Contents

1 Intelligent Agents

Michael Wooldridge

1.1 Introduction

Computers are not very good at knowing what to do: every action a computer performs must be explicitly anticipated, planned for, and coded by a programmer. If a computer program ever encounters a situation that its designer did not anticipate, then the result is not usually pretty — a system crash at best, multiple loss of life at worst. This mundane fact is at the heart of our relationship with computers. It is so self-evident to the computer literate that it is rarely mentioned. And yet it comes as a complete surprise to those encountering computers for the first time.

For the most part, we are happy to accept computers as obedient, literal, unimaginative servants. For many applications (such as payroll processing), it is entirely acceptable. However, for an increasingly large number of applications, we require systems that can *decide for themselves* what they need to do in order to satisfy their design objectives. Such computer systems are known as *agents*. Agents that must operate robustly in rapidly changing, unpredictable, or open environments, where there is a significant possibility that actions can *fail* are known as *intelligent agents*, or sometimes *autonomous agents*. Here are examples of recent application areas for intelligent agents:

- When a space probe makes its long flight from Earth to the outer planets, a ground crew is usually required to continually track its progress, and decide how to deal with unexpected eventualities. This is costly and, if decisions are required *quickly*, it is simply not practicable. For these reasons, organisations like NASA are seriously investigating the possibility of making probes more autonomous — giving them richer decision making capability and responsibilities.
- Searching the Internet for the answer to a specific query can be a long and tedious process. So, why not allow a computer program — an agent — do searches for us? The agent would typically be given a query that would require synthesising pieces of information from various different Internet information sources. Failure would occur when a particular resource was unavailable, (perhaps due to network failure), or where results could not be obtained.

This chapter is about intelligent agents. Specifically, it aims to give you a thorough introduction to the main issues associated with the design and implementation of intelligent agents. After reading it, I hope that you will understand:

- why agents are perceived to be an important new way of conceptualising and implementing certain types of software application;
- what intelligent agents are (and are not), and how agents relate to other software paradigms — in particular, expert systems and object-oriented programming;
- the main approaches that have been advocated for designing and implementing intelligent agents, the issues surrounding these approaches, their relative merits, and the challenges that face the agent implementor;
- the characteristics of the main programming languages available for building agents today.

The chapter is structured as follows. First, in section 1.2, I describe what I mean by the term *agent*. In section 1.3, I present some *abstract architectures* for agents. That is, I discuss some general models and properties of agents without regard to how such agents might be implemented. In section 1.4, I discuss *concrete* architectures for agents. The various major design routes that one can follow in implementing an agent system are outlined in this section. In particular, *logic-based* architectures, *reactive* architectures, *belief-desire-intention* architectures, and finally, *layered* architectures for intelligent agents are described. Finally, section 1.5 introduces some prototypical programming languages for agent systems.

Comments on notation

This chapter makes use of simple mathematical notation in order to make ideas precise. The formalism used that of discrete maths: a basic grounding in sets and first-order logic should be quite sufficient to make sense of the various definitions presented. In addition: if S is an arbitrary set, then $\wp(S)$ is the powerset of S, and S^* is the set of sequences of elements of S; the symbol $\neg$ is used for logical negation (so $\neg p$ is read "not p"); $\wedge$ is used for conjunction (so $p \wedge q$ is read "p and q"); $\vee$ is used for disjunction (so $p \vee q$ is read "p or q"); and finally, $\Rightarrow$ is used for material implication (so $p \Rightarrow q$ is read "p implies q").

1.2 What are 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 multi-agent systems — surely we must all agree on what an agent is? Surprisingly, there is no such agreement: there is no universally accepted definition of the term agent, and indeed there is a good deal of 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

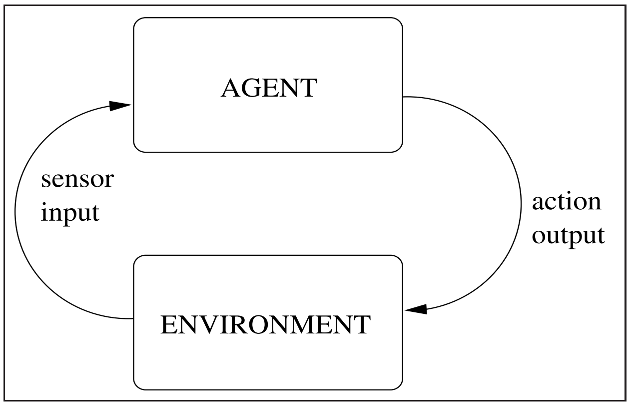

Figure 1.1 An agent in its environment. The agent takes sensory input from the environment, and produces as output actions that affect it. The interaction is usually an ongoing, non-terminating one.

their experiences is of paramount importance; for other applications, learning is not only unimportant, it is undesirable.

Nevertheless, some sort of definition is important — otherwise, there is a danger that the term will lose all meaning (cf. "user friendly"). The definition presented here is adapted from [71]: 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.

There are several points to note about this definition. First, the definition refers to "agents" and not "intelligent agents". The distinction is deliberate: it is discussed in more detail below. Second, the definition does not say anything about what *type* of environment an agent occupies. Again, this is deliberate: agents can occupy many different types of environment, as we shall see below. Third, we have not defined *autonomy*. Like agency itself, autonomy is a somewhat tricky concept to tie down precisely, but I mean it in the sense that agents are able to act without the intervention of humans or other systems: they have control both over their own internal state, and over their behaviour. In section 1.2.3, we will contrast agents with the objects of object-oriented programming, and we will elaborate this point there. In particular, we will see how agents embody a much stronger sense of autonomy than do objects.

Figure 1.1 gives an abstract, top-level 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 *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 area available to do so. Actions therefore have *pre-conditions* 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 several examples later in this article, are really software architectures for decision making systems that are embedded in an environment. The complexity of the decision-making process can be affected by a number of different environmental properties. Russell and Norvig suggest the following classification of environment properties [59, p46]:

- *Accessible* vs *inaccessible*.
  An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state. Most moderately complex environments (including, for example, the everyday physical world and the Internet) are inaccessible. The more accessible an environment is, the simpler it is to build agents to operate in it.
- *Deterministic* vs *non-deterministic*.
  As we have already mentioned, 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. The physical world can to all intents and purposes be regarded as non-deterministic. Non-deterministic environments present greater problems for the agent designer.
- *Episodic* vs *non-episodic*.
  In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of an agent in different scenarios. An example of an episodic environment would be a mail sorting system [60]. Episodic environments are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current episode — it need not reason about the interactions between this and future episodes.
- *Static* vs *dynamic*.
  A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agent. 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.
- *Discrete* vs *continuous*.
  An environment is discrete if there are a fixed, finite number of actions and percepts in it. Russell and Norvig give a chess game as an example of a discrete environment, and taxi driving as an example of a continuous one.

As Russell and Norvig observe [59, p46], 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. The most complex general class of environments are those that are inaccessible, non-deterministic, non-episodic, dynamic, and continuous.

1.2.1 Examples of Agents

At this point, it is worth pausing to consider some examples of agents (though not, as yet, intelligent agents):

- 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:

$$\begin{array}{rcl} \text{too cold} & \longrightarrow & \text{heating on} \\ \text{temperature OK} & \longrightarrow & \text{heating off} \end{array}$$

  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.

- Most software daemons, (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 summarise, 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.

1.2.2 Intelligent Agents

We are not used to thinking of thermostats or UNIX daemons as agents, and certainly not as *intelligent* agents. So, when do we consider an agent to be intelligent? The question, like the question *what is intelligence?* itself, is not an easy one to answer. But for me, an intelligent agent is one that is capable of *flexible* autonomous action in order to meet its design objectives, where by flexible, I mean three things [71]:

- *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;
- *pro-activeness*: 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 are more demanding than they might at first appear. To see why, let us consider them in turn. First, consider *pro-activeness*: 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 *pre-condition*) and the *effect* it has if the assumptions are valid (its *post-condition*). The effects of the procedure are its *goal*: what the author of the software intends the procedure to achieve. If the pre-condition holds when the procedure is invoked, then we expect that the procedure will execute *correctly*: that it will terminate, and that upon termination, the post-condition 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, its works well when we consider *functional systems* — those that simply take some input x, and produce as output some some function $f(x)$ of this input. Compilers are a classic example of functional systems.

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 (pre-condition) 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 *multi-agent* (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 in this chapter, 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 don't 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 focussing 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 who do this very well. How many of us have had a manager who stayed blindly focussed on some project long after the relevance of the project was passed, or it was clear that the project plan was doomed to failure? Similarly, how many have encountered managers who seem unable to stay focussed at all, who flit from one project to another without ever managing to pursue a goal long enough to achieve *anything*? 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.

1.2.3 Agents and Objects

Object-oriented programmers often fail to see anything novel or new in the idea of agents. When one stops to consider the relative properties of agents and objects, this is perhaps not surprising. 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 are autonomous. 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 it's *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 *must* 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 multi-agent system, (in particular, those that contain agents built by different organisations or individuals), no such common goal can be assumed. It cannot be 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. I have heard this distinction between objects and agents nicely summarised in the following slogan: *Objects do it for free; agents do it for money*.

Note that 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, pro-active, 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 we have 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. Passive objects, on the other hand, can only undergo a state change when explicitly acted upon. [5, p91]

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

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

- agents embody 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, pro-active, social) behaviour, and the standard object model has nothing to say about such types of behaviour;
- a multi-agent system is inherently multi-threaded, in that each agent is assumed to have at least one thread of control.

1.2.4 Agents and Expert Systems

Expert systems were the most important AI technology of the 1980s [31]. An expert system is one that is capable of solving problems or giving advice in some knowledge-rich domain [32]. 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, we 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, we do not generally require expert systems to be capable of co-operating with other agents. 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 [33].

Sources and Further Reading

A view of artificial intelligence as the process of agent design is presented in [59], and in particular, Chapter 2 of [59] presents much useful material. The definition of agents presented here is based on [71], which also contains an extensive review of agent architectures and programming languages. In addition, [71] contains a detailed survey of *agent theories* — formalisms for reasoning about intelligent, rational agents, which is outside the scope of this chapter. This question of "what is an agent" is one that continues to generate some debate; a collection of answers may be found in [48]. The relationship between agents and objects has not been widely discussed in the literature, but see [24]. Other readable introductions to the idea of intelligent agents include [34] and [13].

1.3 Abstract Architectures for Intelligent Agents

We can easily formalise the abstract view of agents presented so far. First, we will assume that the state of the agent's environment can be characterised as a set $S = \{s_1, s_2, \ldots\}$ of *environment states*. At any given instant, the environment is assumed to be in one of these states. The effectoric capability of an agent is assumed to be represented by a set $A = \{a_1, a_2, \ldots\}$ of *actions*. Then abstractly, an agent can be viewed as a function

$$action : S^* \to A$$

which maps sequences of environment states to actions. We will refer to an agent modelled by a function of this form as a *standard agent*. The intuition is that an agent decides what action to perform on the basis of its history — its experiences to date. These experiences are represented as a sequence of environment states — those that the agent has thus far encountered.

The (non-deterministic) behaviour of an an environment can be modelled as a function

$$env : S \times A \to \wp(S)$$

which takes the current state of the environment $s \in S$ and an action $a \in A$ (performed by the agent), and maps them to a set of environment states $env(s, a)$ — those that could result from performing action a in state s. If all the sets in the range of *env* are all singletons, (i.e., if the result of performing any action in any state is a set containing a single member), then the environment is *deterministic*, and its behaviour can be accurately predicted.

