4.1 ARCHITECTURE FOR INTELLIGENT AGENTS

Agent architectures are software architectures for decision-making systems that are embedded in an environment. Four important classes of agents are:

1. *Logic-based agents* – Decision about what action to perform is made via logical deduction.
2. *Reactive agents* – Decision making is implemented in some form of direct mapping from situation to action.
3. *Belief-desire-intention agents* – Decision making depends upon the manipulation of data structures representing the beliefs, desires, and intentions of the agent.
4. *Layered architectures* – Decision making is realized via various software layers, each of which is more or less explicitly reasoning about the environment at different levels of abstraction.

4.1.1 Logic-Based Architectures

Decision about what action to perform is made via logical deduction. Traditional approach to building artificially intelligent systems suggests giving that system a *symbolic* representation of its environment and desired behavior. Symbolic representations are *logical formulae*, and syntactic manipulation corresponds to *logical deduction* or *theorem proving*.

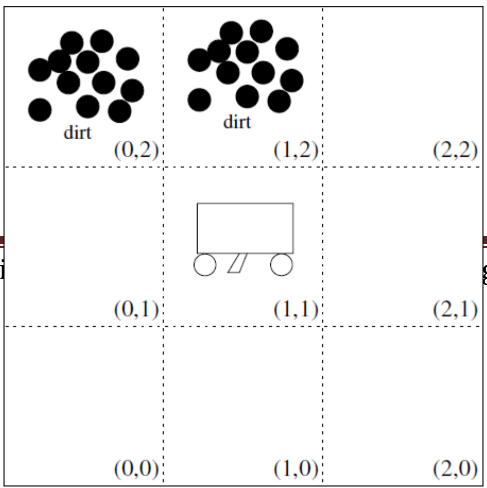
An example is *deliberate* agents, which assume to maintain an internal database of formulae of classical first-order predicate logic, which represents in a symbolic form the information they have about their environment. For example, an agent's belief database might contain formulae such as the following:

$$Open(valve)$$

$$Temperature(reactor, 321)$$

$$Pressure(tank, 28)$$

An agent's decision-making process is modeled through a set of *deduction rules.* The *action* takes as input the beliefs of the agent and deduction rules and returns as output either an action or else null when nothing is found.



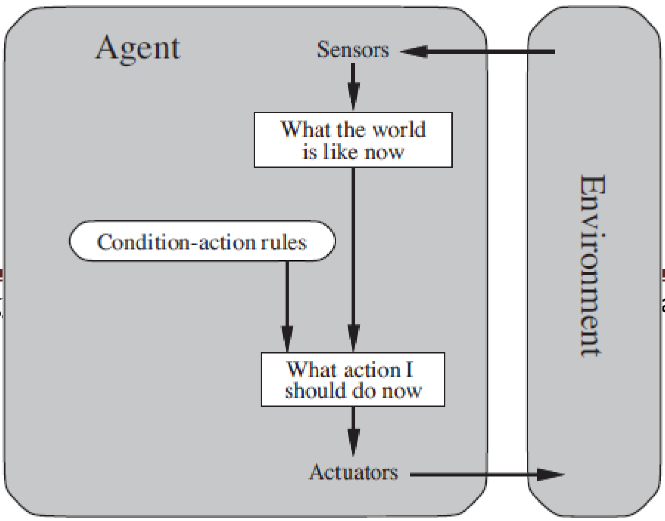


Figure 4.2 Vacuum Cleaning World **Figure 4.3 Simple Agent Architecture**

Let us consider a small example based on the vacuum cleaning world example. 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 (North, East, West and South) and turns right 90^{O}. The agent moves around a room, which is divided grid-like into a number of equally sized squares. We will assume that our agent does nothing but clean – it never leaves the room.

