3

Deductive Reasoning Agents

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 chapter, 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*.

I will begin by giving an example to informally introduce the ideas behind deductive reasoning agents. Suppose we have some robotic agent, the purpose of which is to navigate around an office building picking up trash. There are many possible ways of implementing the control system for such a robot – we shall see several in the chapters that follow – but one way is to give it a description, or *representation* of the environment in which it is to operate. Figure 3.1 illustrates the idea (adapted from Konolige (1986, p. 15)).

> RALPH is an autonomous robot agent that operates in a real-world environment of corridors and big blocks. Sensory input is from a video camera; a subsystem labelled 'interp' in Figure 3.1 translates the video feed into an internal representation format, based on first-order logic.

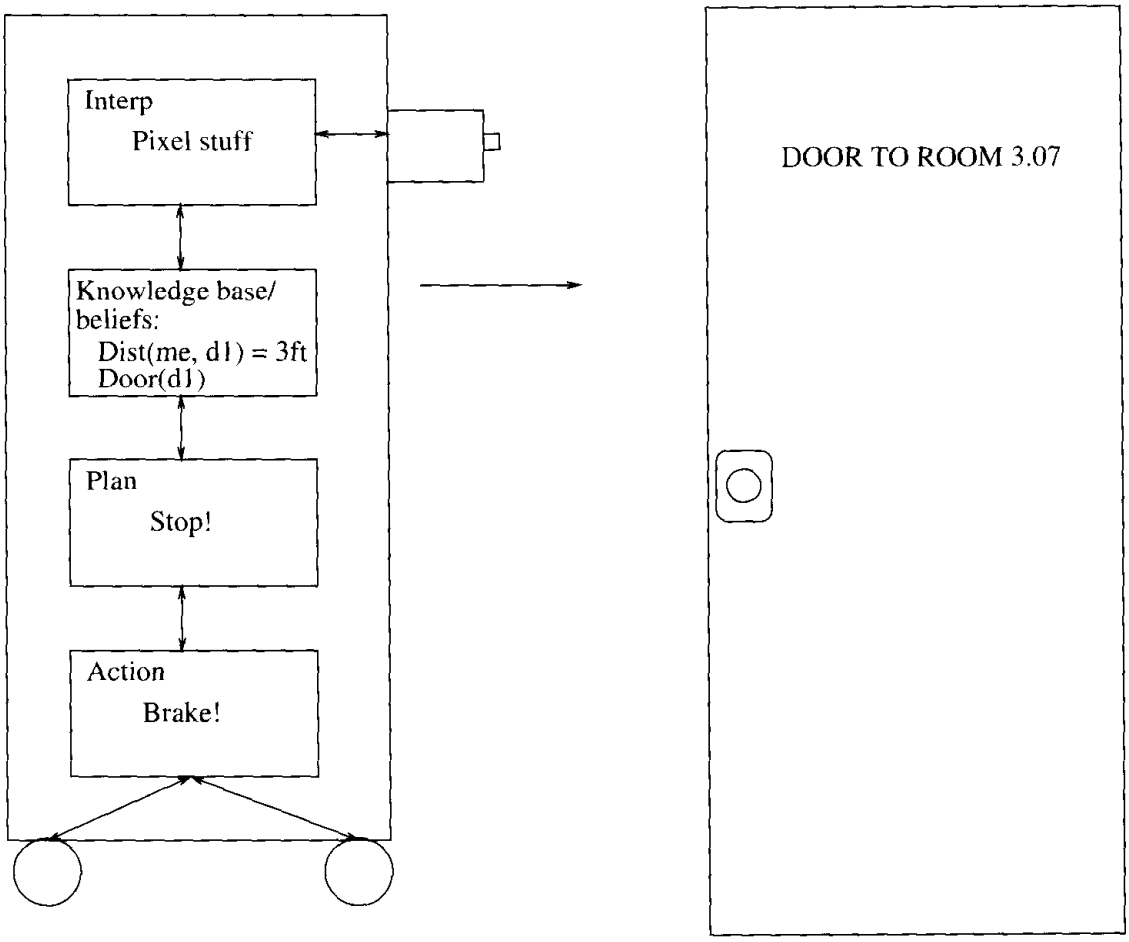

Figure 3.1 A robotic agent that contains a symbolic description of its environment.