We can represent the interaction of agent and environment as a *history*. A history h is a sequence:

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \cdots$$

where s_0 is the initial state of the environment (i.e., its state when the agent starts executing), a_u is the u'th action that the agent chose to perform, and s_u is the u'th environment state (which is one of the possible results of executing action a_{u-1} in state s_{u-1}). If $action : S^* \to A$ is an agent, $env : S \times A \to \wp(S)$ is an environment, and s_0 is the initial state of the environment, then the sequence

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \cdots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \cdots$$

will represent a possible history of the agent in the environment iff the following two conditions hold:

$$\forall u \in \mathbb{N}, a_u = action((s_0, s_1, \ldots, s_u))$$

and

$$\forall u \in \mathbb{N} \text{ such that } u > 0, s_u \in env(s_{u-1}, a_{u-1}).$$

The *characteristic behaviour* of an agent $action : S^* \to A$ in an environment $env : S \times A \to \wp(S)$ is the set of all the histories that satisfy these properties. If some property ϕ holds of all these histories, this this property can be regarded as an invariant property of the agent in the environment. For example, if our agent is a nuclear reactor controller, (i.e., the environment is a nuclear reactor), and in all possible histories of the controller/reactor, the reactor does not blow up, then this can be regarded as a (desirable) invariant property. We will denote by $hist(agent, environment)$ the set of all histories of *agent* in *environment*. Two agents ag_1 and ag_2 are said to be *behaviourally equivalent* with respect to environment *env* iff $hist(ag_1, env) = hist(ag_2, env)$, and simply behaviourally equivalent iff they are behaviourally equivalent with respect to all environments.

In general, we are interested in agents whose interaction with their environment *does not end*, i.e., they are *non-terminating*. In such cases, the histories that we consider will be infinite.

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

$$action : S \to A.$$

It should be easy to see that for every purely reactive agent, there is an equivalent standard agent; 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's action function is simply

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

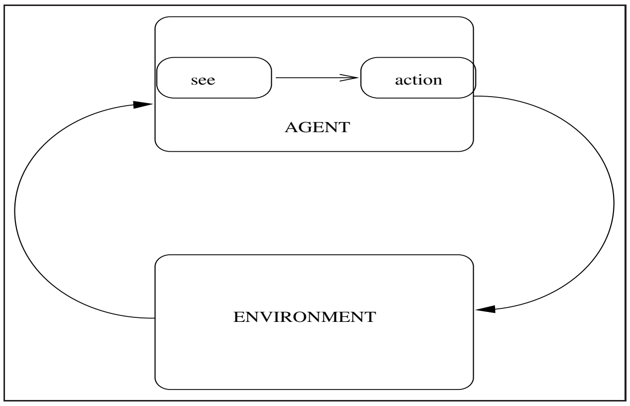

Figure 1.2 Perception and action subsystems.

1.3.2 Perception

Viewing agents at this abstract level makes for a pleasantly simply analysis. However, it does not help us to construct them, since it gives us no clues about how to design the decision function *action*. 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 chapter, 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. In 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 1.2.

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 infra-red 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 P be a (non-empty) set of percepts. Then *see* is a function

$$see : S \to P$$

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

$$action : P^* \to A$$

which maps sequences of percepts to actions.

These simple definitions allow us to explore some interesting properties of agents and perception. Suppose that we have two environment states, $s_1 \in S$ and $s_2 \in S$, such that $s_1 \neq s_2$, but $see(s_1) = see(s_2)$. 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, s_1 and s_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 S of environment states contains exactly four elements:

$$S = \{\underbrace{\{\neg x, \neg y\}}_{s_1}, \underbrace{\{\neg x, y\}}_{s_2}, \underbrace{\{x, \neg y\}}_{s_3}, \underbrace{\{x, y\}}_{s_4}\}$$

Thus in state s_1, the room temperature is not OK, and John Major is not Prime Minister; in state s_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:

$$see(s) = \begin{cases} p_1 & \text{if } s = s_1 \text{ or } s = s_2 \\ p_2 & \text{if } s = s_3 \text{ or } s = s_4. \end{cases}$$

Given two environment states $s \in S$ and $s' \in S$, let us write $s \equiv s'$ if $see(s) = see(s')$. It is not hard to see that $\equiv$ is an *equivalence relation* over environment states, which partitions S into mutually indistinguishable sets of states. Intuitively, the coarser these equivalence classes are, the less effective is the agent's perception. If $| \equiv | = |S|$, (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 $| \equiv | = 1$, 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.

1.3.3 Agents with state

We have so far been modelling an agent's decision function *action* 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,

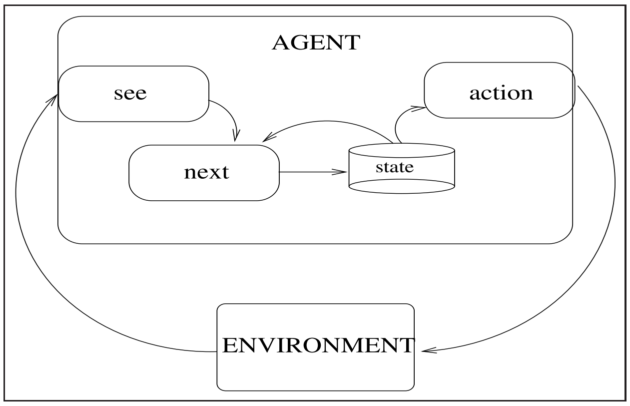

Figure 1.3 Agents that maintain state.

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 1.3.

These agents have some internal data structure, which is typically used to record information about the environment state and history. Let I 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:

$$see : S \to P$$

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

$$action : I \to A$$

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

$$next : I \times P \to I$$

The behaviour of a state-based agent can be summarised as follows. The agent starts in some initial internal state i_0. It then observes its environment state s, and generates a percept $see(s)$. The internal state of the agent is then updated via the *next* function, becoming set to $next(i_0, see(s))$. The action selected by the agent is then $action(next(i_0, see(s)))$. 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.

Sources and Further Reading

The abstract model of agents presented here is based on that given in [25, Chapter 13], and also makes use of some ideas from [61, 60]. The properties of perception as discussed in this section lead to *knowledge theory*, a formal analysis of the information implicit within the state of computer processes, which has had a profound effect in theoretical computer science. The definitive reference is [14], and an introductory survey is [29].

1.4 Concrete Architectures for Intelligent Agents

Thus far, we have considered agents only in the abstract. So while we have examined the properties of agents that do and do not maintain state, we have not stopped to consider what this state might look like. Similarly, we have modelled an agent's decision making as an abstract function *action*, which somehow manages to indicate which action to perform — but we have not discussed how this function might be implemented. In this section, we will rectify this omission. We will consider four classes of agents:

- *logic based agents* — in which decision making is realised through logical deduction;
- *reactive agents* — in which decision making is implemented in some form of direct mapping from situation to action;
- *belief-desire-intention agents* — in which decision making depends upon the manipulation of data structures representing the beliefs, desires, and intentions of the agent; and finally,
- *layered architectures* — in which decision making is realised via various software layers, each of which is more-or-less explicitly reasoning about the environment at different levels of abstraction.

In each of these cases, we are moving away from the abstract view of agents, and beginning to make quite specific commitments about the internal structure and operation of agents. In each section, I will try to explain the nature of these commitments, the assumptions upon which the architectures depend, and the relative advantages and disadvantages of each.

1.4.1 Logic-based Architectures

The "traditional" approach to building artificially intelligent systems, (known as *symbolic AI*) suggests that intelligent behaviour can be generated in a system by giving that system a *symbolic* representation of its environment and its desired behaviour, and syntactically manipulating this representation. In this section, we focus on the apotheosis of this tradition, in which these symbolic representations are *logical formulae*, and the syntactic manipulation corresponds to *logical deduction*, or *theorem proving*.

The idea of agents as theorem provers is seductive. Suppose we have some theory of agency — some theory that explains how an intelligent agent should behave. This theory might explain, for example, how an agent generates goals so as to satisfy its design objective, how it interleaves goal-directed and reactive behaviour in order to achieve these goals, and so on. Then this theory ϕ can be considered as a *specification* for how an agent should behave. The traditional approach to implementing a system that will satisfy this specification would involve *refining* the specification through a series of progressively more concrete stages, until finally an implementation was reached. In the view of agents as theorem provers, however, no such refinement takes place. Instead, ϕ is viewed as an *executable specification*: it is *directly executed* in order to produce the agent's behaviour.

To see how such an idea might work, we shall develop a simple model of logic-based agents, which we shall call *deliberate* agents. In such agents, the internal state is assumed to be a database of formulae of classical first-order predicate logic. For example, the agent's database might contain formulae such as:

$$Open(valve221)$$
$$Temperature(reactor4726, 321)$$
$$Pressure(tank776, 28)$$

It is not difficult to see how formulae such as these can be used to represent the properties of some environment. The database is the *information* that the agent has about its environment. An agent's database plays a somewhat analogous role to that of *belief* in humans. Thus a person might have a belief that valve 221 is open — the agent might have the predicate $Open(valve221)$ in its database. Of course, just like humans, agents can be wrong. Thus I might believe that valve 221 is open when it is in fact closed; the fact that an agent has $Open(valve221)$ in its database does not mean that valve 221 (or indeed any valve) is open. The agent's sensors may be faulty, its reasoning may be faulty, the information may be out of date, or the interpretation of the formula $Open(valve221)$ intended by the agent's designer may be something entirely different.

Let L be the set of sentences of classical first-order logic, and let $D = \wp(L)$ be the set of L *databases*, i.e., the set of sets of L-formulae. The internal state of an agent is then an element of D. We write $\Delta, \Delta_1, \ldots$ for members of D. The internal state of an agent is then simply a member of the set D. An agent's decision making process is modelled through a set of *deduction rules*, ρ. These are simply rules of inference for the logic. We write $\Delta \vdash_\rho \phi$ if the formula ϕ can be proved from the database Δ using only the deduction rules ρ. An agents perception function *see* remains unchanged:

$$see : S \to P.$$

Similarly, our *next* function has the form

$$next : D \times P \to D$$

It thus maps a database and a percept to a new database. However, an agent's action selection function, which has the signature

$$action : D \to A$$

is defined in terms of its deduction rules. The pseudo-code definition of this function is as follows.

```
1.   function action(Δ : D) : A
2.   begin
3.       for each a ∈ A do
4.           if Δ ⊢ρ Do(a) then
5.               return a
6.           end-if
7.       end-for
8.       for each a ∈ A do
9.           if Δ ⊬ρ ¬Do(a) then
10.              return a
11.          end-if
12.      end-for
13.      return null
14.  end function action
```

The idea is that the agent programmer will encode the deduction rules ρ and database Δ in such a way that if a formula $Do(a)$ can be derived, where a is a term that denotes an action, then a is the best action to perform. Thus, in the first part of the function (lines (3)–(7)), the agent takes each of its possible actions a in turn, and attempts to prove the form the formula $Do(a)$ from its database (passed as a parameter to the function) using its deduction rules ρ. If the agent succeeds in proving $Do(a)$, then a is returned as the action to be performed.

What happens if the agent fails to prove $Do(a)$, for all actions $a \in A$? In this case, it attempts to find an action that is *consistent* with the rules and database, i.e., one that is not explicitly forbidden. In lines (8)–(12), therefore, the agent attempts to find an action $a \in A$ such that $\neg Do(a)$ cannot be derived from its database using its deduction rules. If it can find such an action, then this is returned as the action to be performed. If, however, the agent fails to find an action that is at least consistent, then it returns a special action *null* (or *noop*), indicating that no action has been selected.

In this way, the agent's behaviour is determined by the agent's deduction rules (its "program") and its current database (representing the information the agent has about its environment).

To illustrate these ideas, let us consider a small example (based on the vacuum cleaning world example of [59, p51]). The idea is that we have a small robotic agent that will clean up a house. The robot is equipped with a sensor that will tell it whether it is over any dirt, and a vacuum cleaner that can be used to suck up dirt. In addition, the robot always has a definite orientation (one of *north*, *south*, *east*, or *west*). In addition to being able to suck up dirt, the agent can move forward one "step" or turn right 90°. The agent moves around a room, which is divided grid-like into a number of equally sized squares (conveniently corresponding to the unit of movement of the agent). We will assume that our agent does nothing but clean — it never leaves the room, and further, we will assume in the interests of simplicity that the room is a 3×3 grid, and the agent always starts in grid square $(0,0)$ facing north.

