Lecture 04.02 Reactive robot control architecture

reactive control architecture

Robot control that is characterized by sense data being simply mapped to simple actions that work together to achieve tasks is said to have a *reactive control architecture*. By "simply mapped," we mean a long calculation is not required to determine the appropriate action. Frequently, the maps are simple rules like, "If s then a." For instance, "if a dropoff is detected ahead, stop." This structure is called a *finite state machine* (FSM). A FSM models a robot-environment "world" as consisting of a finite number of states, exactly one of which exists at each moment. *State transitions* occur from one state to another when some conditions are met. In the case of "If s_1 then a_1," we define a *state transition function* (map) f_1 that maps (sense) event s_1 to action a_1, which presumably will change the actual state to some (usually) new state s_2.

finite state machines

state transitions

state transition function

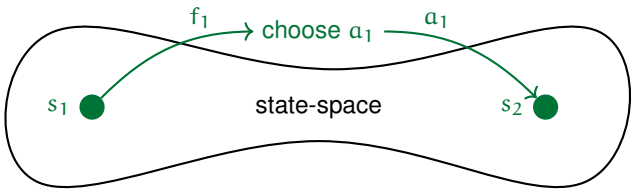

Figure 04.2: a state transition from s_1 to s_2 via the state transition function f_1 and action a_1.

For simple actions, it is easy to see how these maps work. For more-complicated actions, especially those involving long sequences of simple actions, it is not so clear how to go about designing such maps. This is especially true when we consider the frequently large number of possible states in which the robot could be: for every position, orientation, speed, distance from objects, etc., actions must be specified. In other words, the state-space is usually large and if we imagine, as designers, assigning an action to each state ... we see the trouble: there are too many possible states to choose an action for each. In other words, the problem is usually *intractable*.

intractable

One approach is to break the state-space into subspaces and assign actions to these, instead of individual states. But a further complication here arises: what if the subspace domains of these maps aren't mutually exclusive? Consider Figure 04.3. In the region of overlap $S_1 \cap S_2$, both f_1 and f_2 apply, leading to different actions a_1 and a_2. Sometimes there is no conflict and both actions are desirable (and non-conflicting); other

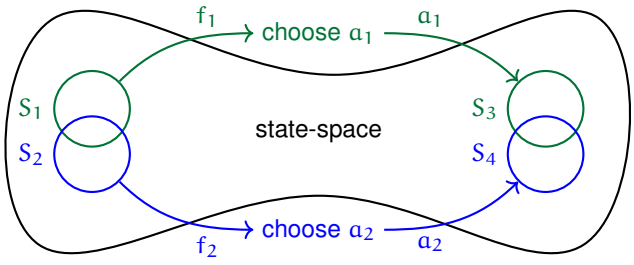

Figure 04.3: two overlapping subspaces with corresponding state transitions.

times, only one or the other is desirable, so *arbitration* is necessary; finally, sometimes a *fusion* is desirable in which the original actions are mixed in some way. For instance, perhaps $S_1 = \{\text{an object is on the left}\}$ and $S_2 = \{\text{an object is on the right}\}$. In, for instance, a corner of a room, both will be true, so the state $s \in S_1 \cap S_2$ obtains. If a_1 is "continue and angle right" and a_2 is "continue and angle left," which seems reasonable, something must be be done because there is clearly a conflict here. If we proceed by arbitration, either a_1 or a_2 is chosen, but neither is probably desirable. We could proceed by a simple fusion in which we simply "add" the two actions (programmatically and not electro-mechanically, which would waste of power and could cause damage to the robot): the robot would just continue forward. No, instead, we probably want what could be considered a new subspace-function-action or a more complex fusion, something like "stop, rotate by some angle, and continue."

arbitration
fusion

But even if a designer could go through each subspace and assign it an action in a reasonable amount of (design) time, which actions (and arbitrations) should they choose in each state to consistently achieve desired tasks?

To even further complicate things, the state of the robot must be estimated from measurements, from which it is not always possible to completely or accurately reconstruct the state. And even when it is possible, the estimation process can be model-dependent and therefore it may take (run) time—something generally discouraged in a reactive control architecture.

These challenges indicate a systematic design approach. This is provided by the *subsumption [reactive] architecture* (SA), to which we now turn. But before we describe its structure, it is worth considering some of its fundamental design principles.

subsumption architecture (SA)

04.02.1 The world is its own best model

The motto, (Brooks, 1999)

> *The world is its own best model*

is one of the fundamental principles of the SA and other reactive architectures. The idea here is that it is better to get information about the world from itself than from models thereof—that is: measure it, and now! This means the SA relies very little on models and computation. For instance, consider a robot performing the three-action task T_1: (a_1) pick up an object; (a_2) open the hatch; and (a_3) place the object inside. We could reason as follows: when we pick up an object, open the hatch, then place the object inside. That is, $a_1 \Rightarrow a_2 \Rightarrow a_3$. The implicit assumption, here, is that we know how things will go. But the world is a fickle place, my friend. The object was slippery and part way through being picked up (a_1), it was dropped, and nothing was placed inside! Or the hatch got stuck (a_2) and our manipulator crashed into it (a_3)! However, using the principle that the world is its own best model, we would not rely on such (FSM) logic. Instead, we might use realtime sensor information, interpreted as events, say

- $s_1 = \{$sensed an object to pick up$\}$,
- $s_2 = \{$sensed an object near the hatch$\}$, and
- $s_3 = \{$sensed the hatch is open$\}$.

