensional grid environment on which there are agents, tiles, obstacles, and holes. An agent can move in four directions, up, down, left, or right, and if it is located next to a tile, it can push it. An obstacle is a group of immovable grid cells: agents are not allowed to travel freely through obstacles. Holes have to be filled up with tiles by the agent. An agent scores points by filling holes with tiles, with the aim being to fill as many

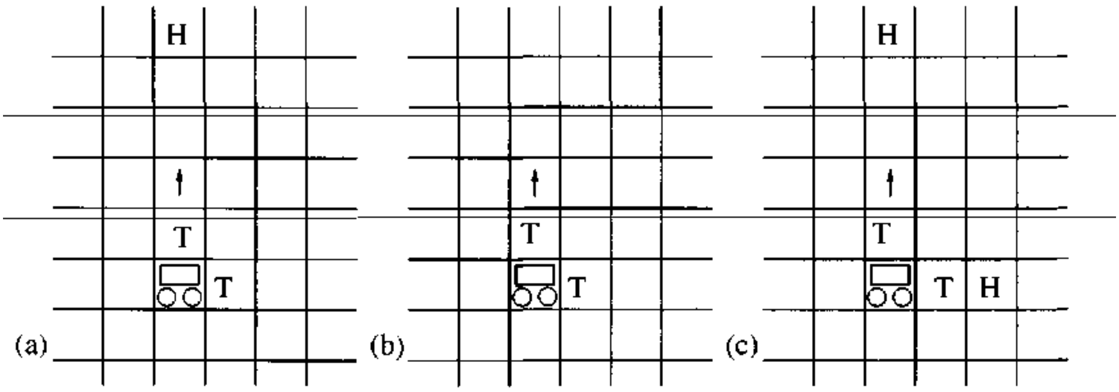

Figure 2.4 Three scenarios in the Tileworld are (a) the agent detects a hole ahead, and begins to push a tile towards it; (b) the hole disappears before the agent can get to it - the agent should recognize this change in the environment, and modify its behaviour appropriately; and (c) the agent was pushing a tile north, when a hole appeared to its right; it would do better to push the tile to the right, than to continue to head north.

holes as possible. The Tileworld is an example of a *dynamic* environment: starting in some randomly generated world state, based on parameters set by the experimenter, it changes over time in discrete steps, with the random appearance and disappearance of holes. The experimenter can set a number of Tileworld parameters, including the frequency of appearance and disappearance of tiles, obstacles, and holes; and the choice between hard bounds (instantaneous) or soft bounds (slow decrease in value) for the disappearance of holes. In the Tileworld, holes appear randomly and exist for as long as their *life expectancy,* unless they disappear because of the agent's actions. The interval between the appearance of successive holes is called the *~~hole~~ gestation time.* The performance of an agent in the Tileworld is measured by running the Tileworld testbed for a predetermined number of time steps, and measuring the number of holes that the agent succeeds in filling. The performance of an agent on some particular run is then defined as

$$u(r) = \frac{\text{number of holes filled in } r}{\text{number of holes that appeared in } r}$$

This gives a normalized performance measure in the range 0 (the agent did not succeed in filling even one hole) to 1 (the agent succeeded in filling every hole that appeared). Experimental error is eliminated by running the agent in the environ~~ment~~ a number of times, and computing the average of the performance.

Despite its simplicity, the Tileworld allows us to examine several important capabilities of agents. Perhaps the most important of these is the ability of an agent to *react* to changes in the environment, and to *exploit opportunities* when they arise. For example, suppose an agent is pushing a tile to a hole (Figure 2.4(a)), when this hole disappears (Figure 2.4(b)). At this point, pursuing the original objective is pointless, and the agent would do best if it noticed this change, and as a consequence 'rethought' its original objective. To illustrate what I mean by ~~rec~~ognizing opportunities, suppose that in the same situation, a hole appears to the right of the agent (Figure 2.4(c)). The agent is more likely to be able to fill this hole than its originally planned one, for the simple reason that it only has to push the tile one step, rather than four. All other things being equal, the chances of the hole on the right still being there when the agent arrives are four times greater.