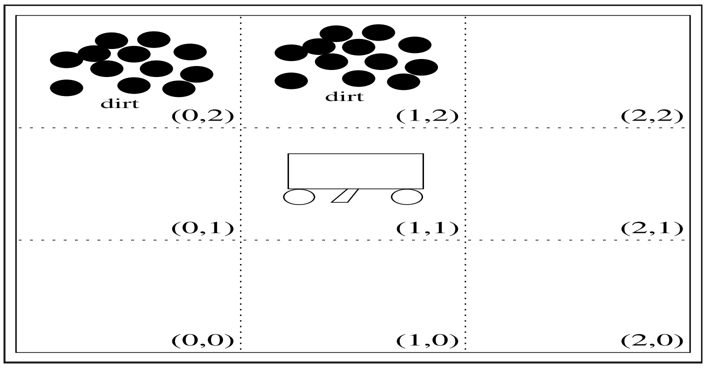

Figure 1.4 Vacuum world

To summarise, our agent can receive a percept *dirt* (signifying that there is dirt beneath it), or *null* (indicating no special information). It can perform any one of three possible actions: $forward$, *suck*, or $turn$. The goal is to traverse the room continually searching for and removing dirt. See Figure 1.4 for an illustration of the vacuum world.

First, note that we make use of three simple *domain predicates* in this exercise:

$In(x,y)$	agent is at (x,y)
$Dirt(x,y)$	there is dirt at (x,y)
$Facing(d)$	the agent is facing direction d

Now we specify our *next* function. This function must look at the perceptual information obtained from the environment (either *dirt* or *null*), and generate a new database which includes this information. But in addition, it must *remove* old or irrelevant information, and also, it must try to figure out the new location and orientation of the agent. We will therefore specify the *next* function in several parts. First, let us write $old(\Delta)$ to denote the set of "old" information in a database, which we want the update function *next* to remove:

$$old(\Delta) = \{P(t_1, \ldots, t_n) \mid P \in \{In, Dirt, Facing\} \text{ and } P(t_1, \ldots, t_n) \in \Delta\}$$

Next, we require a function *new*, which gives the set of new predicates to add to the database. This function has the signature

$$new : D \times P \to D$$

The definition of this function is not difficult, but it is rather lengthy, and so we will leave it as an exercise. (It must generate the predicates $In(\ldots)$, describing the new position of the agent, $Facing(\ldots)$ describing the orientation of the agent, and $Dirt(\ldots)$ if dirt has been detected at the new position.) Given the *new* and *old* functions, the *next* function is defined as follows:

$$next(\Delta, p) = (\Delta \setminus old(\Delta)) \cup new(\Delta, p)$$

Now we can move on to the rules that govern our agent's behaviour. The deduction rules have the form

$\phi(\ldots) \longrightarrow \psi(\ldots)$

where ϕ and ψ are predicates over some arbitrary list of constants and variables. The idea being that if ϕ matches against the agent's database, then ψ can be concluded, with any variables in ψ instantiated.

The first rule deals with the basic cleaning action of the agent: this rule will take priority over all other possible behaviours of the agent (such as navigation).

$$In(x,y) \wedge Dirt(x,y) \longrightarrow Do(suck) \tag{1.1}$$

Hence if the agent is at location (x,y) and it perceives dirt, then the prescribed action will be to suck up dirt. Otherwise, the basic action of the agent will be to traverse the world. Taking advantage of the simplicity of our environment, we will hardwire the basic navigation algorithm, so that the robot will always move from $(0,0)$ to $(0,1)$ to $(0,2)$ and then to $(1,2)$, $(1,1)$ and so on. Once the agent reaches $(2,2)$, it must head back to $(0,0)$. The rules dealing with the traversal up to $(0,2)$ are very simple.

$$In(0,0) \wedge Facing(north) \wedge \neg Dirt(0,0) \longrightarrow Do(forward) \tag{1.2}$$
$$In(0,1) \wedge Facing(north) \wedge \neg Dirt(0,1) \longrightarrow Do(forward) \tag{1.3}$$
$$In(0,2) \wedge Facing(north) \wedge \neg Dirt(0,2) \longrightarrow Do(turn) \tag{1.4}$$
$$In(0,2) \wedge Facing(east) \longrightarrow Do(forward) \tag{1.5}$$

Notice that in each rule, we must explicitly check whether the antecedent of rule (1.1) fires. This is to ensure that we only ever prescribe one action via the $Do(\ldots)$ predicate. Similar rules can easily be generated that will get the agent to $(2,2)$, and once at $(2,2)$ back to $(0,0)$. It is not difficult to see that these rules, together with the *next* function, will generate the required behaviour of our agent.

At this point, it is worth stepping back and examining the pragmatics of the logic-based approach to building agents. Probably the most important point to make is that a literal, naive attempt to build agents in this way would be more or less entirely impractical. To see why, suppose we have designed out agent's rule set ρ such that for any database Δ, if we can prove $Do(a)$ then a is an *optimal* action — that is, a is the best action that could be performed when the environment is as described in Δ. Then imagine we start running our agent. At time t_1, the agent has generated some database Δ_1, and begins to apply its rules ρ in order to find which action to perform. Some time later, at time t_2, it manages to establish $\Delta_1 \vdash_\rho Do(a)$ for some $a \in A$, and so a is the optimal action that the agent could perform at time t_1. But if the environment has *changed* between t_1 and t_2, then there is no guarantee that a will *still* be optimal. It could be far from optimal, particularly if much time has elapsed between t_1 and t_2. If $t_2 - t_1$ is infinitesimal — that is, if decision making is effectively instantaneous — then we could safely disregard this problem. But in fact, we know that reasoning of the kind our logic-based agents use will be anything *but* instantaneous. (If our agent uses classical first-order predicate logic to represent the environment, and its rules are sound and complete, then there is no guarantee that the decision making procedure will even *terminate*.) An agent is said to enjoy the property of *calculative rationality* if and only if its decision making apparatus will suggest an action that was optimal *when the decision making process began*. Calculative rationality is clearly not acceptable in environments that change faster than the agent can make decisions — we shall return to this point later.

One might argue that this problem is an artifact of the pure logic-based approach adopted here. There is an element of truth in this. By moving away from strictly logical representation languages and complete sets of deduction rules, one can build agents that enjoy respectable performance. But one also loses what is arguably the greatest advantage that the logical approach brings: a simple, elegant logical semantics.

There are several other problems associated with the logical approach to agency. First, the *see* function of an agent, (its perception component), maps its environment to a percept. In the case of a logic-based agent, this percept is likely to be symbolic — typically, a set of formulae in the agent's representation language. But for many environments, it is not obvious how the mapping from environment to symbolic percept might be realised. For example, the problem of transforming an image to a set of declarative statements representing that image has been the object of study in AI for decades, and is still essentially open. Another problem is that actually *representing* properties of dynamic, real-world environments is extremely hard. As an example, representing and reasoning about *temporal information* — how a situation changes over time — turns out to be extraordinarily difficult. Finally, as the simple vacuum world example illustrates, representing even rather simple *procedural* knowledge (i.e., knowledge about "what to do") in traditional logic can be rather unintuitive and cumbersome.

To summarise, in logic-based approaches to building agents, decision making is viewed as deduction. An agent's "program" — that is, its decision making strategy — is encoded as a logical theory, and the process of selecting an action reduces to a problem of proof. Logic-based approaches are elegant, and have a clean (logical) semantics — wherein lies much of their long-lived appeal. But logic-based approaches have many disadvantages. In particular, the inherent computational complexity of theorem proving makes it questionable whether agents as theorem provers can operate effectively in time-constrained environments. Decision making in such agents is predicated on the assumption of calculative rationality — the assumption that the world will not change in any significant way while the agent is deciding what to do, and that an action which is rational when decision making begins will be rational when it concludes. The problems associated with representing and reasoning about complex, dynamic, possibly physical environments are also essentially unsolved.

Sources and Further Reading

My presentation of logic based agents is based largely on the discussion of *deliberate agents* presented in [25, Chapter 13], which represents the logic-centric view of AI and agents very well. The discussion is also partly based on [38]. A number of more-or-less "pure" logical approaches to agent programming have been developed. Well-known examples include the CONGOLOG system of Lespérance and colleagues [39] (which is based on the *situation calculus* [45]) and the METATEM and Concurrent METATEM programming languages developed by Fisher and colleagues [3, 21] (in which agents are programmed by giving them *temporal logic* specifications of the behaviour they should exhibit). Concurrent METATEM is discussed as a case study in section 1.5. Note that these architectures (and the discussion above) assume that if one adopts a logical approach to agent-building, then this means agents are essentially theorem provers, employing explicit symbolic reasoning (theorem proving) in order to make decisions. But just because we find logic a useful tool for conceptualising or specifying agents, this does not mean that we must view decision-making as logical manipulation. An alternative is to *compile* the logical specification of an agent into a form more amenable to efficient decision making. The difference is rather like the distinction between interpreted and compiled programming languages. The best-known example of this work is the *situated automata* paradigm of Leslie Kaelbling and Stanley Rosenschein [58]. A review of the role of logic in intelligent agents may be found in [70]. Finally, for a detailed discussion of calculative rationality and the way that it has affected thinking in AI, see [60].

1.4.2 Reactive Architectures

The seemingly intractable problems with symbolic/logical approaches to building agents led some researchers to question, and ultimately reject, the assumptions upon which such approaches are based. These researchers have argued that minor changes to the symbolic approach, such as weakening the logical representation language, will not be sufficient to build agents that can operate in time-constrained environments: nothing less than a whole new approach is required. In the mid-to-late 1980s, these researchers began to investigate alternatives to the symbolic AI paradigm. It is difficult to neatly characterise these different approaches, since their advocates are united mainly by a rejection of symbolic AI, rather than by a common manifesto. However, certain themes do recur:

- the rejection of symbolic representations, and of decision making based on syntactic manipulation of such representations;
- the idea that intelligent, rational behaviour is seen as innately linked to the *environment* an agent occupies — intelligent behaviour is not disembodied, but is a product of the *interaction* the agent maintains with its environment;
- the idea that intelligent behaviour *emerges* from the interaction of various simpler behaviours.

Alternative approaches to agency are sometime referred to as *behavioural* (since a common theme is that of developing and combining individual behaviours), *situated* (since a common theme is that of agents actually situated in some environment, rather than being disembodied from it), and finally — the term I will use — *reactive* (because such systems are often perceived as simply reacting to an environment, without reasoning about it). This section presents a survey of the *subsumption architecture*, which is arguably the best-known reactive agent architecture. It was developed by Rodney Brooks — one of the most vocal and influential critics of the symbolic approach to agency to have emerged in recent years.

There are two defining characteristics of the subsumption architecture. The first is that an agent's decision-making is realised through a set of *task accomplishing behaviours*; each behaviour may be though of as an individual *action* function, as we defined above, which continually takes perceptual input and maps it to an action to perform. Each of these behaviour modules is intended to achieve some particular task. In Brooks' implementation, the behaviour modules are finite state machines. An important point to note is that these task accomplishing modules are assumed to include *no* complex symbolic representations, and are assumed to do *no* symbolic reasoning at all. In many implementations, these behaviours are implemented as rules of the form

situation $\longrightarrow$ action

which simple map perceptual input directly to actions.

The second defining characteristic of the subsumption architecture is that many behaviours can "fire" simultaneously. There must obviously be a mechanism to choose between the different actions selected by these multiple actions. Brooks proposed arranging the modules into a *subsumption hierarchy*, with the behaviours arranged into *layers*. Lower layers in the hierarchy are able to *inhibit* higher layers: the lower a layer is, the higher is its priority. The idea is that higher layers represent more abstract behaviours. For example, one might desire a behaviour in a mobile robot for the behaviour "avoid obstacles". It makes sense to give obstacle avoidance a high priority — hence this behaviour will typically be encoded in a *low-level* layer, which has *high* priority. To illustrate the subsumption architecture in more detail, we will now present a simple formal model of it, and illustrate how it works by means of a short example. We then discuss its relative advantages and shortcomings, and point at other similar reactive architectures.

The *see* function, which represents the agent's perceptual ability, is assumed to remain unchanged. However, in implemented subsumption architecture systems, there is assumed to be quite tight coupling between perception and action — raw sensor input is not processed or transformed much, and there is certainly no attempt to transform images to symbolic representations.

