Reactive Architecture

An approach to agency is 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). This section presents a survey of the *subsumption architecture*, which is arguably the best-known reactive agent architecture. It was developed by Rodney Brooks – one of the most vocal and influential critics of the symbolic approach to agency to have emerged in recent years.

There are two defining characteristics of the subsumption architecture. The first is that an agent's decision-making is 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' 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

$$\text{situation} \rightarrow \text{action}$$

which simple map perceptual input directly to actions.

The second defining characteristic of the subsumption architecture is that many behaviours can "fire" simultaneously. There must obviously be a mechanism to choose between the different actions selected by these multiple actions. Brooks proposed arranging the modules into a *subsumption hierarchy*, with the behaviours arranged into *layers*. Lower layers in the hierarchy are able to *inhibit* higher layers: the lower a layer is, the higher is its priority. The idea is that higher layers represent more abstract behaviours. For example, one might desire a behaviour in a mobile

robot for the behaviour "avoid obstacles". It makes sense to give obstacle avoidance a high priority – hence this behaviour will typically be encoded in a *low-level* layer, which has *high* priority. To illustrate the subsumption architecture in more detail, we will now present a simple formal model of it, and illustrate how it works by means of a short example. We then discuss its relative advantages and shortcomings, and point at other similar reactive architectures.

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

The decision function *action* is 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 $s \in S$ iff $see(s) \in c$. Let $Beh = \{(c, a) \mid c \subseteq P \text{ and } a\}$ be the set of all such rules.

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

```
1.  function action(p : P) : A
2.  var fired : ℘(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
```

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

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

To illustrate the subsumption architecture in more detail, we will show how subsumption architecture agents were built for the following scenario:

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 re-enter 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 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. Instead, he proposes a solution using the subsumption architecture.

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

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

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

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

if detect an obstacle *then* change direction (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 (2)
if carrying samples and *not* at the base *then* travel up gradient (3)

Behaviour (8) ensures that agents carrying samples will return to the mother-ship (by heading towards the origin of the gradient field). The next behaviour ensures that agents will collect samples they find.

if detect a sample *then* pick sample up (4)

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

if true *then* move randomly (5)

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

$$1 \prec 2 \prec 3 \prec 4 \prec 5$$

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.

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

- If agents do not employ models of their environment, then they must have sufficient information available in their *local* environment for them to determine an acceptable action;
- Since purely reactive agents make decisions based on *local* information, (i.e., information about the agents *current* state), it is difficult to see how such decision making could take into account *non-local* information – it must inherently take a "short term" view;
- It is difficult to see how purely reactive agents can be designed that *learn* from experience, and improve their performance over time;
- A major selling point of purely reactive systems is that overall behaviour *emerges* from the interaction of the component behaviours when the agent is placed in its environment. But the very term "emerges" suggests that the relationship between individual behaviours, environment, and overall behaviour is not understandable. This necessarily makes it very hard to *engineer* agents to 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.

References

M. Wooldridge, *Intelligent Agents*, in G. Weiss (ed.), Multiagent systems, The MIT Press, 1999.