Assuming that the utility function u has some upper bound to the utilities it assigns (i.e. that there exists a $k \in \mathbb{R}$such that for all r e $\mathcal{R}$, we have $u(r) \leqslant k)$, then we can talk about *optimal* agents: the optimal agent is the one that maximizes expected utility.

Let us write $P(r \mid Ag,Env)$ to denote the probability that run r occurs when agent Ag is placed in environment Env. Clearly,

$$\sum_{r \in \mathcal{R}(Ag,Env)} P(r \mid Ag, Env) = 1.$$

Then the optimal agent Ag_{opt} in an environment Env is defined as the one that *maximizes expected utility:*

$$Ag_{\text{opt}} = \arg \max_{Ag \in \mathcal{AG}} \sum_{r \in \mathcal{R}(Ag,Env)} u(r)P(r \mid Ag, Env). \tag{2.1}$$

This idea is essentially identical to the notion of maximizing expected utility in *decision theory* (see Russell and Norvig, 1995, p. 472),

Notice that while (2.1) tells us the properties of the desired agent Ag_{opt}, it sadly does not give us any clues about how to *implement* this agent. Worse still, some agents cannot be implemented on some actual machines. To see this, simply note that agents as we have considered them so far are just abstract mathematical functions $Ag : \mathcal{R}^E \to Ac$. These definitions take no account of (for example) the amount of memory required to implement the function, or how complex the computation of this function is. It is quite easy to define functions that cannot actually be computed by any real computer, and so it is just as easy to define agent functions that cannot ever actually be implemented on a real computer.

Russell and Subramanian (1995) introduced the notion of *bounded optimal* agents in an attempt to try to address this issue. The idea is as follows. Suppose m is a particular computer - for the sake of argument, say it is the Dell Latitude L400 I am currently typing on: a laptop with 128 MB of RAM, 6 GB of disk space, and a 500 MHz Pentium III processor. There are only a certain set of programs that can run on this machine. For example, any program requiring more than 128 MB RAM clearly cannot run on it. In just the same way, only a certain subset of the set of all agents $\mathcal{AG}$ can be implemented on this machine. Again, any agent Ag that required more than 128 MB RAM would not run. Let us write $\mathcal{AG}_m$ to denote the subset of $\mathcal{AG}$ that can be implemented on m:

$$- \{Ag \mid Ag \in \mathcal{AG} \text{ and } Ag \text{ can be implemented on } m\}.$$

Now, assume we have machine (i.e. computer) m, and we wish to place this machine in environment Env; the task we wish m to carry out is defined by utility function $u : \mathcal{R} \to \mathbb{R}$. Then we can replace Equation (2.1) with the following, which more precisely defines the properties of the desired agent Ag_{opt}:

$$Ag_{\text{opt}} = \arg \max_{Ag \in \mathcal{AG}_m} \sum_{r \in \mathcal{R}(Ag, Env)} u(r) P(r \mid Ag, Env). \tag{2.2}$$

The subtle change in (2.2) is that we are no longer looking for our agent from the set of all possible agents $\mathcal{AG}$, but from the set $\mathcal{AG}_m$ of agents that can actually be implemented on the machine that we have for the task.

Utility-based approaches to specifying tasks for agents have several disadvantages. The most important of these is that it is very often difficult to derive an appropriate utility function; the Tileworld is a useful environment in which to experiment with agents, but it represents a gross simplification of real-world scenarios. The second is that usually we find it more convenient to talk about tasks in terms of 'goals to be achieved' rather than utilities. This leads us to what I call *predicate* task specifications.

Predicate task specifications

Put simply, a predicate task specification is one where the utility function acts as a *predicate* over runs. Formally, we will say a utility function $u : \mathcal{R} \to \mathbb{R}$ is a predicate if the range of u is the set $\{0, 1\}$, that is, if u guarantees to assign a run either 1 ('true') or 0 ('false'). A run $r \in \mathcal{R}$ will be considered to satisfy the specification *u* if $u(r) = 1$, and fails to satisfy the specification otherwise.

