Analysis of Adaptation and Environment

Ian Horswill
MIT Artificial Intelligence Laboratory
ian@ai.mit.edu

Abstract

Designers often improve the performance of artifical agents by specializing them. We can make a rough, but useful distinction between specialization to a task and specialization to an environment. Specialization to an environment can be difficult to understand: it may be unclear on what properties of the environment the agent depends, or in what manner it depends on each individual property. In this paper, I discuss a method for analyzing specialization into a series of *conditional optimizations*: formal transformations which, given some constraint on the environment, map mechanisms to more efficient mechanisms with equivalent behavior. I apply the technique to the analysis of the vision and control systems of a working robot system in day to day use in our laboratory.

The method is not intended as a general theory for automated synthesis of arbitrary specialize agents. Nonetheless, it can be used to perform *post-hoc* analysis of agents so as to make explicit the environment properties required by the agent and the computational value of each property. This *post-hoc* analysis helps explain performance in normal environments and predict performance in novel environments. In addition, the transformations brought out in the analysis of one system can be reused in the synthesis of future systems.

1 Introduction

Scientists and mathematicians seek general principles: individual principles that each explain a large class of phenomena. Engineers seek general mechanisms, but are often forced for one reason or another to use highly specialized ones. When one needs to solve a wide range of problems, it may be more desirable to design a set of specialized mechanisms than to pay the price needed to build a single mechanism that can solve all problems.

Computer science, being a curious combination of engineering and mathematics, often pushes both extremes of specialization and generality at once. Theorists and programming language designers search for ever simpler more compact abstract computing machines that are still Turing-equivalent (*e.g.* the 2-counter Turing machine [33] or the lambda calculus [10][41]), while computer architects search for the best collections of specialized circuits with which to emulate the behavior of these general computing machines (Hennessy and Patterson [17]). Finaly, compiler designers search for better methods for automatically mapping the general machines into specialized machines (Aho, Sethi, and Ullman [3]).

Throughout this paper, I will adopt the somewhat artificial distinction between specialization to a task (*e.g.* navigation vs. car assembly) and to an environment (*e.g.* forests vs highways). Specialization of an agent to a task is no different than the specialization of a normal computer program to a task. The designer usually has an explicit definition of the task and consciously uses that definition in the design of the agent or program. Often the internal structure of the mechanism reflects the internal structure of the task, with modules of the mechanism corresponding to subproblems of the overall task. (This is not as clear in the case of biological agents, see Beer [7].) However, it is rare for a designer to have a complete formal description of the behavior of her agent's environment (the exception being simple virtual worlds). In addition, the agent's assumptions about its environment are often not explicitly represented within the agent. Such tacit knowledge may be spread diffusely throughout the agent. These factors conspire to make specialized agents difficult to understand.

The fundamental claim of this paper is that environmental specialization can be usefully described in terms of transformations over possible agents that provably preserve behavior when the agent is situated in some specific type of environment. The issue of when specialized mechanisms should be used in the first place is outside the scope of the paper.

2 Example

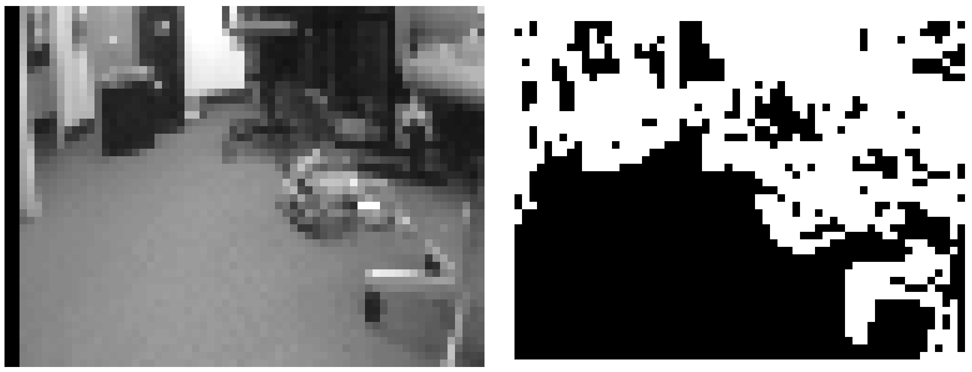

Figure 1: (left) Image of an office taken from the robot's camera. The dots in the lower middle of the image are artifacts due to the quantization in the rendering process. The structure in the lower right hand portion of the image is a 5-legged office chair. The structures in the top-left are (left to right) a doorway viewed from an oblique angle, a small trash can, and a file cabinet. The homogeneous region in the lower and middle left is the carpet. (Right) The pixels with significant texture.

Figure 1 shows an image of an office taken with a camera mounted on a robot. Suppose we want the robot to avoid obstacles by turning left when there is more free space to the left and right when there is more free space to the right. To do this, the robot must determine which side of the image has more free space. This amounts to the problem of finding which regions of the floor are free and which have objects on top of them. The problem is difficult because the image projection process loses information, depth information in particular, and so we cannot uniquely determine the structure of the scene without additional information either in the form of additional images or of additional assumptions.

A common way of solving the problem is to build a complete depth map of the scene and then project the features in the depth map into the floor plane. Those parts of the floor onto which no features are projected will be free space. A common way of building depth maps is to use two cameras in a stereo configuration. Distinctive features (usually edges) can be found in the two images and matched to one another. Given the matching of the features, we can compute each feature's shift due to parallax, and from that, the 3D positions of the features (see Barnard and Fischler [4]).