> The agent's information about the world is contained in a data structure which for historical reasons is labelled as a 'knowledge base' in Figure 3.1.

In order to build RALPH, it seems we must solve two key problems.

(1) **The transduction problem.** The problem of translating the real world into an accurate, adequate symbolic description of the world, in time for that description to be useful.

(2) **The representation/reasoning problem.** The problem of representing information symbolically, and getting agents to manipulate/reason with it, in time for the results to be useful.

The former problem has led to work on vision, speech understanding, learning, etc. The latter has led to work on knowledge representation, automated reasoning, automated planning, etc. Despite the immense volume of work that the problems have generated, many people would argue that neither problem is anywhere near solved. Even seemingly trivial problems, such as common sense reasoning, have turned out to be extremely difficult.

Despite these problems, 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 so as to optimize some performance measure (see Chapter 2). 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.

3.1 Agents as Theorem Provers

To see how such an idea might work, we shall develop a simple model of logic-based agents, which we shall call *deliberate* agents (Genesereth and Nilsson, 1987, Chapter 13). 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. 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 $\Delta \vdash_\rho \varphi$ if the formula φ can be proved from the database Δ using only the deduction rules ρ. An agent's

```
Function: Action Selection as Theorem Proving
1.  function action(Δ : D) returns an action Ac
2.  begin
3.      for each α ∈ Ac do
4.          if Δ ⊢ρ Do(α) then
5.              return α
6.          end-if
7.      end-for
8.      for each α ∈ Ac do
9.          if Δ ⊬ρ ¬Do(α) then
10.             return α
11.         end-if
12.     end-for
13.     return null
14. end function action
```

Figure 3.2 Action selection as theorem-proving.

perception function *see* remains unchanged:

$$see : S \rightarrow Per.$$

Similarly, our *next* function has the form

$$next : D \times Per \rightarrow D.$$

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

$$action : D \rightarrow Ac,$$

is defined in terms of its deduction rules. The pseudo-code definition of this function is given in Figure 3.2.

The idea is that the agent programmer will encode the deduction rules ρ and database Δ in such a way that if a formula $Do(\alpha)$ can be derived, where α is a term that denotes an action, then α 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 α in turn, and attempts to prove the formula $Do(\alpha)$ from its database (passed as a parameter to the function) using its deduction rules ρ. If the agent succeeds in proving $Do(\alpha)$, then α is returned as the action to be performed.

What happens if the agent fails to prove $Do(\alpha)$, for all actions $a \in Ac$? 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 $a \in Ac$ such that $\neg Do(\alpha)$ cannot be derived from

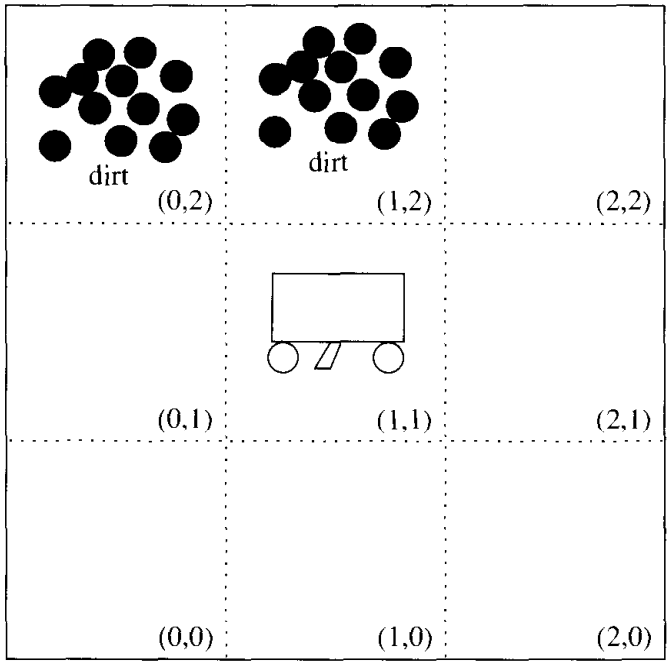

Figure 3.3 Vacuum world.