To summarize, our agent can receive a percept *dirt*, or *null*. It can perform any one of three possible actions: *forward*, *suck*, or *turn*. The robot will always move from (0,0) to (0,1) to (0,2) and then to (1,2), to (1,1), and so on. The goal is to traverse the room, continually searching for and removing dirt. First, make use of three simple *domain predicates*:

In(*x*, *y*) agent is at (*x*,*y*)

Dirt(*x*, *y*) there is dirt at (*x*,*y*)

Facing(*d*) the agent is facing direction *d*

Some of the rules that govern our agent's behavior are:

***In*(*x*, *y*)*Dirt*(*x*, *y*) −→ *Do*(*suck*)---(4.1)**

***In*(0,0)∧*Facing*(*north*)∧¬*Dirt*(0,0) −→ *Do*(*forward*)--------------------------------------(4.2)**

***In*(0,1)∧*Facing*(*north*)∧¬*Dirt*(0,1) −→ *Do*(*forward*)--------------------------------------(4.3)**

***In*(0,2)∧*Facing*(*north*)∧¬*Dirt*(0,2) −→ *Do*(*turn*)--(4.4)**

***In*(0,2)∧*Facing*(*east*) −→ *Do*(*forward*) ---(4.5)**

The problems with this vacuum cleaning world are:

1. 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.
2. Representing and reasoning ***temporal information.*** Temporal information is how a situation changes over time. Representing it turns out to be extraordinarily difficult.
3. The problems associated with ***representing and reasoning*** about complex, dynamic, possibly physical environments are also essentially unsolved.

4.1.2 Reactive Architectures (or) Subsumption Architecture

The *subsumption architecture* is arguably the best-known reactive agent architecture.

There are two defining characteristics of the subsumption architecture.

1. The first is a set of *task accomplishing behaviors*. Each behavior may be thought of as an individual action selection process, which continually takes perceptual input and maps it to an action to perform. These behaviors are implemented as rules of the form,

   situation −→ action.

2. The second is that *subsumption hierarchy* has behaviors arranged into *layers*. Many behaviors can ―fire‖ simultaneously. There must be a mechanism to choose between the different actions selected by these multiple actions. The lower a layer is, the higher is its priority, which represent more abstract behaviors. For example, in a mobile robot, it makes sense to give obstacle avoidance a high priority.

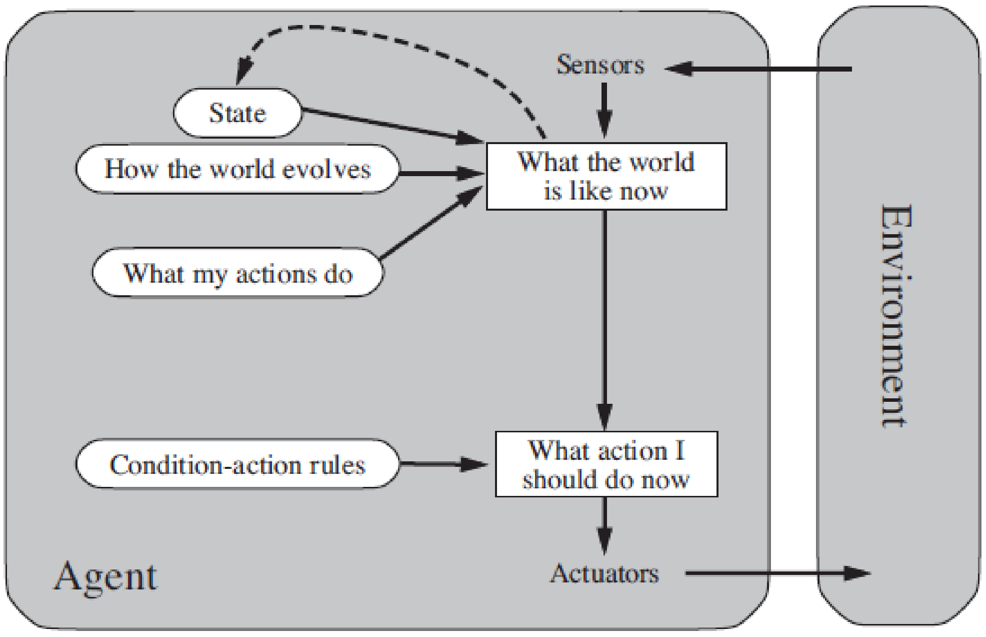


Figure 4.4 Subsumption Architecture

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

exchanging any communication.

The problem we are faced with is that of building agent control architecture for each vehicle, so that they will cooperate among themselves. The solution makes use of two mechanisms introduced by Steels. The first is a *gradient field*, the range of radio signal generated by mothership to find its location. The second is communication mechanism. Agents will carry ―radioactive crumbs,‖ which can be dropped, picked up, and detected by passing robots.