The decision function *action* is realised through a set of behaviours, together with an *inhibition* relation holding between these behaviours. A behaviour is a pair (c, a), where $c \subseteq P$ is a set of percepts called the *condition*, and $a \in A$ is an action. A behaviour (c, a) will *fire* when the environment is in state $s \in S$ iff $see(s) \in c$. Let $Beh = \{(c, a) \mid c \subseteq P \text{ and } a \in A\}$ be the set of all such rules.

Associated with an agent's set of behaviour rules $R \subseteq Beh$ is a binary *inhibition relation* on the set of behaviours: $\prec \subseteq R \times R$. This relation is assumed to be a total ordering on R (i.e., it is transitive, irreflexive, and antisymmetric). We write $b_1 \prec b_2$ if $(b_1, b_2) \in \prec$, and read this as "b_1 inhibits b_2", that is, b_1 is lower in the hierarchy than b_2, and will hence get priority over b_2. The action function is then defined as follows:

1. function $action(p : P) : A$
2. var $fired : \wp(R)$

3. var $selected : A$
4. begin
5. $fired := \{(c,a) \mid (c,a) \in R$ and $p \in c\}$
6. for each $(c,a) \in fired$ do
7. if $\neg(\exists(c',a') \in fired$ such that $(c',a') \prec (c,a))$ then
8. return a
9. end-if
10. end-for
11. return $null$
12. end function $action$

Thus action selection begins by first computing the set $fired$ of all behaviours that fire (5). Then, each behaviour (c,a) that fires is checked, to determine whether there is some other higher priority behaviour that fires. If not, then the action part of the behaviour, a, is returned as the selected action (8). If no behaviour fires, then the distinguished action *null* will be returned, indicating that no action has been chosen.

Given that one of our main concerns with logic-based decision making was its theoretical complexity, it is worth pausing to examine how well our simple behaviour-based system performs. The overall time complexity of the subsumption action function is no worse than $O(n^2)$, where n is the larger of the number of behaviours or number of percepts. Thus, even with the naive algorithm above, decision making is tractable. In practice, we can do *considerably* better than this: the decision making logic can be encoded into hardware, giving *constant* decision time. For modern hardware, this means that an agent can be guaranteed to select an action within nano-seconds. Perhaps more than anything else, this computational simplicity is the strength of the subsumption architecture.

To illustrate how the subsumption architecture in more detail, we will show how subsumption architecture agents were built for the following scenario (this example is adapted from [66]):

The objective is to explore a distant planet, more concretely, to collect samples of a particular type of precious rock. The location of the rock samples is not known in advance, but they are typically clustered in certain spots. A number of autonomous vehicles are available that can drive around the planet collecting samples and later reenter the a mothership spacecraft to go back to earth. There is no detailed map of the planet available, although it is known that the terrain is full of obstacles — hills, valleys, etc. — which prevent the vehicles from exchanging any communication.

The problem we are faced with is that of building an agent control architecture for each vehicle, so that they will cooperate to collect rock samples from the planet surface as efficiently as possible. Luc Steels argues that logic-based agents, of the type we described above, are "entirely unrealistic" for this problem [66]. Instead, he proposes a solution using the subsumption architecture.

The solution makes use of two mechanisms introduced by Steels: The first is a *gradient field*. In order that agents can know in which direction the mothership lies, the mothership generates a radio signal. Now this signal will obviously weaken as distance to the source increases — to find the direction of the mothership, an agent need therefore only travel "up the gradient" of signal strength. The signal need not carry any information — it need only exist.

The second mechanism enables agents to communicate with one another. The characteristics of the terrain prevent direct communication (such as message passing), so Steels adopted an *indirect* communication method. The idea is that agents will carry "radioactive crumbs", which can be dropped, picked up, and detected by passing robots. Thus if an agent drops some of these crumbs in a particular location, then later, another agent happening upon this location will be able to detect them. This simple mechanism enables a quite sophisticated form of cooperation.

The behaviour of an individual agent is then built up from a number of behaviours, as we indicated above. First, we will see how agents can be programmed to *individually* collect samples. We will then see how agents can be programmed to generate a *cooperative* solution.

For individual (non-cooperative) agents, the lowest-level behaviour, (and hence the behaviour with the highest "priority") is obstacle avoidance. This behaviour can can be represented in the rule:

if detect an obstacle *then* change direction. (1.6)

The second behaviour ensures that any samples carried by agents are dropped back at the mother-ship.

if carrying samples *and* at the base *then* drop samples (1.7)

if carrying samples and *not* at the base *then* travel up gradient. (1.8)

Behaviour (1.8) ensures that agents carrying samples will return to the mother-ship (by heading towards the origin of the gradient field). The next behaviour ensures that agents will collect samples they find.

if detect a sample *then* pick sample up. (1.9)

The final behaviour ensures that an agent with "nothing better to do" will explore randomly.

if true *then* move randomly. (1.10)

The pre-condition of this rule is thus assumed to always fire. These behaviours are arranged into the following hierarchy:

$$(1.6) \prec (1.7) \prec (1.8) \prec (1.9) \prec (1.10)$$

The subsumption hierarchy for this example ensures that, for example, an agent will *always* turn if any obstacles are detected; if the agent is at the mother-ship and is carrying samples, then it will *always* drop them if it is not in any immediate danger of crashing, and so on. The "top level" behaviour — a random walk — will only every be carried out if the agent has nothing more urgent to do. It is not difficult to see how this simple set of behaviours will solve the problem: agents will search for samples (ultimately by searching randomly), and when they find them, will return them to the mother-ship.

If the samples are distributed across the terrain entirely at random, then equipping a large number of robots with these very simple behaviours will work extremely well. But we know from the problem specification, above, that this is not the case: the samples tend to be located in clusters. In this case, it makes sense to have agents *cooperate* with one-another in order to find the samples. Thus when one agent finds a large sample, it would be helpful for it to communicate this to the other agents, so they can help it collect the rocks. Unfortunately, we also know from the problem specification that *direct* communication is impossible. Steels developed a simple solution to this problem, partly inspired by the foraging behaviour of ants. The idea revolves around an agent creating a "trail" of radioactive crumbs whenever it finds a rock sample. The trail will be created when the agent returns the rock samples to the mother ship. If at some later point, another agent comes across this trail, then it need only follow it down the gradient field to locate the source of the rock samples. Some small refinements improve the efficiency of this ingenious scheme still further. First, as an agent follows a trail to the rock sample source, it picks up some of the crumbs it finds, hence making the trail fainter. Secondly, the trail is *only* laid by agents returning to the mothership. Hence if an agent follows the trail out to the source of the nominal rock sample only to find that it contains no samples, it will reduce the trail on the way out, and will not return with samples to reinforce it. After a few agents have followed the trail to find no sample at the end of it, the trail will in fact have been removed.

The modified behaviours for this example are as follows. Obstacle avoidance, (1.6), remains unchanged. However, the two rules determining what to do if carrying a sample are modified as follows.

if carrying samples *and* at the base *then* drop samples (1.11)

if carrying samples and *not* at the base
then drop 2 crumbs *and* travel up gradient. (1.12)

The behaviour (1.12) requires an agent to drop crumbs when returning to base with a sample, thus either reinforcing or creating a trail. The "pick up sample" behaviour, (1.9), remains unchanged. However, an additional behaviour is required for dealing with crumbs.

if sense crumbs *then* pick up 1 crumb *and* travel down gradient (1.13)

Finally, the random movement behaviour, (1.10), remains unchanged. These behaviour are then arranged into the following subsumption hierarchy.

$$(1.6) \prec (1.11) \prec (1.12) \prec (1.9) \prec (1.13) \prec (1.10)$$

Steels shows how this simple adjustment achieves near-optimal performance in many situations. Moreover, the solution is *cheap* (the computing power required by each agent is minimal) and *robust* (the loss of a single agent will not affect the overall system significantly).

In summary, there are obvious advantages to reactive approaches such as that Brooks' subsumption architecture: simplicity, economy, computational tractability, robustness against failure, and elegance all make such architectures appealing. But there are some fundamental, unsolved problems, not just with the subsumption architecture, but with other purely reactive architectures:

- If agents do not employ models of their environment, then they must have sufficient information available in their *local* environment for them to determine an acceptable action.
- Since purely reactive agents make decisions based on *local* information, (i.e., information about the agents *current* state), it is difficult to see how such decision making could take into account *non-local* information — it must inherently take a "short term" view.
- It is difficult to see how purely reactive agents can be designed that *learn* from experience, and improve their performance over time.
- A major selling point of purely reactive systems is that overall behaviour *emerges* from the interaction of the component behaviours when the agent is placed in its environment. But the very term "emerges" suggests that the relationship between individual behaviours, environment, and overall behaviour is not understandable. This necessarily makes it very hard to *engineer* agents to fulfill specific tasks. Ultimately, there is no principled *methodology* for building such agents: one must use a laborious process of experimentation, trial, and error to engineer an agent.
- While effective agents can be generated with small numbers of behaviours (typically less that ten layers), it is *much* harder to build agents that contain many layers. The dynamics of the interactions between the different behaviours become too complex to understand.

Various solutions to these problems have been proposed. One of the most popular of these is the idea of *evolving* agents to perform certain tasks. This area of work has largely broken away from the mainstream AI tradition in which work on, for example, logic-based agents is carried out, and is documented primarily in the *artificial life* (alife) literature.

Sources and Further Reading

Brooks' original paper on the subsumption architecture — the one that started all the fuss — was published as [8]. The description and discussion here is partly based on [15]. This original paper seems to be somewhat less radical than many of his later ones, which include [9, 11, 10]. The version of the subsumption architecture used in this chapter is actually a simplification of that presented by Brooks. The subsumption architecture is probably the best-known reactive architecture around — but there are many others. The collection of papers edited by Pattie Maes [41] contains papers that describe many of these, as does the collection by Agre and Rosenschein [2]. Other approaches include:

- the *agent network architecture* developed by Pattie Maes [40, 42, 43];
- Nilsson's *teleo reactive programs* [49];
- Rosenchein and Kaelbling's *situated automata* approach, which is particularly interesting in that it shows how agents can be *specified* in an abstract, logical framework, and *compiled* into equivalent, but computationally very simple machines [57, 36, 35, 58];

- Agre and Chapman's PENGI system [1];
- Schoppers' *universal plans* — which are essentially decision trees that can be used to efficiently determine an appropriate action in any situation [62];
- Firby's *reactive action packages* [19].

Kaelbling [34] gives a good discussion of the issues associated with developing resource-bounded rational agents, and proposes an agent architecture somewhat similar to that developed by Brooks.

1.4.3 Belief-Desire-Intention Architectures

In this section, we shall discuss *belief-desire-intention* (BDI) architectures. These architectures have their roots in the philosophical tradition of understanding *practical reasoning* — the process of deciding, moment by moment, which action to perform in the furtherance of our goals.

Practical reasoning involves two important processes: deciding *what* goals we want to achieve, and *how* we are going to achieve these goals. The former process is known as *deliberation*, the latter as *means-ends* reasoning. To gain an understanding of the BDI model, it is worth considering a simple example of practical reasoning. When you leave university with a first degree, you are faced with a decision to make — about what to do with your life. The decision process typically begins by trying to understand what the *options* available to you are. For example, if you gain a good first degree, then one option is that of becoming an academic. (If you fail to obtain a good degree, this option is not available to you.) Another option is entering industry. After generating this set of alternatives, you must *choose between them*, and *commit* to some. These chosen options become *intentions*, which then determine the agent's actions. Intentions then feed back into the agent's future practical reasoning. For example, if I decide I want to be an academic, then I should commit to this objective, and devote time and effort to bringing it about.

Intentions play a crucial role in the practical reasoning process. Perhaps the most obvious property of intentions is that they tend to lead to action. If I truly have an intention to become an academic, then you would expect me to *act* on that intention — to try to achieve it. For example, you might expect me to apply to various PhD programs. You would expect to to make a *reasonable attempt* to achieve the intention. By this, I mean that you would expect me to carry our some course of action that I believed would best satisfy the intention. Moreover, if a course of action fails to achieve the intention, then you would expect me to *try again* — you would not expect me to simply give up. For example, if my first application for a PhD programme is rejected, then you might expect me to apply to alternative universities.

