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.

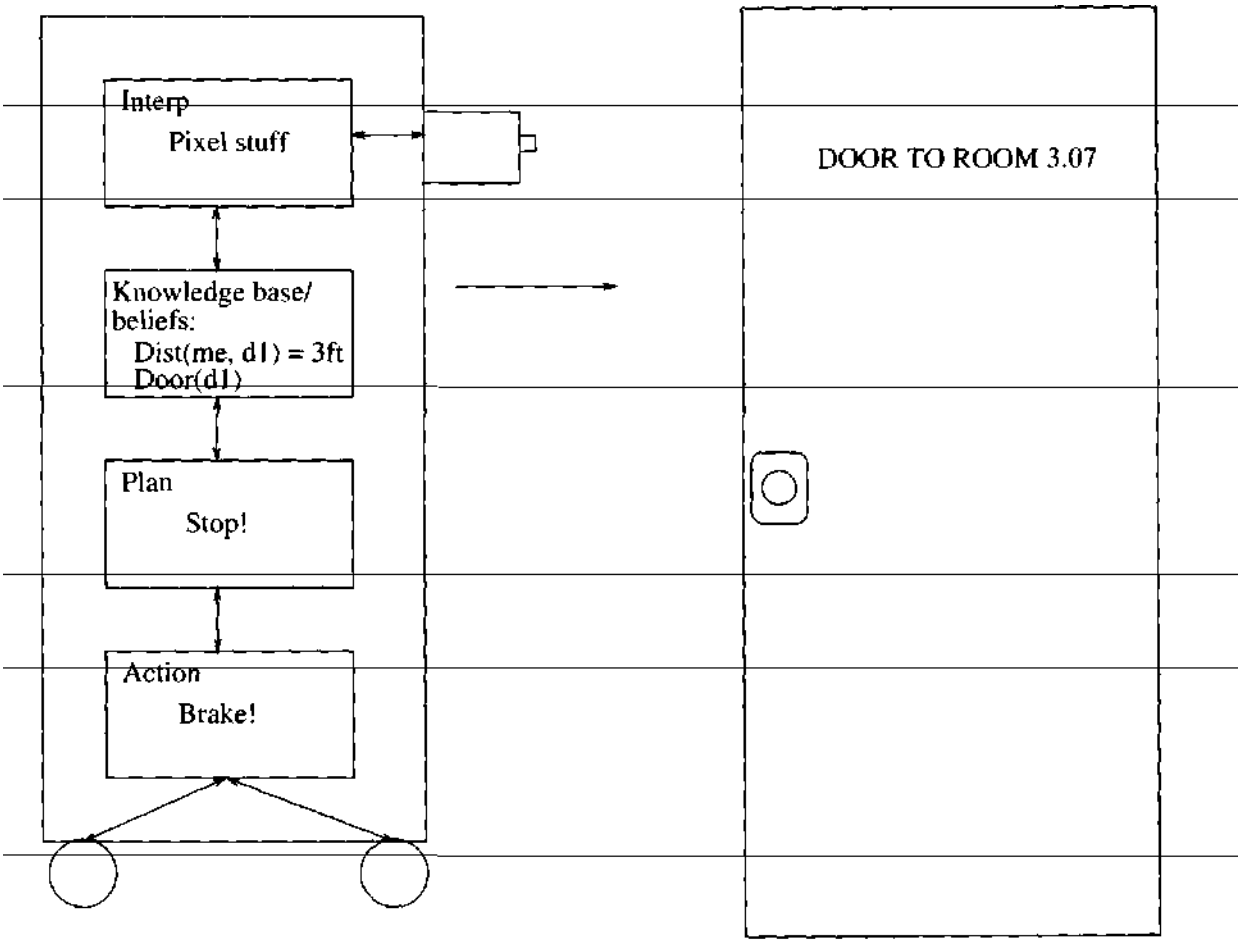

Figure 3.1 A robotic agent that contains a symbolic description of its environment.