We will use Ψ to denote a predicate specification, and write $\Psi(r)$ to indicate that run r $\in \mathcal{R}$ which satisfies Ψ. In other words, *¥{r)* is true if and only if $u(r) = 1$. For the moment, we will leave aside the questions of what form a predicate task specification might take.

Task environments

A *task environment* is defined to be a pair $\langle Env, \Psi \rangle$, where *Env* is an environment, and

$$\Psi : \mathcal{R} \to \{0, 1\}$$

is a predicate over runs. Let $\mathcal{TE}$ be the set of all task environments. A task environment thus specifies:

- the properties of the system the agent will inhabit (i.e. the environment *Env);* and also
- the criteria by which an agent will be judged to have either failed or succeeded in its task (i.e. the specification Ψ).

Given a task environment $\langle Env, \Psi \rangle$, we write $\mathcal{R}_\Psi(Ag, Env)$ to denote the set of all runs of the agent *Ag* in the environment *Env* that satisfy Ψ. Formally,

$$\mathcal{R}_\Psi(Ag, Env) = \{r \mid r \in n(Ag,Env) \quad \text{and}$$

We then say that an agent *Ag* succeeds in task environment *{Env,¥}* if

$$\mathcal{R}_{\Psi}(Ag, Env) = \mathcal{R}(Ag, Env).$$

In other words, *Ag* succeeds in $\langle Env, \Psi \rangle$ if every run of *Ag* in *Env* satisfies specification ¥, i.e. if

Vr $\in \mathcal{R}(Ag, Env)$ we have *¥{r).*

Notice that this is in one sense a *pessimistic* definition of success, as an agent is only deemed to succeed if every possible run of the agent in the environment satisfies the specification. An alternative, *optimistic* definition of success is that the agent succeeds if *at least one* run of the agent satisfies Ψ:

3r $\in \mathcal{R}(Ag, Env)$ such that $\Psi(r)$.

If required, we could easily modify the definition of success by extending the state transformer function T to include a probability distribution over possible outcomes, and hence induce a probability distribution over runs. We can then define the success of an agent as the probability that the specification ¥ is satisfied by the agent. As before, let $P(r \mid Ag, Env)$ denote the probability that run r occurs if agent *Ag* is placed in environment *Env.* Then the probability *P{Y \ Ag,Env)* that ¥ is satisfied by *Ag* in *Env* would simply be

$$P(\Psi \mid Ag, Env) = \sum_{r \in \mathcal{R}_{\Psi}(Ag, Env)} P(r \mid Ag, Env).$$

The notion of a predicate task specification may seem a rather abstract way of describing tasks for an agent to carry out. In fact, it is a generalization of certain very common forms of tasks. Perhaps the two most common types of tasks that we encounter are *achievement tasks* and *maintenance tasks.*

(1) **Achievement tasks.** Those of the form 'achieve state of affairs φ'.

laintenance tasks. Those of the form 'maintain state of affairs ψ'.

Intuitively, an achievement task is specified by a number of *goal states;* the agent is required to bring about one of these goal states (we do not care which one - all are considered equally good). Achievement tasks are probably the most commonly studied form of task in AI. Many well-known AI problems (e.g. the Blocks World) are achievement tasks. A task specified by a predicate ¥ is an achievement task if we can identify some subset $\mathcal{G}$ of environment states £ such that *¥(r)* is true just in case one or more of $\mathcal{G}$ occur in r; an agent is successful if it is guaranteed to bring about one of the states $\mathcal{G}$, that is, if every run of the agent in the environment results in one of the states $\mathcal{G}$.

Formally, the task environment *{Env, ¥)* specifies an achievement task if and only if there is some set $\mathcal{G} \subseteq E$ such that for all *r* $\in \mathcal{R}(Ag, Env)$ the predicate *¥{r)* is true if and only if there exists some *e* $\in \mathcal{G}$ such that *e e r.* We refer to the set $\mathcal{G}$ of an achievement task environment as the *goal states* of the task; we use $\langle Env, \mathcal{G} \rangle$ to denote an achievement task environment with goal states $\mathcal{G}$ and environment Env.