In addition, once I have adopted an intention, then the very fact of having this intention will constrain my future practical reasoning. For example, while I hold some particular intention, I will not entertain options that are inconsistent with that intention. Intending to become an academic, for example, would preclude the option of partying every night: the two are mutually exclusive.

Next, intentions *persist*. If I adopt an intention to become an academic, then I should *persist* with this intention and attempt to achieve it. For if I immediately drop my intentions without devoting resources to achieving them, then I will never achieve anything. However, I should not persist with my intention for too long — if it becomes clear to me that I will *never* become an academic, then it is only rational to drop my intention to do so. Similarly, if the reason for having an intention goes away, then it is rational of me to drop the intention. For example, if I adopted the intention to become an academic because I believed it would be an easy life, but then discover that I would be expected to actually *teach*, then the justification for the intention is no longer present, and I should drop the intention.

Finally, intentions are closely related to beliefs about the future. For example, if I intend to become an academic, then I should believe that I will indeed become an academic. For if I truly believe that I will never be an academic, it would be non-sensical of me to have an intention to become one. Thus if I intend to become an academic, I should at least believe that there is a good chance I will indeed become one.

From this discussion, we can see that intentions play a number of important roles in practical reasoning:

- *Intentions drive means-ends reasoning*.
  If I have formed an intention to become an academic, then I will attempt to achieve the intention, which involves, amongst other things, deciding *how* to achieve it, for example, by applying for a PhD programme. Moreover, if one particular course of action fails to achieve an intention, then I will typically attempt others. Thus if I fail to gain a PhD place at one university, I might try another university.
- *Intentions constrain future deliberation*.
  If I intend to become an academic, then I will not entertain options that are inconsistent with this intention. For example, a rational agent would not consider being rich as an option while simultaneously intending to be an academic. (While the two are not actually mutually exclusive, the probability of simultaneously achieving both is infinitesimal.)
- *Intentions persist*.
  I will not usually give up on my intentions without good reason — they will persist, typically until either I believe I have successfully achieved them, I believe I cannot achieve them, or else because the purpose for the intention is no longer present.
- *Intentions influence beliefs upon which future practical reasoning is based*.
  If I adopt the intention to become an academic, then I can plan for the future on the assumption that I *will* be an academic. For if I intend to be an academic while simultaneously believing that I will never be one, then I am being irrational.

A key problem in the design of practical reasoning agents is that of of achieving a good *balance* between these different concerns. Specifically, it seems clear that an agent should at times drop some intentions (because it comes to believe that either they will never be achieved, they are achieved, or else because the reason for having the intention is no longer present). It follows that, from time to time, it is worth an agent stopping to *reconsider* its intentions. But reconsideration has a cost — in terms of both time and computational resources. But this presents us with a dilemma:

- an agent that does not stop to reconsider sufficiently often will continue attempting to achieve its intentions even after it is clear that they cannot be achieved, or that there is no longer any reason for achieving them;
- an agent that *constantly* reconsiders its attentions may spend insufficient time actually working to achieve them, and hence runs the risk of never actually achieving them.

This dilemma is essentially the problem of balancing pro-active (goal directed) and reactive (event driven) behaviour, that we introduced in section 1.2.2.

There is clearly a tradeoff to be struck between the degree of commitment and reconsideration at work here. The nature of this tradeoff was examined by David Kinny and Michael Georgeff, in a number of experiments carried out with a BDI agent framework called dMARS [37]. They investigate how *bold* agents (those that never stop to reconsider) and *cautious* agents (those that are constantly stopping to reconsider) perform in a variety of different environments. The most important parameter in these experiments was the *rate of world change*, γ. The key results of Kinny and Georgeff were as follows.

- If γ is low, (i.e., the environment does not change quickly), then bold agents do well compared to cautious ones, because cautious ones waste time reconsidering their commitments while bold agents are busy working towards — and achieving — their goals.
- If γ is high, (i.e., the environment changes frequently), then cautious agents tend to outperform bold agents, because they are able to recognise when intentions are doomed, and also to take advantage of serendipitous situations and new opportunities.

The lesson is that different types of environment require different types of decision strategies. In static, unchanging environment, purely pro-active, goal directed behaviour is adequate. But in more dynamic environments, the ability to react to changes by modififying intentions becomes more important.

The process of practical reasoning in a BDI agent is summarised in Figure 1.5. As this Figure illustrates, there are seven main components to a BDI agent:

- a set of current *beliefs*, representing information the agent has about its current environment;
- a *belief revision function*, (brf), which takes a perceptual input and the agent's current beliefs, and on the basis of these, determines a new set of beliefs;
- an *option generation function*, ($options$), which determines the options available to the agent (its desires), on the basis of its current beliefs about its environment and its current *intentions*;
- a set of *current options*, representing possible courses of actions available to the agent;
- a *filter* function ($filter$), which represents the agent's *deliberation* process, and which determines the agent's intentions on the basis of its current beliefs, desires, and intentions;
- a set of current *intentions*, representing the agent's current focus — those states of

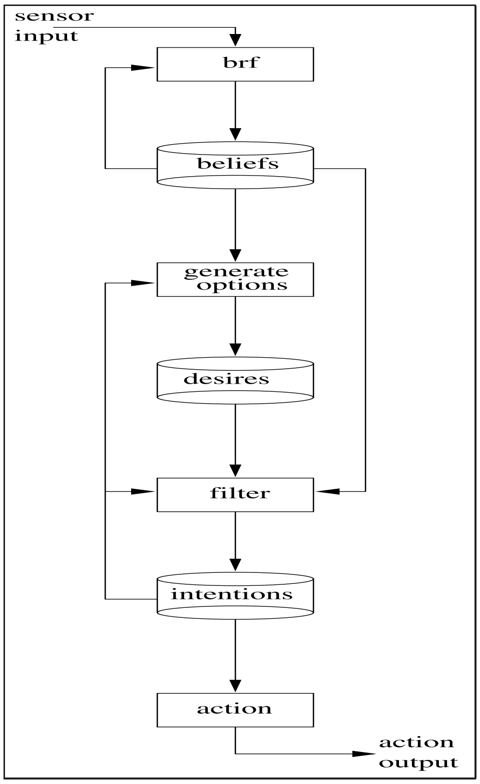

Figure 1.5 Schematic diagram of a generic belief-desire-intention architecture.

affairs that it has committed to trying to bring about;

- an *action selection function* (*execute*), which determines an action to perform on the basis of current intentions.

It is straightforward to formally define these components. First, let *Bel* be the set of all possible beliefs, *Des* be the set of all possible desires, and *Int* be the set of all possible intentions. For the purposes of this chapter, the content of these sets is not important. (Often, beliefs, desires, and intentions are represented as logical formulae, perhaps of first-order logic.) Whatever the content of these sets, its is worth noting that they should have some notion of *consistency* defined upon them, so that one can answer the question of, for example, whether having an intention to achieve x is consistent with the belief that y. Representing beliefs, desires, and intentions as logical formulae permits us to cast such questions as questions as questions of determining whether logical formulae are consistent — a well known and well-understood problem. The state of a BDI agent at any given moment is, unsurprisingly, a triple (B, D, I), where $B \subseteq Bel$, $D \subseteq Des$, and $I \subseteq Int$.

An agent's belief revision function is a mapping

$$brf : \wp(Bel) \times P \to \wp(Bel)$$

which on the basis of the current percept and current beliefs determines a new set of beliefs. Belief revision is out of the scope of this chapter (and indeed this book), and so we shall say no more about it here.

The option generation function, *options*, maps a set of beliefs and a set of intentions to a set of desires.

$$options : \wp(Bel) \times \wp(Int) \rightarrow \wp(Des)$$

This function plays several roles. First, it must be responsible for the agent's means-ends reasoning — the process of deciding how to achieve intentions. Thus, once an agent has formed an intention to x, it must subsequently consider options to *achieve* x. These options will be more concrete — less abstract — than x. As some of these options then become intentions themselves, they will also feedback into option generation, resulting in yet more concrete options being generated. We can thus think of a BDI agent's option generation process as one of recursively elaborating a hierarchical plan structure, considering and committing to progressively more specific intentions, until finally it reaches the intentions that correspond to immediately executable actions.

While the main purpose of the *options* function is thus means-ends reasoning, it must in addition satisfy several other constraints. First, it must be *consistent*: any options generated must be consistent with both the agent's current beliefs and current intentions. Secondly, it must be *opportunistic*, in that it should recognise when environmental circumstances change advantageously, to offer the agent new ways of achieving intentions, or the possibility of achieving intentions that were otherwise unachievable.

A BDI agent's deliberation process (deciding *what* to do) is represented in the *filter* function,

$$filter : \wp(Bel) \times \wp(Des) \times \wp(Int) \rightarrow \wp(Int)$$

which updates the agent's intentions on the basis of its previously-held intentions and current beliefs and desires. This function must fulfill two roles. First, it must *drop* any intentions that are no longer achievable, or for which the expected cost of achieving them exceeds the expected gain associated with successfully achieving them. Second, it should *retain* intentions that are not achieved, and that are still expected to have a positive overall benefit. Finally, it should *adopt* new intentions, either to achieve existing intentions, or to exploit new opportunities.

Notice that we do not expect this function to introduce intentions from nowhere. Thus *filter* should satisfy the following constraint:

$$\forall B \in \wp(Bel), \forall D \in \wp(Des), \forall I \in \wp(Int), filter(B,D,I) \subseteq I \cup D.$$

In other words, current intentions are either previously held intentions or newly adopted options.

The *execute* function is assumed to simply return any executable intentions — by which we mean intentions that correspond to directly executable actions:

$$execute : \wp(Int) \rightarrow A$$

The agent decision function, *action* of a BDI agent is then a function

$$action : P \to A$$

and is defined by the following pseudo-code.

```
function action(p : P) : A
begin
        B := brf(B, p)
        D := options(D, I)
        I := filter(B, D, I)
        return execute(I)
end function action
```

Note that representing an agent's intentions as a *set* (i.e., as an unstructured collection) is generally too simplistic in practice. A simple alternative is to associate a *priority* with each intention, indicating its relative importance. Another natural idea is to represent intentions as a *stack*. An intention is pushed on to the stack when it is adopted, and popped when it is either achieved or else not achievable. More abstract intentions will tend to be at the bottom of the stack, with more concrete intentions towards the top.

To summarise, BDI architectures are practical reasoning architectures, in which the process of deciding what to do resembles the kind of practical reasoning that we appear to use in our everyday lives. The basic components of a BDI architecture are data structures representing the beliefs, desires, and intentions of the agent, and functions that represent its deliberation (deciding *what* intentions to have — i.e., deciding what to do) and means-ends reasoning (deciding how to do it). Intentions play a central role in the BDI model: they provide stability for decision making, and act to focus the agent's practical reasoning. A major issue in BDI architectures is the problem of striking a *balance* between being committed to and overcommitted to one's intentions: the deliberation process must be finely tuned to its environment, ensuring that in more dynamic, highly unpredictable domains, it reconsiders its intentions relatively frequently — in more static environments, less frequent reconsideration is necessary.

The BDI model is attractive for several reasons. First, it is intuitive — we all recognise the processes of deciding what to do and then how to do it, and we all have an informal understanding of the notions of belief, desire, and intention. Second, it gives us a clear functional decomposition, which indicates what sorts of subsystems might be required to build an agent. But the main difficulty, as ever, is knowing how to efficiently implement these functions.

Sources and Further Reading

Belief-desire-intention architectures originated in the work of the Rational Agency project at Stanford Research Institute in the mid 1980s. The origins of the model lie in the theory of human practical reasoning developed by the philosopher Michael Bratman [6], which focusses particularly on the role of intentions in practical reasoning. The conceptual framework of the BDI model is described in [7], which also describes a specific BDI agent architecture called IRMA. The description of the BDI model given here (and in particular Figure 1.5) is adapted from [7]. One of the interesting aspects of the BDI model is that it has been used in one of the most successful agent architectures to date. The Procedural Resoning System (PRS), originally developed by Michael Georgeff and Amy Lansky [26], has been used to build some of the most exacting agent applications to date, including fault diagnosis for the reaction control system of the space shuttle, and an air traffic management system at Sydney airport in Australia — overviews of these systems are described in [27]. In the PRS, an agent is equipped with a library of *plans* which are used to perform means-ends reasoning. Deliberation is achieved by the use of *meta-level plans*, which are able to modify an agent's intention structure at run-time, in order to change the focus of the agent's practical reasoning. Beliefs in the PRS are represented as PROLOG-like facts — essentially, as atoms of first-order logic.