The lowest-level behavior is obstacle avoidance, which can be represented in the rule:

if detect an obstacle *then* change direction---**(4.6)**

Other behaviors ensures any samples carried by agents are dropped back at mothership.

if carrying samples *and* at the base *then* drop samples. -------------------------------**(4.7)**

if carrying samples and *not* at the base *then* travel up gradient. ---------------------**(4.8)**

if detect a sample *then* pick sample up ---**(4.9)**

if true *then* move randomly.---**(4.10)**

The precondition of 4.10 rule is thus assumed to always fire. These behaviors are arranged into the following hierarchy:

$$(4.6) \prec (4.7) \prec (4.8) \prec (4.9) \prec (4.10)$$

However, rule 4.8, determining action on carrying sample and not at base is modified as follows.

if carrying samples and *not* at the base *then* drop 2 crumbs *and* travel up gradient. ------ **(4.11)**

However, an additional behavior is required for dealing with crumbs.

if sense crumbs *then* pick up 1 crumb *and* travel down gradient. ----------------------------**(4.12)**

These behaviors are then arranged into the following subsumption hierarchy:

$(4.6) \prec (4.8) \prec (4.11) \prec (4.9) \prec (4.12) \prec (4.10)$

Advantages and Disadvantages of Reactive Architecture:

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

- Agents need sufficient information available in their *local* environment.
- It is difficult to see how decision making could take into account *non-local* information.
- Relationship between individual behaviors, environment, and overall behavior is not understandable.
- It is *much* harder to build agents that contain many layers. The dynamics of the interactions between the different behaviors become too complex to understand.

Markov Decision Processes (MDP)

Markov models were originally developed by the Russian mathematician **Andrei Markov** as models of stochastic process that is, dynamic processes whose behaviors are probabilistic. Markov models are used in operations research for modeling stochastic processes, in which a sequence of decisions must be made over time. The basic components of Markov models are as follows:

1. A state S, which represents every state that one could be in, within a defined world.
2. A model or transition function T; which is a function of the current state, the action taken and the state where we end up. This transition produces a certain probability of ending up in state S', starting from the state S and taking the action A.
3. Actions are things agent can do in a particular state.
4. A reward is a scaler value (v) for being in a state, tells usefulness of entering the state.

   There are two properties on which the Markovian process is based on:

1. Only the present matters; transition function only depends on the current state S and not any of the previous states. This property is called **Markov assumption.**

2. Things are stationary, therefore rules do no change over time.

Final goal of the MDP is to find a policy that can tell, for any state, which action to take. The optimal policy maximizes the long-term expected reward. A policy (decision rule) is a conditional plan that defines an action to perform for every possible state ($d : S \rightarrow A$).

Finding an optimal policy using **value iteration** proceeds in two steps. First, we compute the value function, v∗ : S→R, which gives the value v∗ of every state s ∈ S. The value v∗ is the expected reward that would be obtained from executing the optimal policy in that state. Now, given the value function v∗, we can then easily ―extract‖ the optimal

```
function $value\_iteration(S,A,p,r,\lambda)$ return optimal policy $v^*$
    initialize $v^*$ randomly
    repeat
        $v := v^*$
        for each $s \in S$ do
            $v^*(s) := \max_{a \in A}\left(r(a,s) + \lambda \sum_{s' \in S} p(s' \mid s,a)v(s')\right)$
        end-for
    until $v$ and $v^*$ are sufficiently close
    return $v^*$
end function $value\_iteration$
```

action.

Figure 1.4: The value iteration algorithm for Markov decision processes.

The idea is to iteratively approximate v∗ using two variables v (old) and v∗ (new). We continue to iterate until the difference between the old and new approximation is sufficiently small. ―Sufficiently close‖ means define some convergence threshold ε, and stop when the maximum difference between v and v∗ is ε. When ε = 0, we are requiring exact convergence.

$$d^*(s) = \arg\max_{a \in A}\left(r(s,a) + \lambda \sum_{s' \in S} p(s' \mid s,a)v^*(s')\right)$$

4.2.3Belief-Desire-Intention Architectures

Belief Desire Intention (BDI) architectures have their roots in the philosophical tradition of understanding practical reasoning — the process of deciding, moment by moment, which action to perform in the furtherance of our goals. Practical reasoning involves two important processes:

1. **Deliberation** - deciding what goals we want to achieve, and
2. **Means-ends reasoning** - how we are going to achieve these goals.