A useful way to think about achievement tasks is as the agent *playing a game* against the environment. In the terminology of game theory (Binmore, 1992), this is exactly what is meant by a 'game against nature'. The environment and agent both begin in some state; the agent takes a turn by executing an action, and the environment responds with some state; the agent then takes another turn, and so on. The agent 'wins' if it can *force* the environment into one of the goal states $\mathcal{G}$.

Just as many tasks can be characterized as problems where an agent is required to bring about some state of affairs, so many others can be classified as problems where the agent is required to *avoid* some state of affairs. As an extreme example, consider a nuclear reactor agent, the purpose of which is to ensure that the reactor never enters a 'meltdown' state. Somewhat more mundanely, we can imagine a software agent, one of the tasks of which is to ensure that a particular file is never simultaneously open for both reading and writing. We refer to such task environments as *maintenance* task environments.

A task environment with specification Ψ is said to be a maintenance task environment if we can identify some subset S of environment states, such that $\Psi(r)$ is false if any member of $\mathcal{B}$ occurs in *r*, and true otherwise. Formally, $\langle Env, \Psi \rangle$ is a maintenance task environment if there is some 3 $\subset E$ such that $\Psi(r)$ if and only if for all $e \in$ 3, we have $e \notin r$ for all *r* E *3i(Ag,Env).* We refer to $\mathcal{B}$ as the *failure set.* As with achievement task environments, we write $\langle Env, \mathcal{B} \rangle$ to denote a maintenance task environment with environment *Env* and failure set 3.

It is again useful to think of maintenance tasks as games. This time, the agent wins if it manages to *avoid* all the states in $\mathcal{B}$. The environment, in the role of opponent, is attempting to force the agent into 3; the agent is successful if it has a winning strategy for avoiding $\mathcal{B}$.

More complex tasks might be specified by *combinations* of achievement and maintenance tasks. A simple combination might be 'achieve any one of states $\mathcal{G}$ while avoiding all states 3'. More complex combinations are of course also possible.

2.8 Synthesizing Agents

Knowing that there exists an agent which will succeed in a given task environment is helpful, but it would be more helpful if, knowing this, we also had such an agent to hand. How do we obtain such an agent? The obvious answer is to 'manually' implement the agent from the specification. However, there are at least two other possibilities (see Wooldridge (1997) for a discussion):

(1) we can try to develop an algorithm that will *automatically* ~~*synthesize*~~such agents for us from task environment specifications; or

(2) we can try to ~~develop~~ an algorithm that will *directly execute* agent ~~specifica-~~tions in order to produce the appropriate behaviour.

In this section, I briefly consider these possibilities, focusing primarily on agent synthesis.

Agent synthesis is, in effect, automatic programming: the goal is to have a program that will take as input a task environment, and from this task environment automatically generate an agent that succeeds in this environment. Formally, an agent synthesis algorithm *syn* can be understood as a function

$$syn : \mathcal{TE} \to (\mathcal{AG} \cup \{\perp\}).$$

Note that the function *syn* can output an agent, or else output ± - think of ± as being like `nul1` in Java. Now, we will say a synthesis algorithm is

sound if, whenever it returns an agent, this agent succeeds in the task environment that is passed as input; and

complete if it is guaranteed to return an agent whenever there exists an agent that will succeed in the task environment given as input.

Thus a sound and complete synthesis algorithm will only output ± given input $\langle Env, \Psi\rangle$when no agent exists that will succeed in $\langle Env, \Psi\rangle$.

Formally, a synthesis algorithm *syn* is sound if it satisfies the following condition:

$$syn(\langle Env, \Psi\rangle) = Ag \text{ implies } \mathcal{R}(Ag, Env) = \mathcal{R}_{\Psi}(Ag, Env).$$

Similarly, *syn* is complete if it satisfies the following condition:

$$\exists Ag \in \mathcal{AG} \text{ s.t. } \mathcal{R}(Ag, Env) = \mathcal{R}_{\Psi}(Ag, Env) \text{ implies } syn(\langle Env, \Psi\rangle) \neq \perp.$$

Intuitively, soundness ensures that a synthesis algorithm always delivers agents that do their job correctly, but may not always deliver agents, even where such agents are in principle possible. Completeness ensures that an agent will always be delivered where such an agent is possible, but does not guarantee that these agents will do their job correctly. Ideally, we seek synthesis algorithms that are both sound *and* complete. Of the two conditions, soundness is probably the more important: there is not much point in complete synthesis algorithms that deliver 'buggy' agents.