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 I be the set of sentences of classical first-order logic, and let $D = p(L)$ 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 $A, \Delta_1, \ldots$ for members of D. An agent's decision-making process is modelled through a set of *deduction rules, p.* These are simply rules of inference for the logic. We write $A \vdash_\rho q p$ if the formula φ can be proved from the database A using only the deduction rules p. 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 p 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 p. 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 G AC such that $\neg Do(\alpha)$ cannot be derived from

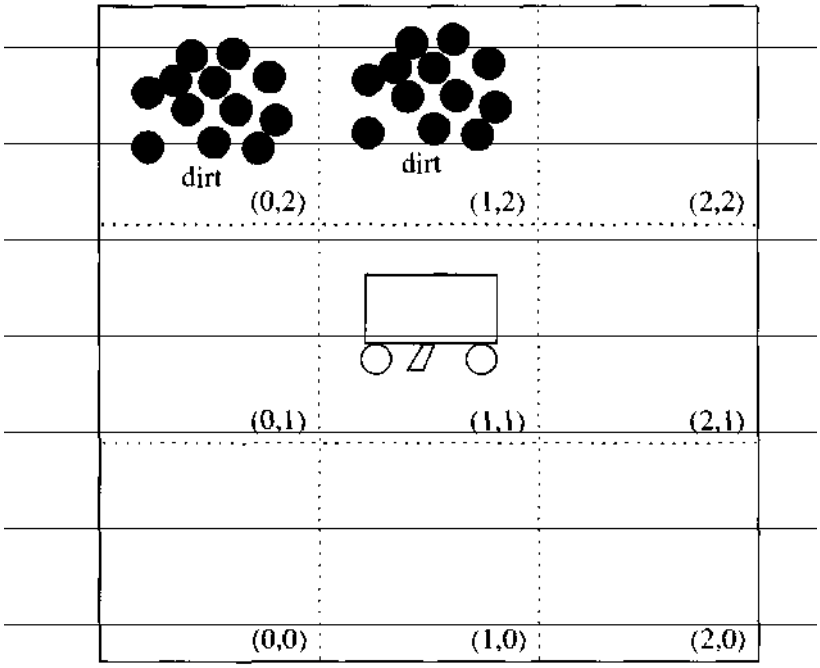

Figure 3.3 Vacuum world.

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 x 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(A)* to denote the set of 'old' information in a database, which we want the update function *next* to remove:

$$old(A) = \{P(t_1,\ldots,t_n)\ p \in \{In, Dirt, Facing\} \text{ and } P(t_1,\ldots t_n) \in A\}.$$

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(...),* describing the new position of the agent, *Facing(...)* describing the orientation of the agent, and *Dirt{...)* if dirt has been detected at the new position.) Given the *new* and *old* functions, the *next* function is defined as follows:

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

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{...)* 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 p such that for any database A, 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 A. Then imagine we start running our agent. At time t\, the agent has generated some database Δ_1, and begins to apply its rules p 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\ 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 objecl 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 AGENTO 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 ietermine 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 AgentO 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 AgentO 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).