The decision process begins by trying to understand what are the options available. After generating this set of alternatives, you must choose between them, called **intentions**, and commit to some, called **future practical reasoning**.

4.2.3.1 Intention

Make a reasonable attempt to achieve the intention. Moreover, if a course of action fails to achieve the intention, then you would expect to try again – you would not expect to simply give up. This intention will constrain future practical reasoning. Intentions play a number of important roles in practical reasoning:

- Intentions drive means-ends reasoning.
- Intentions constrain future deliberation.
- Intentions persist.
- Intentions influence beliefs upon which future practical reasoning is based.

A key problem in the design of practical reasoning agents is that of achieving a good balance between these different concerns. Specifically, it seems clear that *an agent should at times drop some intentions.* From time to time, it is worth an agent to reconsider its intentions and at times stopping to reconsider its intentions.

Reconsideration has a cost, in terms of both time and computational resources. This presents a dilemma, essentially the problem of balancing proactive (goal-directed) and reactive (event-driven) behavior.

- **Proactive or Goal-directed behavior -** A **Bold agent** does not stop to reconsider sufficiently often will continue attempting to achieve its intentions even after it is clear that they cannot be achieved, or that there is no longer any reason for achieving them;

- **Reactive or Event-driven behavior –** A **cautious agent** constantly reconsiders its intentions may spend insufficient time actually working to achieve them, and hence runs the risk of never actually achieving them.

Let us investigate how bold agents (those that never stop to reconsider) and cautious agents (those that are constantly stopping to reconsider). The rate of world change is γ.

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

The lesson is that different types of environments require different types of decision strategies. *In static, unchanging environments, purely proactive, goal directed behavior is adequate. But in more dynamic environments, the ability to react to changes by modifying intentions becomes more important.*

4.2.3.2 Practical Reasoning

There are seven main components to a BDI agent:

i. A **set of current beliefs,** representing information agent has about its current environment;

ii. A **belief revision function (brf),** which takes a perceptual input and the agent's current beliefs, and on the basis of these, determines a new set of beliefs;

iii. An **option generation function(options),** which determines the options available to the agent (its desires), on the basis of current beliefs and its current intentions;

iv. A **set of current options,** representing possible courses of actions available to the agent;

v. A **filter function (filter),** which represents the agent's deliberation process, and which determines the agent's intentions on the basis of its current beliefs, desires, and intentions;

vi. A **set of current intentions**, representing the agent's current focus – those states of affairs that it has committed to trying to bring about;

vii. An **action selection function (execute),** which determines an action to perform on thebasis of current intentions.

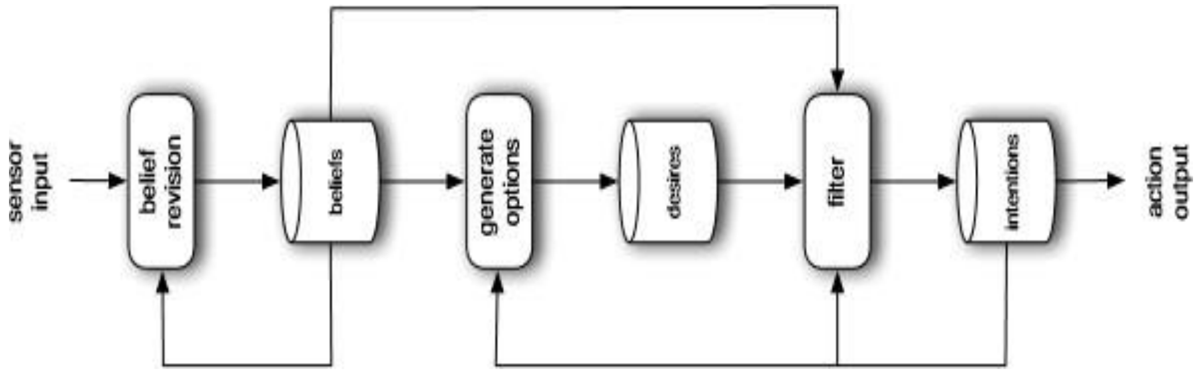


Figure 1.5: Schematic diagram of a generic belief-desire-intention architecture.

