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 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)* G 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: $\prec \subseteq RxR$. This relation is assumed to be a strict total ordering on *R* (i.e. it is transitive, irreflexive, and antisymmetric). We write $b_1 < b_2$ 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 therr as shown in Figure 5.1.

Thus action selection begins by first computing the set *fired* of all behaviours that fire (5). Then, eachbehaviour (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.

individual (non-cooperative) agents, the lowest-lev
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 e s 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:

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

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 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.

The behaviour (5./) 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'. [illegible]t as most decisions are routine, they ran 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 5, 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 *qp,* and module *b* has precondition *qp,* 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.

5.3 Hybrid Agents

Given the requirement that an agent be capable of reactive and proactive 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.

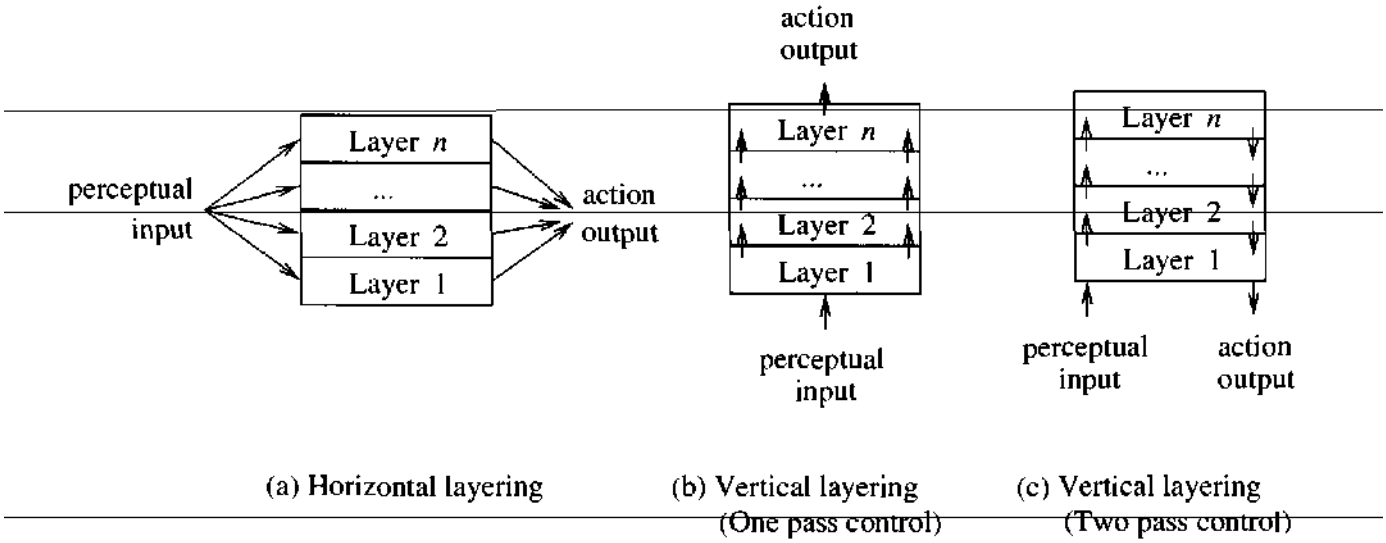

Figure 5.2 Information and control flows in three types of layered agent architecture. (Source: Muller *et al.* (1995, p. 263).)

Typically, there will be at least two layers, to deal with reactive and proactive behaviours, respectively. In principle, there is no reason why there should not be many more layers. It is useful to characterize such architectures in terms of the information and control flows within the layers. Broadly speaking, we can identify two types of control flow within layered architectures as follows (see Figure 5.2).

Horizontal layering. In horizontally layered architectures (Figure 5.2(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 (see parts (b) and (c) of Figure 5.2), sensory input and action output are each dealt with by at most one layer.

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 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.