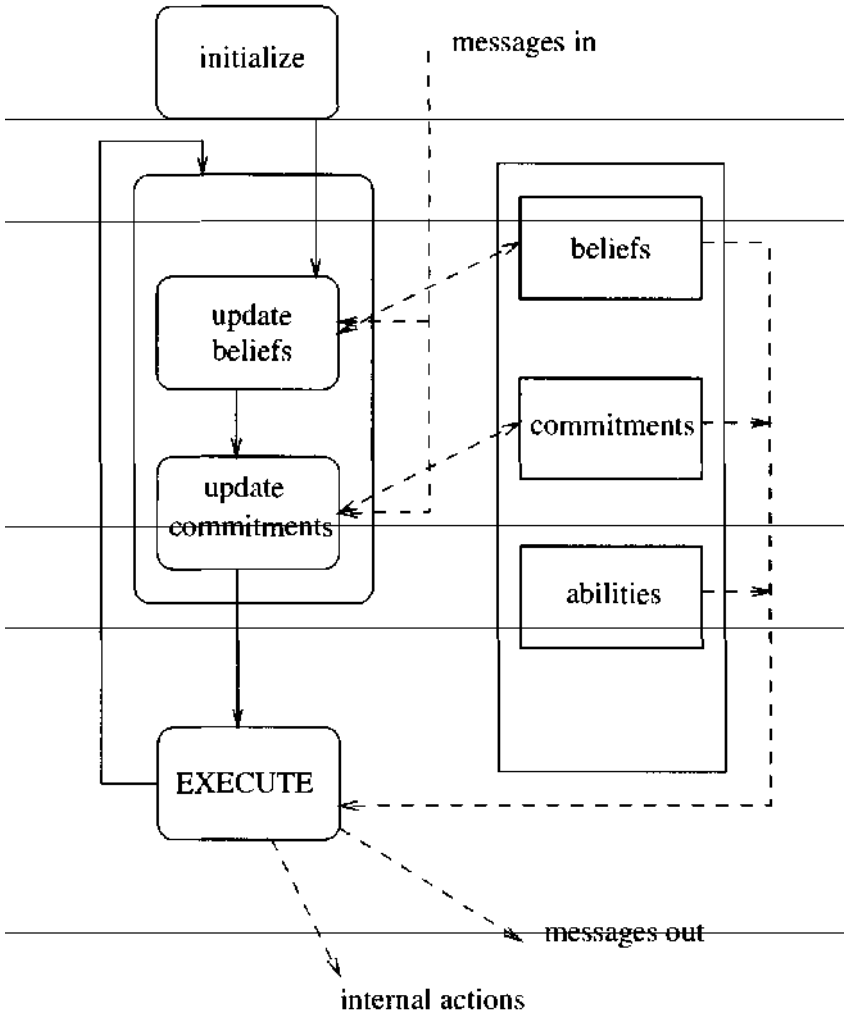

Figure 3.4 The flow of control in AgentO.

It should be clear how more complex agent behaviours can be designed and built in AgentO. 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:

antecedent about past => consequent about present and future.

ntecedent 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'
$\bullet \varphi$	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
$\varphi Z \psi$	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 *el al.* (1989).) PML is essentially classical proposilional 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 qp and ψ be formulae of PML, then: $\bigcirc qp$ is satisfied at the current moment in time (i.e. now) if qp is satisfied at the next moment in time; $\Diamond \varphi$ is satisfied now if qp is satisfied either now or at some future moment in time; $\Box \varphi$ is satisfied now if qp is satisfied now and at all future moments; $qp \mathcal{U} \psi$ is satisfied now if ψ is satisfied at some future moment, and qp 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: $\bullet qp$ and $\cdot qp$ are satisfied now if qp 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, $\bullet qp$ cannot be satisfied, whereas $\# qp$ will always be satisfied, regardless of qp; $\blacklozenge qp$ is satisfied now if qp was satisfied at some previous moment in time; $\blacksquare \varphi$ is satisfied now if qp 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 qp has been satisfied since then - 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.

$$rp(ask1, ask2)[give1, give2]:$$
$$\bullet ask1 \Rightarrow \Diamond give1;$$
$$\bullet ask2 \Rightarrow \Diamond give2;$$
$$\text{start} \Rightarrow \Box\neg (give1 \wedge give2).$$

$$rc1(give1)[ask1]:$$
$$\text{start} \Rightarrow ask1;$$
$$\bullet ask1 \Rightarrow ask1.$$

$$rc2(ask1, give2)[ask2]:$$
$$\bullet (ask1 \text{ A } \neg ask2) \Rightarrow ask2.$$

Figure 3.5 A simple Concurrent MetateM system.

Time	Agent		
	rp	*rcl*	*rc2*
0.		*ask*1	
1.	*ask*1		*ask?*
2.	*ask*1, *ask*2, *give*1	*askl*	
3.	*ask*1, *give*2	*askl,givel*	*askl*
4.	*ask*1, *ask*2, *give*1	*askl*	*give2*
5.			

Figure 3.6 An example run of Concurrent MetateM.

Figure 3.5 shows a simple system containing three agents: *rp, rcl,* and *rc2.* The agent *rp* is a 'resource producer': it ran '*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 *askl* and *ask2,* and can only send *give*1and *give2* messages. The interface of agent *rcl* states that it will only accept *give*1messages, and can only send *askl* messages. The rules for agent *rcl* ensure that an *askl* 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 *rcl* asks for the resource on every cycle, using an *askl* message. The interface for agent *rc2* states that it will accept both *askl* 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 *askl* message (from agent *rcl).* Figure 3.6 shows a fragment of an example run of the system **in** Figure 3.5.