Notes and Further Reading

A view of artificial intelligence as the process of agent design is presented in Russell and Norvig (1995), and, in particular, Chapter 2 of Russell and Norvig

(1995) presents much useful material. The definition of agents presented here is based on Wooldridge and Jennings (1995), which also contains an extensive review of agent architectures and programming languages. The question of 'what is an agent' is one that continues to generate some debate; a collection of answers may be found in Müller *et al.* (1997). The relationship between agents and objects has not been widely discussed in the literature, but see Gasser and Briot (1992). Other interesting and readable introductions to the idea of intelligent agents include Kaelbling (1986) and Etzioni (1993).

The abstract model of agents presented here is based on that given in Genesereth and Nilsson (1987, Chapter 13), and also makes use of some ideas from Russell and Wefald (1991) and Russell and Subramanian (1995). The properties of perception as discussed in this section lead to *knowledge theory,* a formal analysis of the information implicit within the slate of computer processes, which has had a profound effect in theoretical computer science: this issue is discussed in Chapter 12.

The relationship between artificially intelligent agents and software complexity has been discussed by several researchers: Simon (1981) was probably the first. More recently, Booch (1994) gives a good discussion of software complexity and the role that object-oriented development has to play in overcoming it. Russell and Norvig (1995) introduced the five-point classification of environments that we reviewed here, and distinguished between the 'easy' and 'hard' cases. Kaelbling (1986) touches on many of the issues discussed here, and Jennings (1999) also discusses the issues associated with complexity and agents.

The relationship between agent and environment, and, in particular, the problem of understanding how a given agent will perform in a given environment, has been studied empirically by several researchers. Pollack and Ringuette (1990) introduced the Tileworld, an environment for experimentally evaluating agents that allowed a user to experiment with various environmental parameters (such as the rate at which the environment changes - its *dynamism).* Building on this work, Kinny and Georgeff (1991) investigated how a specific class of agents, based on the *belief-desire-intention* model (Wooldridge, 2000b), could be tailored to perform well in environments with differing degrees of change and complexity. An attempt to prove some results corresponding to Kinny and Georgeff (1991) was Wooldridge and Parsons (1999); an experimental investigation of some of these relationships, building on Kinny and Georgeff (1991), was Schut and Wooldridge (2000). An informal discussion on the relationship between agent and environment is Miiller (1999).

In artificial intelligence, the planning problem is most closely related to achievement-based task environments (Allen *et al,* 1990). STRIPS was the archetypal planning system (Fikes and Nilsson, 1971). The STRIPS system is capable of taking a description of the initial environment state eo, a specification of the goal to be achieved, E_{good}, and the actions *Ac* available to an agent, and generates a sequence of actions $n \in Ac^*$ such that when executed from e_0, π will achieve one of the states E_{good}. The initial state, goal state, and actions were characterized in STRIPS using a subset of first-order logic. Bylander showed that the (propositional) STRIPS decision problem (given e_0, Ac, and E_{good} specified in propositional logic, does there exist a π G Ac^* such that π achieves E_{good}?) is PSPACE-complete (Bylander, 1994).

More recently, there has been renewed interest by the artificial intelligence planning community in *decision theoretic* approaches to planning (Blythe, 1999). One popular approach involves representing agents and their environments as 'partially observable Markov decision processes' (POMDPs) (Kaelbling *et al.*,1998). Put simply, the goal of solving a POMDP is to determine an optimal policy for acting in an environment in which there is uncertainty about the environment state (cf. our visibility function), and which is non-deterministic. Work on POMDP approaches to agent design are at an early stage, but show promise for the future.

The discussion on task specifications is adapted from Wooldridge (2000a) and Wooldridge a

Class reading: Franklin and Graesser (1997). This paper informally discusses various different notions of agency. The focus of the discussion might be on a comparison with the discussion in this chapter.