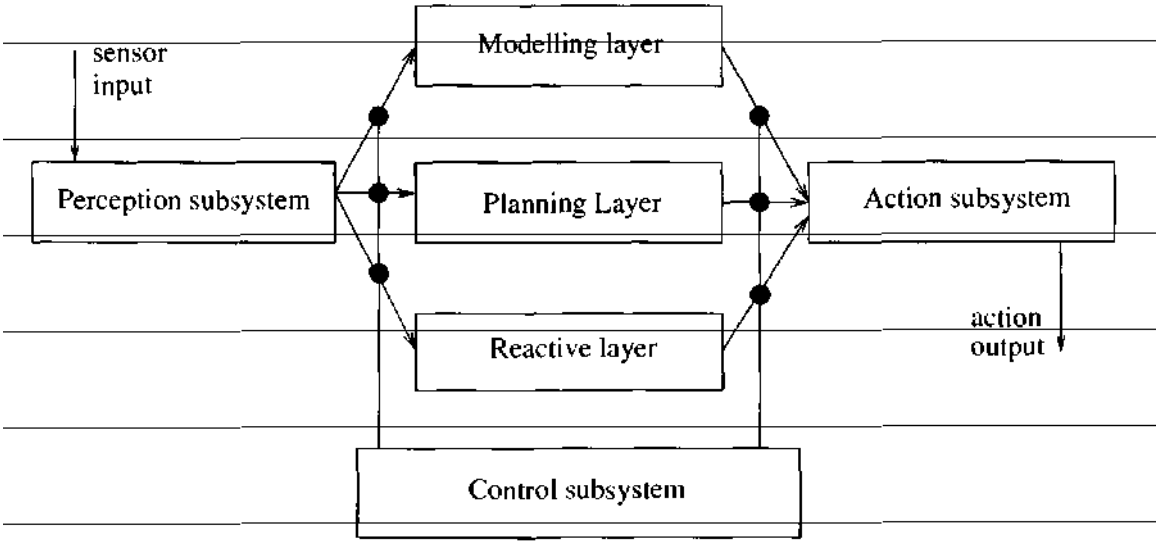

Figure 5.3 TouringMachines: a horizontally layered agent architecture.

These problems are partly alleviated in a vertically layered architecture. We can subdivide vertically layered architectures into *one-pass* architectures (Figure 5.2(b)) and *two-pass* architectures (Figure 5.2(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 nf two-pass vertically layered architectures and the way that organizations work, with information flowing up to the highest levels of the organization, 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 Miiller's InteRRaP. The former is an example of a horizontally layered architecture; the latter is a (two-pass) vertically layered architecture.

53.1 TouringMachines

The TouringMachines architecture is illustrated in Figure 5.3. As this figure shows, TouringMachines consists nf 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's subsumption architecture (see Section 5.1). 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 (Ferguson, 1992a, p. 59)):

```
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 proactive 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 Touring-Machines planning layer elaborates at run time in order to decide what to do (cf. the PRS architecture discussed in Chapter 4). 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 *modelling* layer represents the various entities in the world (including the agent itself, as well as other agents). The modelling 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

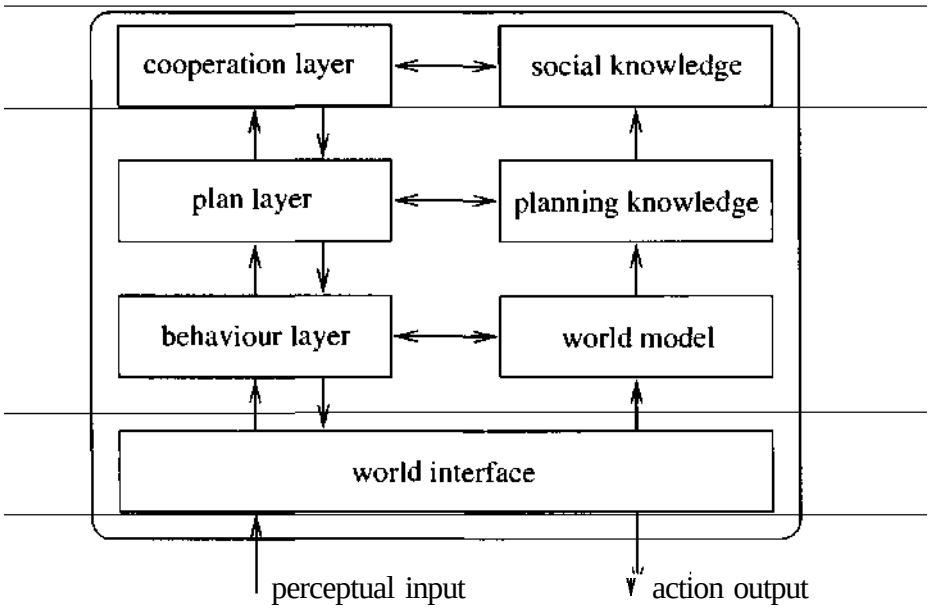

Figure 5.4 InteRRaP – a vertically layered two-pass agent architecture.

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 (Ferguson, 1995, p. ?07):

```
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.

5.3.2 InteRRaP

InteRRaP is an example of a vertically layered two-pass agent architecture - see Figure 5.4. As Figure 5.4 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. 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 interact 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 architecture. 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 chapter. However, it is worth noting that each layer implements two general functions. The first of these is a *situation recognition and goal activation* function. 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, proactive, social behaviour can be generated by the reactive, proactive, 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.