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 Russell and Norvig (1995, p. 51)). 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.

To summarize, 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 3.3 for an illustration of the vacuum world.

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

$$In(x,y) \quad \text{agent is at } (x,y), \tag{3.1}$$

$$Dirt(x,y) \quad \text{there is dirt at } (x,y), \tag{3.2}$$

$$Facing(d) \quad \text{the agent is facing direction } d. \tag{3.3}$$

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

$$new : D \times Per \rightarrow D.$$

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

$$\varphi(\ldots) \longrightarrow \psi(\ldots),$$

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):

$$In(x,y) \wedge Dirt(x,y) \longrightarrow Do(suck). \tag{3.4}$$

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:

$$In(0,0) \wedge Facing(north) \wedge \neg Dirt(0,0) \rightarrow Do(forward), \quad (3.5)$$

$$In(0,1) \wedge Facing(north) \wedge \neg Dirt(0,1) \rightarrow Do(forward), \quad (3.6)$$

$$In(0,2) \wedge Facing(north) \wedge \neg Dirt(0,2) \rightarrow Do(turn), \quad (3.7)$$

$$In(0,2) \wedge Facing(east) \rightarrow Do(forward). \quad (3.8)$$

Notice that in each rule, we must explicitly check whether the antecedent of rule (3.4) 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 $Do(\alpha)$, then α is an *optimal* action – that is, α 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 $\Delta_1 \vdash_\rho Do(\alpha)$ for some $\alpha \in Ac$, and so α 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 α 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 that 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 artefact 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 environ-

ment 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 realized. 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 summarize, 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.

3.2 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 idea of programming computer systems in terms of mental states was articulated in Shoham (1993).

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*, a set of initial *commitments*, 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 in Agent0 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 (in Chapter 8, we will see that this style of communication is very common in multiagent systems). 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 3.4).

(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.

(3) Goto (1).