The BDI model is also interesting because a great deal of effort has been devoted to formalising it. In particular, Anand Rao and Michael Georgeff have developed a range of BDI logics, which they use to axiomatise properties of BDI-based practical reasoning agents [52, 56, 53, 54, 55, 51]. These models have been extended by others to deal with, for example, communication between agents [28].

1.4.4 Layered Architectures

Given the requirement that an agent be capable of reactive and pro-active behaviour, an obvious decomposition involves creating separate subsystems to deal with these different types of behaviours. This idea leads naturally to a class of architectures in which the various subsystems are arranged into a hierarchy of interacting *layers*. In this section, we will consider some general aspects of layered architectures, and then go on to consider two examples of such architectures: INTERRAP and TOURINGMACHINES.

Typically, there will be at least two layers, to deal with reactive and pro-active behaviours respectively. In principle, there is no reason why there should not be many more layers. However many layers there are, a useful typology for such architectures is by the information and control flows within them. Broadly speaking, we can identify two types of control flow within layered architectures (see Figure 1.6):

- *Horizontal layering*.
  In horizontally layered architectures (Figure 1.6(a)), the software layers are each directly connected to the sensory input and action output. In effect, each layer itself acts like an agent, producing suggestions as to what action to perform.
- *Vertical layering*.
  In vertically layered architectures (Figure 1.6(b) and 1.6(c)), sensory input and action output are each dealt with by at most one layer each.

The great advantage of horizontally layered architectures is their conceptual simplicity: if we need an agent to exhibit n different types of behaviour, then we implement n different layers. However, because the layers are each in effect competing with one-another to

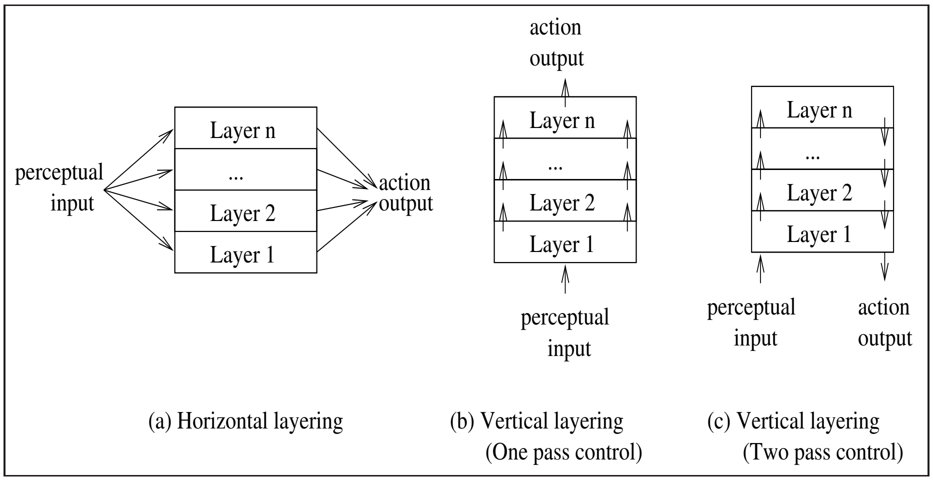

Figure 1.6 Information and control flows in three types of layered agent architecture (Source: [47, p263]).

generate action suggestions, there is a danger that the *overall* behaviour of the agent will not be coherent. In order to ensure that horizontally layered architectures *are* consistent, they generally include a *mediator* function, which makes decisions about which layer has "control" of the agent at any given time. The need for such central control is problematic: it means that the designer must potentially consider all possible interactions between layers. If there are n layers in the architecture, and each layer is capable of suggesting m possible actions, then this means there are m^n such interactions to be considered. This is clearly difficult from a design point of view in any but the most simple system. The introduction of a central control system also introduces a *bottleneck* into the agent's decision making.

These problems are partly alleviated in a vertically layered architecture. We can subdivide vertically layered architectures into *one pass* architectures (Figure 1.6(b)) and *two pass* architectures (Figure 1.6(c)). In one-pass architectures, control flows sequentially through each layer, until the final layer generates action output. In two-pass architectures, information flows up the architecture (the first pass) and control then flows back down. There are some interesting similarities between the idea of two-pass vertically layered architectures and the way that organisations work, with information flowing up to the highest levels of the organisation, and commands then flowing down. In both one pass and two pass vertically layered architectures, the complexity of interactions between layers is reduced: since there are $n-1$ interfaces between n layers, then if each layer is capable of suggesting m actions, there are at most $m^2(n-1)$ interactions to be considered between layers. This is clearly much simpler than the horizontally layered case. However, this simplicity comes at the cost of some flexibility: in order for a vertically layered architecture to make a decision, control must pass between *each* different layer. This is not fault tolerant: failures in any one layer are likely to have serious consequences for agent performance.

In the remainder of this section, we will consider two examples of layered architectures: Innes Ferguson's TOURINGMACHINES, and Jörg Müller's INTERRAP. The former is an

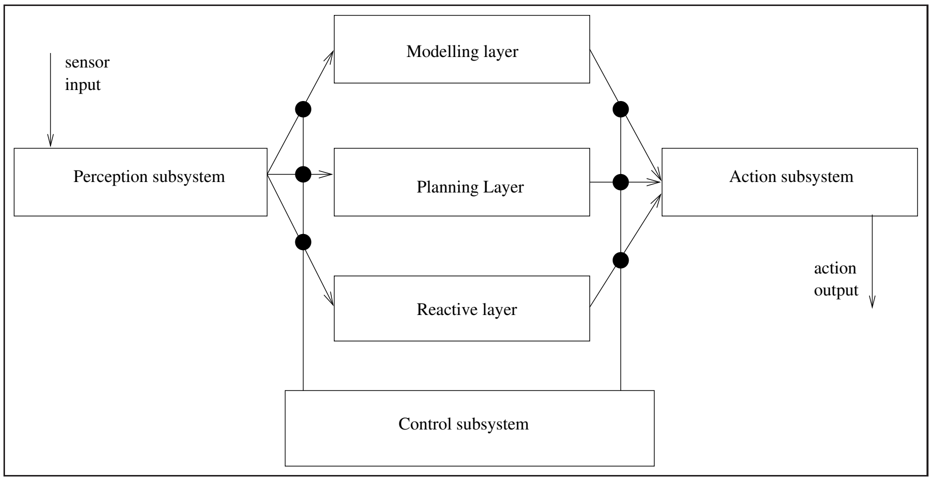

Figure 1.7 TOURINGMACHINES: a horizontally layered agent architecture

example of a horizontally layered architecture; the latter is a (two pass) vertically layered architecture.

TouringMachines

The TOURINGMACHINES architecture is illustrated in Figure 1.7. As this Figure shows, TOURINGMACHINES consists of three *activity producing layers*. That is, each layer continually produces "suggestions" for what actions the agent should perform. The *reactive layer* provides a more-or-less immediate response to changes that occur in the environment. It is implemented as a set of situation-action rules, like the behaviours in Brooks' subsumption architecture (section 1.4.2). These rules map sensor input directly to effector output. The original demonstration scenario for TOURINGMACHINES was that of autonomous vehicles driving between locations through streets populated by other similar agents. In this scenario, reactive rules typically deal with functions like obstacle avoidance. For example, here is an example of a reactive rule for avoiding the kerb (from [16, p59]):

```
rule-1: kerb-avoidance
    if
        is-in-front(Kerb, Observer) and
        speed(Observer) > 0 and
        separation(Kerb, Observer) < KerbThreshHold
    then
        change-orientation(KerbAvoidanceAngle)
```

Here `change-orientation(...)` is the action suggested if the rule fires. The rules can only make references to the agent's current state — they cannot do any explicit reasoning about the world, and on the right hand side of rules are *actions*, not predicates. Thus if this rule fired, it would not result in any central environment model being updated, but would just result in an action being suggested by the reactive layer.

The TOURINGMACHINES *planning layer* achieves the agent's pro-active behaviour. Specifically, the planning layer is responsible for the "day-to-day" running of the agent — under normal circumstances, the planning layer will be responsible for deciding what the agent does. However, the planning layer does not do "first-principles" planning. That is, it does not attempt to generate plans from scratch. Rather, the planning layer employs a *library* of plan "skeletons" called *schemas*. These skeletons are in essence hierarchically structured plans, which the TOURINGMACHINES planning layer elaborates at run time in order to decide what to do. So, in order to achieve a goal, the planning layer attempts to find a schema in its library which matches that goal. This schema will contain sub-goals, which the planning layer elaborates by attempting to find other schemas in its plan library that match these sub-goals.

The *modeling* layer represents the various entities in the world (including the agent itself, as well as other agents). The modeling layer thus predicts conflicts between agents, and generates new goals to be achieved in order to resolve these conflicts. These new goals are then posted down to the planning layer, which makes use of its plan library in order to determine how to satisfy them.

The three control layers are embedded within a *control subsystem*, which is effectively responsible for deciding which of the layers should have control over the agent. This control subsystem is implemented as a set of *control rules*. Control rules can either *suppress* sensor information between the control rules and the control layers, or else *censor* action outputs from the control layers. Here is an example censor rule [18, p207]:

```
censor-rule-1:
    if
        entity(obstacle-6) in perception-buffer
    then
        remove-sensory-record(layer-R, entity(obstacle-6))
```

This rule prevents the reactive layer from ever knowing about whether `obstacle-6` has been perceived. The intuition is that although the reactive layer will in general be the most appropriate layer for dealing with obstacle avoidance, there are certain obstacles for which other layers are more appropriate. This rule ensures that the reactive layer never comes to know about these obstacles.

InteRRaP

INTERRAP is an example of a vertically layered two-pass agent architecture — see Figure 1.8.

As Figure 1.8 shows, INTERRAP contains three control layers, as in TOURINGMACHINES. Moreover, the purpose of each INTERRAP layer appears to be rather similar to the purpose of each corresponding TOURINGMACHINES layer. Thus the lowest (*behaviour based*) layer deals with reactive behaviour; the middle (*local planning*) layer deals with everyday planning to achieve the agent's goals, and the uppermost (*cooperative planning*) layer deals with social interactions. Each layer has associated with it a *knowledge base*, i.e.,

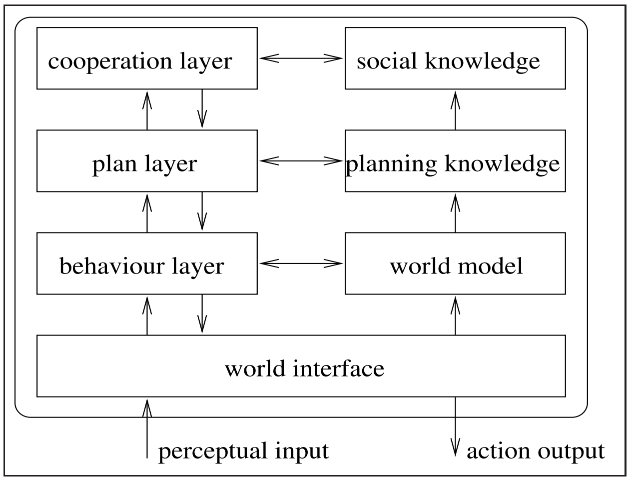

Figure 1.8 INTERRAP – a vertically layered two-pass agent architecture.

a representation of the world appropriate for that layer. These different knowledge bases represent the agent and its environment at different levels of abstraction. Thus the highest level knowledge base represents the plans and actions of other agents in the environment; the middle-level knowledge base represents the plans and actions of the agent itself; and the lowest level knowledge base represents "raw" information about the environment. The explicit introduction of these knowledge bases distinguishes TOURINGMACHINES from INTERRAP.