Then events would proceed as follows:

- if s_1 then a_1 (should cause s_2),
- if s_2 then a_2 (should cause s_3), and
- if s_3 then a_3.

This is much more resilient; if, for instance, the hatch gets stuck, then $\neg s_3$ and therefore $\neg a_3$—that is, the manipulator would not crash into the hatch.[2]

communication

At this point, we can see another way of thinking about this design principle: it is as if *communication* among the modules that act is channeled *through the world itself*. Instead of communicating among modules through

[2] This simple example ignores the obvious fact that even a non-reactive control architecture would probably make more extensive use of sensor data than imagined here. Similarly, the "model" here is a simplistic FSM logic: if action a_i, then the event I expect will certainly follow. Most models would be more nuanced and be updated from sensor data; however, added model detail leads to slower responses.

software or hardware signals, the results of each one's action in the world (environment-robot) are simply *there* and need no other "model."

Although this principle was originally developed by the founders of the reactive control architecture, it has really become a general design principle in all robotic control. And let's not kid ourselves: it has its limitations. The most significant limitation is *temporal*: sometimes the past and the (modeled) future are relevant to what actions we would be best taken now. Furthermore, sensor data is imperfect and incomplete: although we have said the world is "simply there," this is actually a fantasy, and we always have to *estimate* what is going on from measurements. It is more honest to say "it is easier to measure most things than to model them."

temporal limitations

estimate

04.02.2 Evolution and emergence

The next design principle of the subsumption architecture is

> *Start with the simplest actions. I.e.—design bottom-up!*

The apparent banality of this is deceiving: it is easy to get stuck thinking about "high-level" behaviors when we begin designing. While we cannot forget that these are the goal, in a subsumption architecture (and beyond), the simplest actions are first. The next principle is related:

> *Iteratively include more actions, debugging along the way.*

The idea is to try to form more-complex tasks by including more actions. How might these actions combine? The following design principle begins to answer this question:

> *Higher actions can override lower ones.*

We mean "higher" in a sense already alluded to, but which will become more precise in the next section. Given our bottom-up approach, lower-levels are designed early and higher-levels are designed later. In this sense, the subsumption architecture design process follows biological *evolution*, which starts simply, builds incrementally, and overrides selectively.

evolution

Finally, consider the final design principle:

> *Complex tasks emerge from combinations of simpler actions.*

This is a sort of "promise" that complexity can be achieved by following these design principles: it is reasonable to expect *emergence*. Given the success of this architecture in many robot applications, it seems well-founded.

emergence

04.02.3 The subsumption architecture

The subsumption architecture uses a type of finite state machine (FSM) model.[3] Transition functions map subsets of the state-space among each other.

module **layer** Design proceeds incrementally by *module* aka *layer*, each of which contains one or (usually) several state transition function definitions. A layer is designed to achieve a *task* like "stand up" or "drive forward" or "wander." Layers are *stacked* "up," with the higher layers having two privileged capabilities over lower layers:

task **stacked**

suppression A higher layer can *suppress* (turn off) one or more of a lower layer's input(s).

inhibition A higher layer can *inhibit* (turn off) one or more of a lower layer's output(s).

This provides a great deal of flexibility in the design process. For instance, consider mobile robot with two layers: a layer L_1: *wander* and a higher layer L_2: *avoid obstacles*. Most likely, it will be necessary for L_2 to inhibit at least some of the outputs of L_1, with L_2, doing its best impersonation of Jesus, "taking the wheel," if you will.

04.02.4 Similar reactive architectures

It is worth mentioning that many reactive architectures have been developed from some or all of the principles of the subsumption architecture. In particular, the behavior-based architecture of Lecture 04.04 is a direct extension thereof. Others, such as SMACH (`wiki.ros.org/smach`) from the Robot Operating System (ROS – we will introduce ROS in Part II). SMACH uses what is called a *hierarchical state machine* that has several advantages.

hierarchical state machine

[3] Brooks uses some nonstandard terminology here that can cause confusion. He calls the fundamental building unit of the subsumption architecture an "augmented finite state machine" or AFSM. By "state machine," he seems to mean what we have called a state transition function and action. By "augmented," he seems to mean the inclusion of a regular transition-action, registers, timers, and connections thereamong (Brooks, 1999, p. 30).