The state of BDI agent at any given moment is a triple (B,D,I), where $B \subseteq Bel$, $D \subseteq Des$, and $I \subseteq Int$. If we denote the set of possible percepts that the agent can receive by P, then $brf : 2^{Bel} \times P \to 2^{Bel}$

$options : 2^{Bel} \times 2^{Int} \to$

2^{Des} $filter:2^{Bel} \times 2^{Des} \times 2^{Int}$

$\to 2^{Int}$

Thus filter should satisfy the following constraint:

$\forall B \in 2^{Bel}, \forall D \in 2^{Des}, \forall I \in 2^{Int}, filter(B, D, I) \subseteq I \cup$

$D.execute : 2^{Int} \to A$

```
function $action(p)$ returns an action
begin
    $B := brf(B,p)$
    $D := options(B,I)$
    $I := filter(B,D,I)$
    return $execute(I)$
end function $action$
```

Figure 4.6 Pseudo-code of function action

4.1.3 Layered Architectures

Layered Architectures involves creating separate subsystems, as a hierarchy of interacting layers, to deal with reactive and proactive behaviors. Two examples of such architectures: INTERRAP and TOURINGMACHINES. There are two types of control flow within layered architectures.

- **Horizontal layering** - In horizontally layered architectures (Figure 1.6(a)), the software layers are each directly connected to the sensory input and action output. In effect, each layer itself acts like an agent, producing suggestions as to what action to perform. Advantage – Simplicity. Drawback - overall behavior of the agent will not be coherent.
- **Vertical layering** - In vertically layered architectures (Figure 1.6(b) and 1.6(c)), sensory input and action output are each dealt with by at most one layer each. A mediator function makes decisions about which layer has ―control‖ of the agent at any given time. Drawback - designer must potentially consider all possible interactions between layers.

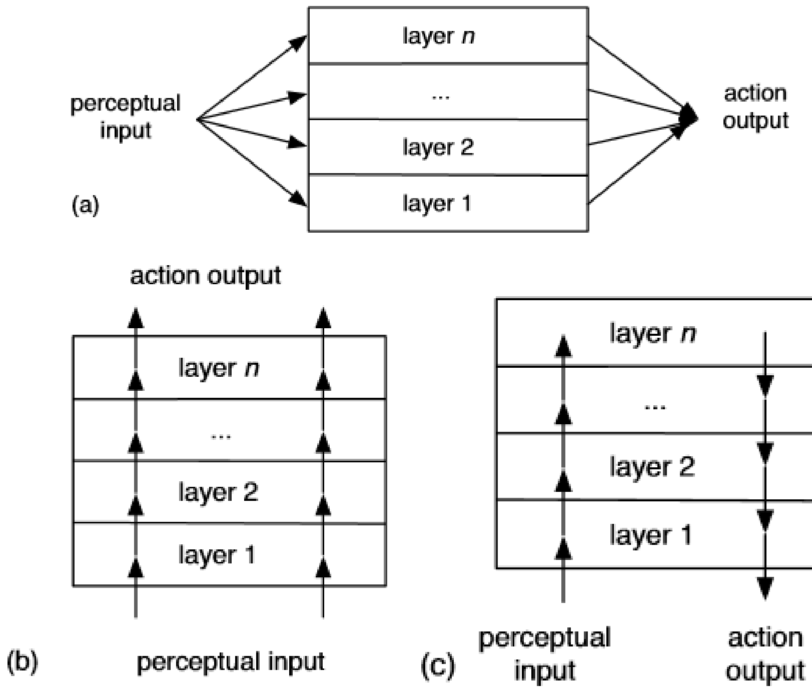


Figure 1.6: Information and control flow in three types of layered agent architectures

Vertically layered architecture is subdivided into one-pass architectures (Figure 1.6(b)) and two-pass architectures (Figure 1.6(c)). The complexity of interactions between layers is reduced in vertical architecture. Vertical architecture is much simpler than the horizontally layered case. However, this simplicity comes at the cost of some flexibility.

4.2.4.1 Touring Machines

The Touring Machines architecture consists of perception and action subsystems, which interface directly with the agent's environment, and control layers embedded in a control framework, which mediates between the layers. TOURING MACHINES consists of three activity producing layers.