The way the different layers in INTERRAP conspire to produce behaviour is also quite different from TOURINGMACHINES. The main difference is in the way the layers interract with the environment. In TOURINGMACHINES, each layer was directly coupled to perceptual input and action output. This necessitated the introduction of a supervisory control framework, to deal with conflicts or problems between layers. In INTERRAP, layers interact with *each other* to achieve the same end. The two main types of interaction between layers are *bottom-up activation* and *top-down execution*. Bottom-up activation occurs when a lower layer passes control to a higher layer because it is not *competent* to deal with the current situation. Top-down execution occurs when a higher layer makes use of the facilities provided by a lower layer to achieve one of its goals. The basic flow of control in INTERRAP begins when perceptual input arrives at the lowest layer in the achitecture. If the reactive layer can deal with this input, then it will do so; otherwise, bottom-up activation will occur, and control will be passed to the local planning layer. If the local planning layer can handle the situation, then it will do so, typically by making use of top-down execution. Otherwise, it will use bottom-up activation to pass control to the highest layer. In this way, control in INTERRAP will flow from the lowest layer to higher layers of the architecture, and then back down again.

The internals of each layer are not important for the purposes of this article. However, it is worth noting that each layer implements two general functions. The first of these is a *situation recognition and goal activation* function. This function acts rather like the *options* function in a BDI architecture (see section 1.4.3). It maps a knowledge base (one of the three layers) and current goals to a new set of goals. The second function is responsible for *planning and scheduling* — it is responsible for selecting which plans to execute, based on the current plans, goals, and knowledge base of that layer.

Layered architectures are currently the most popular general class of agent architecture available. Layering represents a natural decomposition of functionality: it is easy to see how reactive, pro-active, social behaviour can be generated by the reactive, pro-active, and social layers in an architecture. The main problem with layered architectures is that while they are arguably a *pragmatic* solution, they lack the conceptual and semantic clarity of unlayered approaches. In particular, while logic-based approaches have a clear logical semantics, it is difficult to see how such a semantics could be devised for a layered architecture. Another issue is that of interactions between layers. If each layer is an independent activity producing process (as in TOURINGMACHINES), then it is necessary to consider all possible ways that the layers can interact with one another. This problem is partly alleviated in two-pass vertically layered architecture such as INTERRAP.

Sources and Further Reading

The introductory discussion of layered architectures given here draws heavily upon [47, pp262–264]. The best reference to TOURINGMACHINES is [16]; more accessible references include [17, 18]. The definitive reference to INTERRAP is [46], although [20] is also a useful reference. Other examples of layered architectures include the subsumption architecture [8] (see also section 1.4.2), and the 3T architecture [4].

1.5 Agent Programming Languages

As agent technology becomes more established, we might expect to see a variety of software tools become available for the design and construction of agent-based systems; the need for software support tools in this area was identified as long ago as the mid-1980s [23]. In this section, we will discuss two of the better-known agent programming languages, focussing in particular on Yoav Shoham's AGENT0 system.

1.5.1 Agent-oriented programming

Yoav Shoham has proposed a "new programming paradigm, based on a societal view of computation" which he calls *agent-oriented programming*. The key idea which informs AOP is that of directly programming agents in terms of *mentalistic* notions (such as belief, desire, and intention) that agent theorists have developed to represent the properties of agents. The motivation behind the proposal is that humans use such concepts as an *abstraction* mechanism for representing the properties of complex systems. In the same way that we use these mentalistic notions to describe and explain the behaviour of humans, so it might be useful to use them to program machines.

The first implementation of the agent-oriented programming paradigm was the AGENT0 programming language. In this language, an agent is specified in terms of a set of capabilities (things the agent can do), a set of initial *beliefs* (playing the role of beliefs in BDI architectures), a set of initial *commitments* (playing a role similar to that of intentions in BDI architectures), and a set of *commitment rules*. The key component, which determines how the agent acts, is the commitment rule set. Each commitment rule contains a *message condition*, a *mental condition*, and an action. In order to determine whether such a rule fires, the message condition is matched against the messages the agent has received; the mental condition is matched against the beliefs of the agent. If the rule fires, then the agent becomes committed to the action. Actions may be *private*, corresponding to an internally executed subroutine, or *communicative*, i.e., sending messages. Messages are constrained to be one of three types: "requests" or "unrequests" to perform or refrain from actions, and "inform" messages, which pass on information — Shoham indicates that he took his inspiration for these message types from speech act theory [63, 12]. Request and unrequest messages typically result in the agent's commitments being modified; inform messages result in a change to the agent's beliefs.

Here is an example of an AGENT0 commitment rule:

```
COMMIT(
  ( agent, REQUEST, DO(time, action)
  ), ;;; msg condition
  ( B,
    [now, Friend agent] AND
    CAN(self, action) AND
    NOT [time, CMT(self, anyaction)]
  ), ;;; mental condition
  self,
  DO(time, action)
)
```

This rule may be paraphrased as follows:

if I receive a message from agent which requests me to do action at time, and I believe that:

- *agent is currently a friend;*
- *I can do the action;*
- *at time, I am not committed to doing any other action,*

then commit to doing action at time.

The operation of an agent can be described by the following loop (see Figure 1.9):

1. Read all current messages, updating beliefs — and hence commitments — where necessary;
2. Execute all commitments for the current cycle where the capability condition of the associated action is satisfied;

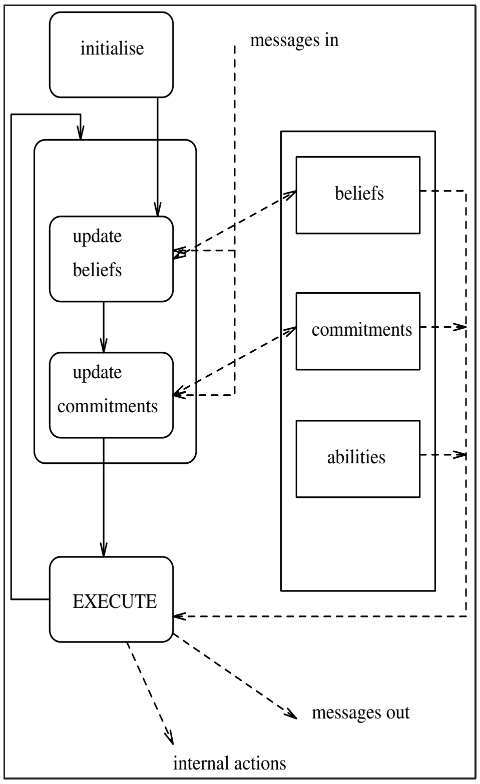

Figure 1.9 The flow of control in AGENT-0.

3. Goto (1).

It should be clear how more complex agent behaviours can be designed and built in AGENT0. However, it is important to note that this language is essentially a *prototype*, not intended for building anything like large-scale production systems. However, it does at least give a feel for how such systems might be built.

1.5.2 Concurrent METATEM

Arguably, one drawback with AGENT0 is that the relationship between the logic and interpreted programming language is only loosely defined. The programming language cannot be said to truly *execute* the associated logic, in the way that our logic-based agents did in section 1.4.1. The Concurrent METATEM language developed by Fisher can make a stronger claim in this respect [21]. A Concurrent METATEM system contains a number of concurrently executing agents, each of which is able to communicate with its peers via asynchronous broadcast message passing. Each agent is programmed by giving it a temporal logic specification of the behaviour that it is intended the agent should exhibit. An agent's specification is executed directly to generate its behaviour. Execution of the agent program corresponds to iteratively building a logical model for the temporal agent specification. It is possible to prove that the procedure used to execute an agent specification is correct, in that if it is possible to satisfy the specification, then the agent will do so [3].

The logical semantics of Concurrent METATEM are closely related to the semantics of temporal logic itself. This means that, amongst other things, the specification and verification of Concurrent METATEM systems is a realistic proposition [22].

An agent program in Concurrent METATEM has the form $\bigwedge_i P_i \Rightarrow F_i$, where P_i is a temporal logic formula referring only to the present or past, and F_i is a temporal logic formula referring to the present or future. The $P_i \Rightarrow F_i$ formulae are known as *rules*. The basic idea for executing such a program may be summed up in the following slogan:

on the basis of the past *do* the future.

Thus each rule is continually matched against an internal, recorded *history*, and if a match is found, then the rule *fires*. If a rule fires, then any variables in the future time part are instantiated, and the future time part then becomes a *commitment* that the agent will subsequently attempt to satisfy. Satisfying a commitment typically means making some predicate true within the agent. Here is a simple example of a Concurrent METATEM agent definition:

$$
\begin{aligned}
&rc(ask)[give]: \\
&\quad \circledcirc ask(x) \Rightarrow \Diamond give(x) \\
&\quad (\neg ask(x)\ \mathcal{Z}\, (give(x) \wedge \neg ask(x)) \Rightarrow \neg give(x) \\
&\quad give(x) \wedge give(y) \Rightarrow (x = y)
\end{aligned}
$$

The agent in this example is a controller for a resource that is infinitely renewable, but which may only be possessed by one agent at any given time. The controller must therefore enforce mutual exclusion over this resource. The first line of the program defines the *interface* to the agent: its name is *rc* (for resource controller), and it will accept *ask* messages and send *give* messages. The following three lines constitute the agent program itself. The predicate $ask(x)$ means that agent x has asked for the resource. The predicate $give(x)$ means that the resource controller has given the resource to agent x. The resource controller is assumed to be the only agent able to 'give' the resource. However, many agents may ask for the resource simultaneously. The three rules that define this agent's behaviour may be summarized as follows:

Rule 1: if someone has just asked for the resource, then eventually give them the resource;

Rule 2: don't give unless someone has asked since you last gave; and

Rule 3: if you give to two people, then they must be the same person (i.e., don't give to more than one person at a time).

Concurrent METATEM is a good illustration of how a quite pure approach to logic-based agent programming can work, even with a quite expressive logic.

Sources and Further Reading

The main references to AGENT0 are [64, 65]. Michael Fisher's Concurrent METATEM language is described in [21]; the execution algorithm that underpins it is described in [3]. Since Shoham's proposal, a number of languages have been proposed which claim to be agent-oriented. Examples include Becky Thomas's Planning Communicating Agents (PLACA) language [67, 68], MAIL [30], and Anand Rao's AGENTSPEAK(L) language [50]. APRIL is a language that is intended to be used for building multi-agent systems, although it is not "agent-oriented" in the sense that Shoham describes [44]. The TELESCRIPT programming language, developed by General Magic, Inc., was the first *mobile* agent programming language [69]. That is, it explicitly supports the idea of agents as processes that have the ability to autonomously move themselves across a computer network and recommence executing at a remote site. Since TELESCRIPT was announced, a number of mobile agent extensions to the JAVA programming language have been developed.

1.6 Conclusions

I hope that after reading this chapter, you understand what agents are and why they are considered to be an important area of research and development. The requirement for systems that can operate autonomously is very common. The requirement for systems capable of *flexible* autonomous action, in the sense that I have described in this chapter, is similarly common. This leads me to conclude that intelligent agents have the potential to play a significant role in the future of software engineering. Intelligent agent research is about the theory, design, construction, and application of such systems. This chapter has focussed on the design of intelligent agents. It has presented a high-level, abstract view of intelligent agents, and described the sort of properties that one would expect such an agent to enjoy. It went on to show how this view of an agent could be refined into various different types of agent architecture — purely logical agents, purely reactive/behavioural agents, BDI agents, and layered agent architectures.

1.7 Exercises

1. *[Level 1]*
   Give other examples of agents (not necessarily intelligent) that you know of. For each, define as precisely as possible:
   - the environment that the agent occupies (physical, software, ...), the states that this environment can be in, and whether the environment is: accessible or inaccessible; deterministic or non-deterministic; episodic or non-episodic; static or dynamic; discrete or continuous.
   - the action repertoire available to the agent, and any pre-conditions associated with these actions;

- the goal, or design objectives of the agent — what it is intended to achieve.