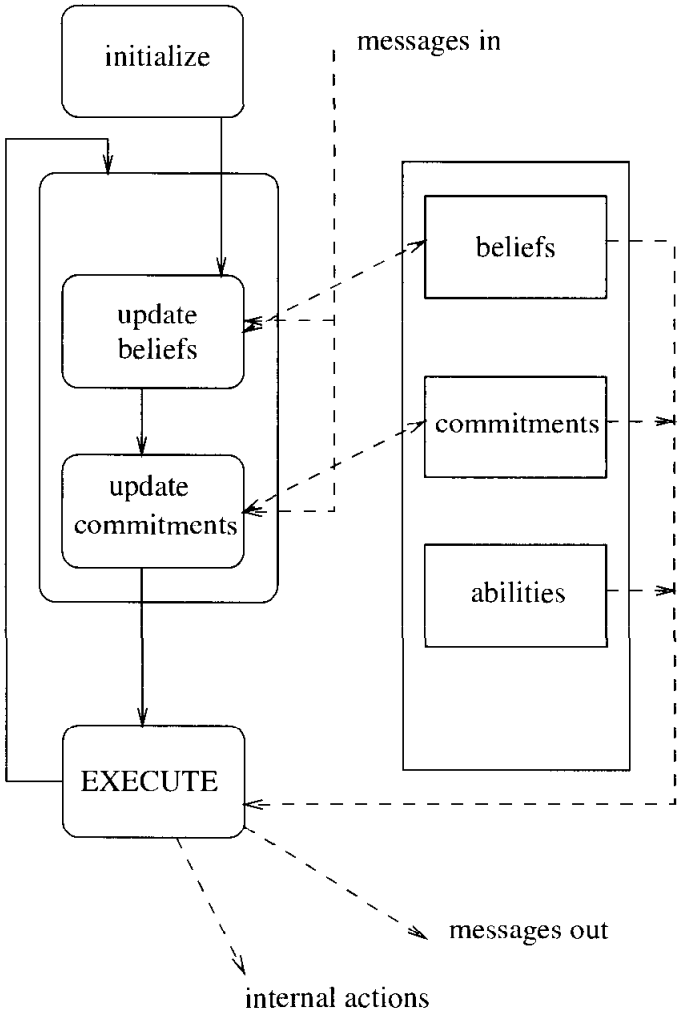

Figure 3.4 The flow of control in Agent0.

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. But it does at least give a feel for how such systems might be built.

3.3 Concurrent MetateM

The Concurrent MetateM language developed by Michael Fisher is based on the *direct execution* of logical formulae. In this sense, it comes very close to the 'ideal' of the agents as deductive theorem provers (Fisher, 1994). 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 (Barringer *et al.*, 1989).

Agents in Concurrent MetateM are concurrently executing entities, able to communicate with each other through broadcast message passing. Each Concurrent MetateM agent has two main components:

- an *interface*, which defines how the agent may interact with its environment (i.e. other agents); and
- a *computational engine*, which defines how the agent will act - in Concurrent MetateM, the approach used is based on the MetateM paradigm of executable temporal logic (Barringer *et al.*, 1989).

An agent interface consists of three components:

- a unique *agent identifier* (or just agent id), which names the agent;
- a set of symbols defining which messages will be accepted by the agent - these are termed *environment propositions*; and
- a set of symbols defining messages that the agent may send - these are termed *component propositions*.

For example, the interface definition of a 'stack' agent might be

$$stack(pop, push)[popped, full].$$

Here, $stack$ is the agent id that names the agent, $\{pop, push\}$ is the set of environment propositions, and $\{popped, full\}$ is the set of component propositions. The intuition is that, whenever a message headed by the symbol pop is broadcast, the $stack$ agent will *accept* the message; we describe what this means below. If a message is broadcast that is not declared in the $stack$ agent's interface, then $stack$ ignores it. Similarly, the only messages that can be sent by the $stack$ agent are headed by the symbols $popped$ and $full$.

The computational engine of each agent in Concurrent MetateM is based on the MetateM paradigm of executable temporal logics (Barringer *et al.*, 1989). The idea is to directly execute an agent specification, where this specification is given as a set of *program rules*, which are temporal logic formulae of the form:

$$\text{antecedent about past} \quad \Rightarrow \quad \text{consequent about present and future.}$$

The antecedent is a temporal logic formula referring to the past, whereas the consequent is a temporal logic formula referring to the present and future. The intuitive interpretation of such a rule is 'on the basis of the past, construct the future', which gives rise to the name of the paradigm: *declarative past and imperative future* (Gabbay, 1989). The rules that define an agent's behaviour can be animated by directly executing the temporal specification under a suitable operational model (Fisher, 1995).

Table 3.1 Temporal connectives for Concurrent MetateM rules.

Operator	Meaning
$\bigcirc \varphi$	φ is true 'tomorrow'
$\mathbf{O} \varphi$	φ was true 'yesterday'
$\Diamond \varphi$	at some time in the future, φ
$\Box \varphi$	always in the future, φ
$\blacklozenge \varphi$	at some time in the past, φ
$\blacksquare \varphi$	always in the past, φ
$\varphi \mathcal{U} \psi$	φ will be true until ψ
$\varphi S \psi$	φ has been true since ψ
$\varphi \mathcal{W} \psi$	φ is true unless ψ
$\varphi \mathcal{Z} \psi$	φ is true zince ψ

To make the discussion more concrete, we introduce a propositional temporal logic, called Propositional MetateM Logic (PML), in which the temporal rules that are used to specify an agent's behaviour will be given. (A complete definition of PML is given in Barringer *et al.* (1989).) PML is essentially classical propositional logic augmented by a set of modal connectives for referring to the *temporal ordering* of events.