- Reactive layer: immediate response.
- Planning layer: ‒day-to-day‖ running under normal circumstances.
- Modelling layer: predicts conflicts and generate goals to be achieved in order to solve these conflicts.
- Control subsystem: decided which of the layers should take control over the agent.

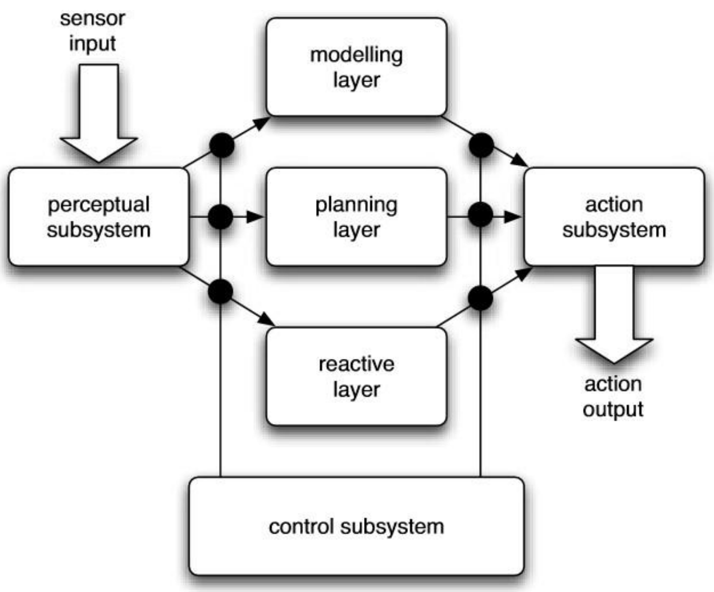


Figure 1.7: TOURING MACHINES - a horizontally layered agent architecture.

4.2.4.2 INTERRAP

INTERRAP defines an agent architecture that supports 1) **situated behavior -** agents can recognize unexpected events and react timely and appropriately to them. 2) **goal-directedbehavior** - agent decides which goals to pursue and how. The agents act under real time constraints, with limited resources and interact with other agents to achieve common goals.

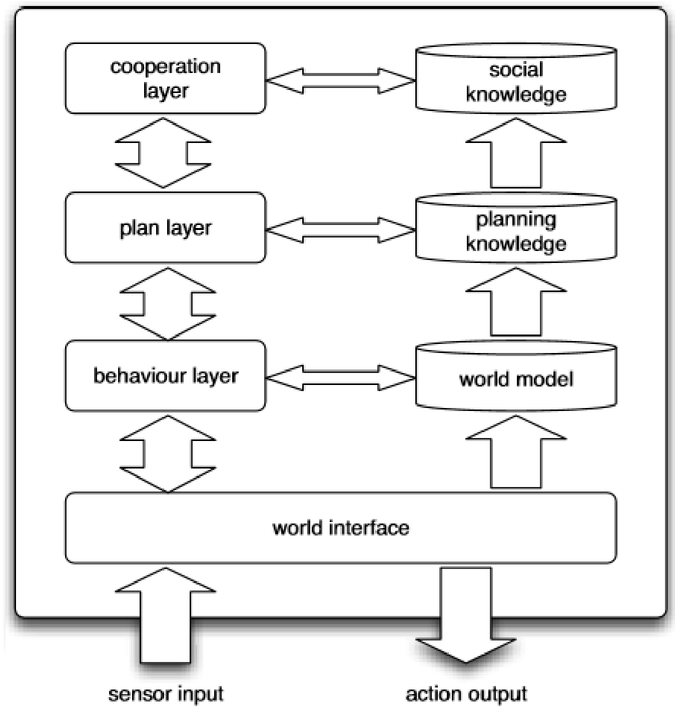


Figure 1.8: INTERRAP - a vertically layered two-pass agent architecture.

Control Component	Corresponding Knowledge base	Function
Cooperation	Cooperation knowledge	Generate joint plans that satisfy the goals of a number of agents, in response to request from the plan-based component.
Plan-based	Planning knowledge + plan library	Generate single-agent plans to requests from the behavior-based component.
Behaviour-based	World Model	Implement and control the basic reactive capability of the agent. Call on the world interface or a higher-level layer to generate a plan.
World Interface	World Model	Manages the interface between the agent and its environment.