2. *[Level 1]*
   (a) Prove that for every purely reactive agent, these is a behaviourally equivalent standard agent.
   (b) Prove that there exist standard agents that have no behaviourally equivalent purely reactive agent.
3. *[Level 1]*
   Prove that state-based agents are equivalent in expressive power to standard agents, i.e., that for every state-based agent there is a behaviourally equivalent standard agent and vice versa.
4. *[Level 2]*
   The following few questions refer to the vacuum world example described in section 1.4.1.
   Give the full definition (using pseudo-code if desired) of the *new* function, which defines the predicates to add to the agent's database.
5. *[Level 2]*
   Complete the vacuum world example, by filling in the missing rules. How intuitive do you think the solution is? How elegant is it? How compact is it?
6. *[Level 2]*
   Try using your favourite (imperative) programming language to code a solution to the basic vacuum world example. How do you think it compares to the logical solution? What does this tell you about trying to encode essentially *procedural* knowledge (i.e., knowledge about what action to perform) as purely logical rules?
7. *[Level 2]*
   If you are familiar with PROLOG, try encoding the vacuum world example in this language and running it with randomly placed dirt. Make use of the `assert` and `retract` meta-level predicates provided by PROLOG to simplify your system (allowing the program itself to achieve much of the operation of the *next* function).
8. *[Level 2]*
   Develop a solution to the vacuum world example using the behaviour-based approach described in section 1.4.2. How does it compare to the logic-based example?
9. *[Level 2]*
   Try scaling the vacuum world up to a 10×10 grid size. Approximately how many rules would you need to encode this enlarged example, using the approach presented above? Try to generalise the rules, encoding a more general decision making mechanism.
10. *[Level 3]*
    Suppose that the vacuum world could also contain *obstacles*, which the agent needs to avoid. (Imagine it is equipped with a sensor to detect such obstacles.) Try to adapt the example to deal with obstacle detection and avoidance. Again, compare a logic-based solution to one implemented in a traditional (imperative) programming language.

11. *[Level 3]*
Suppose the agent's sphere of perception in the vacuum world is enlarged, so that it can see the *whole* of its world, and see *exactly* where the dirt lay. In this case, it would be possible to generate an *optimal* decision-making algorithm — one which cleared up the dirt in the smallest time possible. Try and think of such general algorithms, and try to code them both in first-order logic and a more traditional programming language. Investigate the effectiveness of these algorithms when there is the possibility of *noise* in the perceptual input the agent receives, (i.e., there is a non-zero probability that the perceptual information is wrong), and try to develop decision-making algorithms that are robust in the presence of such noise. How do such algorithms perform as the level of perception is reduced?
12. *[Level 2]*
Try developing a solution to the Mars explorer example from section 1.4.2 using the logic-based approach. How does it compare to the reactive solution?
13. *[Level 3]*
In the programming language of your choice, implement the Mars explorer example using the subsumption architecture. (To do this, you may find it useful to implement a simple subsumption architecture "shell" for programming different behaviours.) Investigate the performance of the two approaches described, and see if you can do better.
14. *[Level 3]*
Using the simulator implemented for the preceding question, see what happens as you increase the number of agents. Eventually, you should see that overcrowding leads to a sub-optimal solution — agents spend too much time getting out of each other's way to get any work done. Try to get around this problem by allowing agents to pass samples to each other, thus implementing *chains*. (See the description in [15, p305].)
15. *[Level 4]*
Read about traditional *control theory*, and compare the problems and techniques of control theory to what are trying to accomplish in building intelligent agents. How are the techniques and problems of traditional control theory similar to those of intelligent agent work, and how do they differ?
16. *[Level 4]*
One advantage of the logic-based approach to building agents is that the logic-based architecture is *generic*: first-order logic turns out to extremely powerful and useful for expressing a range of different properties. Thus it turns out to be possible to use the logic-based architecture to *encode* a range of other architectures. For this exercise, you should attempt to use first-order logic to encode the different architectures (reactive, BDI, layered) described in this chapter. (You will probably need to read the original references to be able to do this.) Once completed, you will have a logical *theory* of the architecture, that will serve both as a formal specification of the architecture, and also as a precise mathematical model of it, amenable to proof. Once you have your logically-specified architecture, try to *animate* it, by mapping your

logical theory of it into, say the PROLOG programming language. What compromises do you have to make? Does it seem worthwhile trying to directly program the system in logic, or would it be simpler to implement your system in a more pragmatic programming language (such as JAVA)?

1. P. Agre and D. Chapman. PENGI: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 268–272, Seattle, WA, 1987.
2. P. E. Agre and S. J. Rosenschein, editors. *Computational Theories of Interaction and Agency*. The MIT Press: Cambridge, MA, 1996.
3. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS Volume 430)*, pages 94–129. Springer-Verlag: Berlin, Germany, June 1989.
4. R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 187–202. Springer-Verlag: Berlin, Germany, 1996.
5. G. Booch. *Object-Oriented Analysis and Design (second edition)*. Addison-Wesley: Reading, MA, 1994.
6. M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press: Cambridge, MA, 1987.
7. M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4:349–355, 1988.
8. R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.
9. R. A. Brooks. Elephants don't play chess. In P. Maes, editor, *Designing Autonomous Agents*, pages 3–15. The MIT Press: Cambridge, MA, 1990.
10. R. A. Brooks. Intelligence without reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991.
11. R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
12. P. R. Cohen and C. R. Perrault. Elements of a plan based theory of speech acts. *Cognitive Science*, 3:177–212, 1979.
13. Oren Etzioni. Intelligence without robots. *AI Magazine*, 14(4), December 1993.
14. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.
15. J. Ferber. Reactive distributed artificial intelligence. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 287–317. John Wiley, 1996.
16. I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, Clare Hall, University of Cambridge, UK, November 1992. (Also available as Technical Report No. 273, University of Cambridge Computer Laboratory).
17. I. A. Ferguson. Towards an architecture for adaptive, rational, mobile agents. In E. Werner and Y. Demazeau, editors, *Decentralized AI 3 — Proceedings of the Third European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-91)*, pages

249–262. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1992.

18. I. A. Ferguson. Integrated control and coordinated behaviour: A case for agent models. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 203–218. Springer-Verlag: Berlin, Germany, January 1995.
19. J. A. Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 202–206, Milan, Italy, 1987.
20. K. Fischer, J. P. Müller, and M. Pischel. A pragmatic BDI architecture. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 203–218. Springer-Verlag: Berlin, Germany, 1996.
21. M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Berlin, Germany, July 1994.
22. M. Fisher and M. Wooldridge. Specifying and verifying distributed intelligent systems. In M. Filgueiras and L. Damas, editors, *Progress in Artificial Intelligence — Sixth Portuguese Conference on Artificial Intelligence (LNAI Volume 727)*, pages 13–28. Springer-Verlag: Berlin, Germany, October 1993.
23. L. Gasser, C. Braganza, and N. Hermann. MACE: A flexible testbed for distributed AI research. In M. Huhns, editor, *Distributed Artificial Intelligence*, pages 119–152. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1987.
24. L. Gasser and J. P. Briot. Object-based concurrent programming and DAI. In *Distributed Artificial Intelligence: Theory and Praxis*, pages 81–108. Kluwer Academic Publishers: Boston, MA, 1992.
25. M. R. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1987.
26. M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.
27. M. P. Georgeff and A. S. Rao. A profile of the Australian AI Institute. *IEEE Expert*, 11(6):89–92, December 1996.
28. A. Haddadi. *Communication and Cooperation in Agent Systems (LNAI Volume 1056)*. Springer-Verlag: Berlin, Germany, 1996.
29. J. Y. Halpern. Using reasoning about knowledge to analyze distributed systems. *Annual Review of Computer Science*, 2:37–68, 1987.
30. H. Haugeneder, D. Steiner, and F. G. McCabe. IMAGINE: A framework for building multi-agent systems. In S. M. Deen, editor, *Proceedings of the 1994 International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, pages 31–64, DAKE Centre, University of Keele, UK, 1994.
31. F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors. *Building Expert Systems*. Addison-Wesley: Reading, MA, 1983.
32. P. Jackson. *Introduction to Expert Systems*. Addison-Wesley: Reading, MA, 1986.
33. N. R. Jennings, J. Corera, I. Laresgoiti, E. H. Mamdani, F. Perriolat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control. *IEEE Expert*, dec 1996.
34. L. P. Kaelbling. An architecture for intelligent reactive systems. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning About Actions & Plans — Proceedings of the 1986 Workshop*,

pages 395–410. Morgan Kaufmann Publishers: San Mateo, CA, 1986.

35. L. P. Kaelbling. A situated automata approach to the design of embedded agents. *SIGART Bulletin*, 2(4):85–88, 1991.
36. L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In P. Maes, editor, *Designing Autonomous Agents*, pages 35–48. The MIT Press: Cambridge, MA, 1990.
37. D. Kinny and M. Georgeff. Commitment and effectiveness of situated agents. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 82–88, Sydney, Australia, 1991.
38. K. Konolige. *A Deduction Model of Belief*. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1986.
39. Y. Lésperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 331–346. Springer-Verlag: Berlin, Germany, 1996.
40. P. Maes. The dynamics of action selection. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 991–997, Detroit, MI, 1989.
41. P. Maes, editor. *Designing Autonomous Agents*. The MIT Press: Cambridge, MA, 1990.
42. P. Maes. Situated agents can have goals. In P. Maes, editor, *Designing Autonomous Agents*, pages 49–70. The MIT Press: Cambridge, MA, 1990.
43. P. Maes. The agent network architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991.
44. F. G. M^{c}Cabe and K. L. Clark. `April` — agent process interaction language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 324–340. Springer-Verlag: Berlin, Germany, January 1995.
45. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.
46. J. Müller. A cooperation model for autonomous agents. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 245–260. Springer-Verlag: Berlin, Germany, 1997.
47. J. P. Müller, M. Pischel, and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 261–276. Springer-Verlag: Berlin, Germany, January 1995.
48. J. P. Müller, M. Wooldridge, and N. R. Jennings, editors. *Intelligent Agents III (LNAI Volume 1193)*. Springer-Verlag: Berlin, Germany, 1995.
49. N. J. Nilsson. Towards agent programs with circuit semantics. Technical Report STAN–CS–92–1412, Computer Science Department, Stanford University, Stanford, CA 94305, January 1992.
50. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 42–55. Springer-Verlag: Berlin, Germany, 1996.
51. A. S. Rao. Decision procedures for propositional linear-time Belief-Desire-Intention logics. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 33–48. Springer-Verlag: Berlin, Germany, 1996.
52. A. S. Rao and M. P. Georgeff. Asymmetry thesis and side-effect problems in linear time and

branching time intention logics. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 498–504, Sydney, Australia, 1991.

53. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.
54. A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.
55. A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324, Chambéry, France, 1993.
56. A. S. Rao, M. P. Georgeff, and E. A. Sonenberg. Social plans: A preliminary report. In E. Werner and Y. Demazeau, editors, *Decentralized AI 3 — Proceedings of the Third European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-91)*, pages 57–76. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1992.
57. S. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–98. Morgan Kaufmann Publishers: San Mateo, CA, 1986.
58. S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. In P. E. Agre and S. J. Rosenschein, editors, *Computational Theories of Interaction and Agency*, pages 515–540. The MIT Press: Cambridge, MA, 1996.
59. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.
60. S. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of AI Research*, 2:575–609, 1995.
61. S. J. Russell and E. Wefald. *Do the Right Thing — Studies in Limited Rationality*. The MIT Press: Cambridge, MA, 1991.
62. M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046, Milan, Italy, 1987.
63. J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press: Cambridge, England, 1969.
64. Y. Shoham. Agent-oriented programming. Technical Report STAN–CS–1335–90, Computer Science Department, Stanford University, Stanford, CA 94305, 1990.
65. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
66. L. Steels. Cooperation between distributed agents through self organization. In Y. Demazeau and J.-P. Müller, editors, *Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)*, pages 175–196. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.
67. S. R. Thomas. *PLACA, an Agent Oriented Programming Language*. PhD thesis, Computer Science Department, Stanford University, Stanford, CA 94305, August 1993. (Available as technical report STAN–CS–93–1487).
68. S. R. Thomas. The PLACA agent programming language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 355–369. Springer-Verlag: Berlin, Germany, January 1995.
69. J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.

70. M. Wooldridge. Agent-based software engineering. *IEE Transactions on Software Engineering*, 144(1):26–37, February 1997.
71. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Acknowledgment

Thanks to Gerhard Weiß, for his patience and enthusiasm.