The meaning of the temporal connectives is quite straightforward: see Table 3.1 for a summary. Let φ and ψ be formulae of PML, then: $\bigcirc \varphi$ is satisfied at the current moment in time (i.e. now) if φ is satisfied at the next moment in time; $\Diamond \varphi$ is satisfied now if φ is satisfied either now or at some future moment in time; $\Box \varphi$ is satisfied now if φ is satisfied now and at all future moments; $\varphi \mathcal{U} \psi$ is satisfied now if ψ is satisfied at some future moment, and φ is satisfied until then - $\mathcal{W}$ is a binary connective similar to $\mathcal{U}$, allowing for the possibility that the second argument might never be satisfied.

The past-time connectives have similar meanings: $\mathbf{O} \varphi$ and $\bullet \varphi$ are satisfied now if φ was satisfied at the previous moment in time - the difference between them is that, since the model of time underlying the logic is bounded in the past, the beginning of time is treated as a special case in that, when interpreted at the beginning of time, $\mathbf{O} \varphi$ cannot be satisfied, whereas $\bullet \varphi$ will always be satisfied, regardless of φ; $\blacklozenge \varphi$ is satisfied now if φ was satisfied at some previous moment in time; $\blacksquare \varphi$ is satisfied now if φ was satisfied at all previous moments in time; $\varphi S \psi$ is satisfied now if ψ was satisfied at some previous moment in time, and φ has been satisfied since then - $\mathcal{Z}$ is similar, but allows for the possibility that the second argument was never satisfied; finally, a nullary temporal operator can be defined, which is satisfied only at the beginning of time - this useful operator is called 'start'.

To illustrate the use of these temporal connectives, consider the following examples:

$$\Box important(agents)$$

means 'it is now, and will always be true that agents are important'.

$$\Diamond important(Janine)$$

means 'sometime in the future, Janine will be important'.

$$(\neg friends(us))\ \mathcal{U}\ apologize(you)$$

means 'we are not friends until you apologize'. And, finally,

$$\bigcirc apologize(you)$$

means 'tomorrow (in the next state), you apologize'.

The actual execution of an agent in Concurrent MetateM is, superficially at least, very simple to understand. Each agent obeys a cycle of trying to match the past-time antecedents of its rules against a *history*, and executing the consequents of those rules that 'fire'. More precisely, the computational engine for an agent continually executes the following cycle.

(1) Update the *history* of the agent by receiving messages (i.e. environment propositions) from other agents and adding them to its history.

(2) Check which rules *fire*, by comparing past-time antecedents of each rule against the current history to see which are satisfied.

(3) *Jointly execute* the fired rules together with any commitments carried over from previous cycles.

This involves first collecting together consequents of newly fired rules with old commitments - these become the *current constraints*. Now attempt to create the next state while satisfying these constraints. As the current constraints are represented by a disjunctive formula, the agent will have to choose between a number of execution possibilities.

Note that it may not be possible to satisfy *all* the relevant commitments on the current cycle, in which case unsatisfied commitments are carried over to the next cycle.

(4) Goto (1).

Clearly, step (3) is the heart of the execution process. Making the wrong choice at this step may mean that the agent specification cannot subsequently be satisfied.

When a proposition in an agent becomes *true*, it is compared against that agent's interface (see above); if it is one of the agent's *component propositions*, then that proposition is broadcast as a message to all other agents. On receipt of a message, each agent attempts to match the proposition against the environment propositions in their interface. If there is a match, then they add the proposition to their history.

$$
\begin{array}{rcl}
rp(ask1, ask2)[give1, give2]: & & \\
\bullet ask1 & \Rightarrow & \Diamond give1; \\
\bullet ask2 & \Rightarrow & \Diamond give2; \\
\text{start} & \Rightarrow & \Box\neg(give1 \wedge give2). \\
 & & \\
rc1(give1)[ask1]: & & \\
\text{start} & \Rightarrow & ask1; \\
\bullet ask1 & \Rightarrow & ask1. \\
 & & \\
rc2(ask1, give2)[ask2]: & & \\
\bullet (ask1 \wedge \neg ask2) & \Rightarrow & ask2.
\end{array}
$$

Figure 3.5 A simple Concurrent MetateM system.

Time	Agent		
	rp	$rc1$	$rc2$
0.		$ask1$	
1.	$ask1$	$ask1$	$ask2$
2.	$ask1, ask2, give1$	$ask1$	
3.	$ask1, give2$	$ask1, give1$	$ask2$
4.	$ask1, ask2, give1$	$ask1$	$give2$
5.	...	...	...

Figure 3.6 An example run of Concurrent MetateM.

Figure 3.5 shows a simple system containing three agents: rp, $rc1$, and $rc2$. The agent rp is a 'resource producer': it can '$give$' to only one agent at a time, and will commit to eventually $give$ to any agent that asks. Agent rp will only accept messages $ask1$ and $ask2$, and can only send $give1$ and $give2$ messages. The interface of agent $rc1$ states that it will only accept $give1$ messages, and can only send $ask1$ messages. The rules for agent $rc1$ ensure that an $ask1$ message is sent on every cycle - this is because start is satisfied at the beginning of time, thus firing the first rule, so $\bullet ask1$ will be satisfied on the next cycle, thus firing the second rule, and so on. Thus $rc1$ asks for the resource on every cycle, using an $ask1$ message. The interface for agent $rc2$ states that it will accept both $ask1$ and $give2$ messages, and can send $ask2$ messages. The single rule for agent $rc2$ ensures that an $ask2$ message is sent on every cycle where, on its previous cycle, it did not send an $ask2$ message, but received an $ask1$ message (from agent $rc1$). Figure 3.6 shows a fragment of an example run of the system in Figure 3.5.

$$
\begin{array}{rcl}
SnowWhite(ask)[give]: & & \\
\bullet ask(x) & \Rightarrow & \Diamond give(x) \\
give(x) \wedge give(y) & \Rightarrow & (x = y) \\
\\
eager(give)[ask]: & & \\
\text{start} & \Rightarrow & ask(eager) \\
\bullet give(eager) & \Rightarrow & ask(eager) \\
\\
greedy(give)[ask]: & & \\
\text{start} & \Rightarrow & \Box ask(greedy) \\
\\
courteous(give)[ask]: & & \\
((\neg ask(courteous)\ S\ give(eager)) \wedge & & \\
(\neg ask(courteous)\ S\ give(greedy))) & \Rightarrow & ask(courteous) \\
\\
shy(give)[ask]: & & \\
\text{start} & \Rightarrow & \Diamond ask(shy) \\
\bullet ask(x) & \Rightarrow & \neg ask(shy) \\
\bullet give(shy) & \Rightarrow & \Diamond ask(shy)
\end{array}
$$

Figure 3.7 Snow White in Concurrent MetateM.

Exercises

(1) **[Level 2.]** (The following few questions refer to the vacuum-world example.)

Give the full definition (using pseudo-code if desired) of the *new* function, which defines the predicates to add to the agent's database.

(2) **[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?

(3) **[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 with 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?

(4) **[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).

(5) **[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 generalize the rules, encoding a more general decision-making mechanism.

(6) **[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 with one implemented in a traditional (imperative) programming language.

(7) **[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?

(8) **[Level 2.]**

Consider the Concurrent MetateM program in Figure 3.7. Explain the behaviour of the agents in this system.

(9) **[Level 4.]**

Extend the Concurrent MetateM language by operators for referring to the beliefs and commitments of other agents, in the style of Shoham's Agent0.

(10) **[Level 4.]**

Give a formal semantics to Agent0 and Concurrent MetateM.

5

Reactive and Hybrid Agents

The many 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 characterize 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 used in this

chapter – *reactive* (because such systems are often perceived as simply reacting to an environment, without reasoning about it).

5.1 Brooks and the Subsumption Architecture

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. Brooks has propounded three key theses that have guided his work as follows (Brooks, 1991b; Brooks, 1991a).

(1) Intelligent behaviour can be generated *without* explicit representations of the kind that symbolic AI proposes.

(2) Intelligent behaviour can be generated *without* explicit abstract reasoning of the kind that symbolic AI proposes.

(3) Intelligence is an *emergent* property of certain complex systems.

Brooks also identifies two key ideas that have informed his research.

(1) Situatedness and embodiment. 'Real' intelligence is situated in the world, not in disembodied systems such as theorem provers or expert systems.

(2) Intelligence and emergence. 'Intelligent' behaviour arises as a result of an agent's interaction with its environment. Also, intelligence is 'in the eye of the beholder' – it is not an innate, isolated property.

These ideas were made concrete in the subsumption architecture. There are two defining characteristics of the subsumption architecture. The first is that an agent's decision-making is realized through a set of *task-accomplishing behaviours*; each behaviour may be thought 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's 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

$$\text{situation} \longrightarrow \text{action},$$

which simply 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

```
Function: Action Selection in the Subsumption Architecture
1.  function action(p : P) : A
2.  var fired : ℘(R)
3.  var selected : A
4.  begin
5.      fired ← {(c,a) | (c,a) ∈ R and p ∈ c}
6.      for each (c,a) ∈ fired do
7.          if ¬(∃(c',a') ∈ fired such that (c',a') ≺ (c,a)) then
8.              return a
9.          end-if
10.     end-for
11.     return null
12. end function action
```

Figure 5.1 Action Selection in the subsumption architecture.

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 realized 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 $s \in S$ if and only if $see(s) \in c$. 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 strict 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 as shown in Figure 5.1.

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 *much* 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 microseconds. Perhaps more than anything else, this computational simplicity is the strength of the subsumption architecture.

Steels's Mars explorer experiments

We will see how subsumption architecture agents were built for the following scenario (this example is adapted from Steels (1990)).

> 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 a mother ship 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 (Steels, 1990). 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 mother ship lies, the mother ship generates a radio signal. Now this signal will obviously weaken as distance from the source increases – to find the direction of the mother ship, 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 be represented in the rule:

if detect an obstacle *then* change direction. (5.1)

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; (5.2)

if carrying samples and *not* at the base *then* travel up gradient. (5.3)

Behaviour (5.3) 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. (5.4)

The final behaviour ensures that an agent with 'nothing better to do' will explore randomly:

if true *then* move randomly. (5.5)

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

$$(5.1) \prec (5.2) \prec (5.3) \prec (5.4) \prec (5.5).$$

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 ever 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 mother ship. 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 (5.1) 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; (5.6)

if carrying samples and *not* at the base
then drop 2 crumbs *and* travel up gradient. (5.7)

The behaviour (5.7) 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 (5.4) remains unchanged. However, an additional behaviour is required for dealing with crumbs:

if sense crumbs *then* pick up 1 crumb *and* travel down gradient. (5.8)

Finally, the random movement behaviour (5.5) remains unchanged. These behaviour are then arranged into the following subsumption hierarchy:

$$(5.1) \prec (5.6) \prec (5.7) \prec (5.4) \prec (5.8) \prec (5.5).$$

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).

Agre and Chapman – PENGI

At about the same time as Brooks was describing his first results with the subsumption architecture, Chapman was completing his Master's thesis, in which

he reported the theoretical difficulties with planning described above, and was coming to similar conclusions about the inadequacies of the symbolic AI model himself. Together with his co-worker Agre, he began to explore alternatives to the AI planning paradigm (Chapman and Agre, 1986).

Agre observed that most everyday activity is 'routine' in the sense that it requires little – if any – new abstract reasoning. Most tasks, once learned, can be accomplished in a routine way, with little variation. Agre proposed that an efficient agent architecture could be based on the idea of 'running arguments'. Crudely, the idea is that as most decisions are routine, they can be encoded into a low-level structure (such as a digital circuit), which only needs periodic updating, perhaps to handle new kinds of problems. His approach was illustrated with the celebrated PENGI system (Agre and Chapman, 1987). PENGI is a simulated computer game, with the central character controlled using a scheme such as that outlined above.

Rosenschein and Kaelbling – situated automata

Another sophisticated approach is that of Rosenschein and Kaelbling (see Rosenschein, 1985; Rosenschein and Kaelbling, 1986; Kaelbling and Rosenschein, 1990; Kaelbling, 1991). They observed that just because an agent is conceptualized in logical terms, it need not be implemented as a theorem prover. In their *situated automata* paradigm, an agent is specified in declarative terms. This specification is then compiled down to a digital machine, which satisfies the declarative specification. This digital machine can operate in a provably time-bounded fashion; it does not do any symbol manipulation, and in fact no symbolic expressions are represented in the machine at all. The logic used to specify an agent is essentially a logic of knowledge:

> [An agent] x is said to carry the information that p in world state s, written $s \models K(x, p)$, if for all world states in which x has the same value as it does in s, the proposition p is true.
>
> (Kaelbling and Rosenschein, 1990, p. 36)

An agent is specified in terms of two components: perception and action. Two programs are then used to synthesize agents: RULER is used to specify the perception component of an agent; GAPPS is used to specify the action component.

RULER takes as its input three components as follows.

> [A] specification of the semantics of the [agent's] inputs ('whenever bit 1 is on, it is raining'); a set of static facts ('whenever it is raining, the ground is wet'); and a specification of the state transitions of the world ('if the ground is wet, it stays wet until the sun comes out'). The programmer then specifies the desired semantics for the output ('if this bit is on, the ground is wet'), and the compiler. . .[synthesizes] a circuit

> whose output will have the correct semantics. ... All that declarative 'knowledge' has been reduced to a very simple circuit.
>
> (Kaelbling, 1991, p. 86)

The GAPPS program takes as its input a set of *goal reduction rules* (essentially rules that encode information about how goals can be achieved) and a top level goal, and generates a program that can be translated into a digital circuit in order to realize the goal. Once again, the generated circuit does not represent or manipulate symbolic expressions; all symbolic manipulation is done at compile time.

The situated automata paradigm has attracted much interest, as it appears to combine the best elements of both reactive and symbolic declarative systems. However, at the time of writing, the theoretical limitations of the approach are not well understood; there are similarities with the automatic synthesis of programs from temporal logic specifications, a complex area of much ongoing work in mainstream computer science (see the comments in Emerson (1990)).

Maes - agent network architecture

Pattie Maes has developed an agent architecture in which an agent is defined as a set of *competence modules* (Maes, 1989, 1990b, 1991). These modules loosely resemble the behaviours of Brooks's subsumption architecture (above). Each module is specified by the designer in terms of preconditions and postconditions (rather like STRIPS operators), and an *activation level*, which gives a real-valued indication of the *relevance* of the module in a particular situation. The higher the activation level of a module, the more likely it is that this module will influence the behaviour of the agent. Once specified, a set of competence modules is compiled into a *spreading activation network*, in which the modules are linked to one another in ways defined by their preconditions and postconditions. For example, if module a has postcondition φ, and module b has precondition φ, then a and b are connected by a *successor* link. Other types of link include predecessor links and conflicter links. When an agent is executing, various modules may become more active in given situations, and may be executed. The result of execution may be a command to an effector unit, or perhaps the increase in activation level of a successor module.

There are obvious similarities between the agent network architecture and neural network architectures. Perhaps the key difference is that it is difficult to say what the meaning of a node in a neural net is; it only has a meaning in the context of the net itself. Since competence modules are defined in declarative terms, however, it is very much easier to say what their meaning is.

5.2 The Limitations of Reactive Agents

There are obvious advantages to reactive approaches such as Brooks's subsumption architecture: simplicity, economy, computational tractability, robust-

ness 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 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.
- One 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 fulfil 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 than 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.