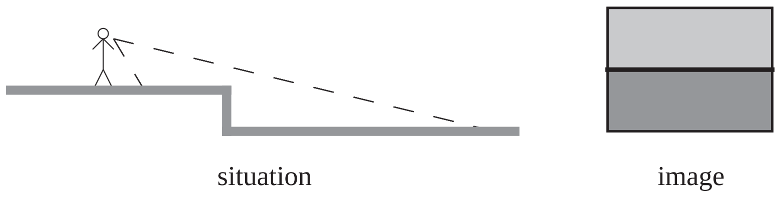

Figure 2: An observer views a cliff of a textureless surface (left). Although variations in lighting of the two sides of the cliff may produce a local variation in image brightness at the point of discontinuity (right), there is still no texture in the image above or below the discontinuity which would allow the observer to infer the depth, or even the presence, of the cliff from stereo data.

The stereo approach, while perfectly reasonable, does have undesirable properties. It is computationally expensive, particularly in the matching phase. It may also require very high resolution data. A more important problem is that the floor in this environment appears textureless from a distance, and so has no features to match. Figure 1 shows a map of the image in which pixels with significant texture (actually, significant intensity gradients) are marked in white. The region corresponding to the floor is uniformly black. The stereo process cannot make any depth measurements in the most important region of the image because there are no features there to be matched. The problem can be remedied by interpolating a flat surface in the absence of texture from which to compute depth. In that case, the stereo system is working not because it is measuring the depth of the floor directly, but because it is making a smoothness assumption that happens to be true of floors in office environments. The assumption is not true in the general case (see figure 2).

This brings out two important points. First *truly general systems are extremely rare*, and so claims of generality should be considered carefully. Often the mechanisms we build have hidden assumptions. These can be particularly difficult to uncover in advance because we may unconsciously choose test data that fit them. This is not to say that implicit assumptions are bad. Quite the contrary: those assumptions can lead to great improvements in performance. However, we as engineers need to make *informed decisions* about our use of specialization. We need to understand more clearly what assumptions our agents make about their environments, and how often those

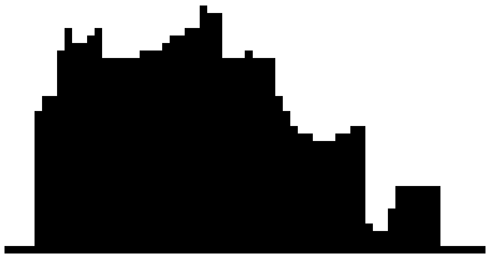

Figure 3: The carpet blob extracted from figure 1 using the coloring algorithm. Note that the blob is taller where there is more exposed carpet.

assumptions are true of the particular environments in which they operate.

2.1 A more efficient algorithm

The stereo system worked on the scene in figure 1 because the floor was flat and the obstacles had texture. We can make a different system to solve the problem, one that is much more efficient, by using these facts directly and by treating the lack of texture on the floor as a useful feature of the environment rather than a problem to be overcome.

Notice that the floor forms a single, connected black blob at the bottom of figure 1. This blob is shown alone in figure 3. I will call this the carpet blob. The carpet blob is easily computed by region coloring: starting at the bottom of the screen, trace up each image column, marking pixels until a textured pixel is found in that column. The marked pixels will form the blob. The height of the blob varies with the amount of exposed floor in the corresponding direction, giving us a rough and ready measure of the amount of free space in that direction.

We can then solve our navigation problem simply by extracting the carpet blob and turning in the direction in which the carpet blob is tallest. This technique is the basis of the low level navigation capabilities the Polly system (Horswill [18][19]), a mobile robot that gives simple tours of the AI lab at MIT. The navigation algorithm can easily be executed in real time on a low-end personal computer.

2.2 Preliminary analysis of coloring algorithm

Both the stereo algorithm and the coloring (blob-based) algorithm are specialized mechanisms that make assumptions about the structure of their environments. They perform properly when run in environments in which their assumptions hold, but may fail otherwise. Unfortunately, their assumptions are not explicitly represented. Neither algorithm would have any mention of the flatness of the floor in its source code listing, except, perhaps, as a comment.

We can understand the coloring algorithm by deriving it from the stereo algorithm by means of a series of transformations. The stereo system measures freespace directly by computing a depth map and projecting it into the floor plane. Since we are only concerned with determining which side has more freespace, however, we do not need to know the exact distances in any particular unit of measure. Any measure will do provided that we use it consistently. In fact, we can substitute for the stereo system *any system* that computes a strictly increasing function of the freespace. It has been known at least since Euclid that image plane height is such a distance measure for points on a ground plane. This means, roughly, that we can replace stereo computations with the image plane heights of obstacles, provided that obstacles rest on the floor and we have some way of labeling each pixel as being either obstacle or carpet. A general obstacle detector might be more difficult to build than the original stereo system. However, the carpet in this environment has a very predictable appearance—it has no texture—and so we can substitute a texture detector for the obstacle detector.

We can summarize this analysis with the following general principles:

- We can substitute any monotonic measure of a quantity for a calibrated measure, provided that the measure will only be used for comparisons.
- We can substitute height in the image plane for some other distance calculation, provided that all objects rest on the floor and there is some way of classifying pixels as being floor or object.
- We can substitute a texture detector for a floor detector, provided that the floor is textureless, and the obstacles do have texture.

These principles concisely describe the specialization of the coloring algorithm. Each describes a general transformation from a possibly inefficient algorithm to a more efficient one, along with the conditions on the task and environment that make it valid. The transformations can be used to predict the performance of the coloring algorithm in novel environments, or reused in the design of new systems. For example, if we wanted to use the blob-based algorithm in an environment with a textured carpet, we would have to abandon the last transformation, but we would still be able to use the other two. If there was some property other than texture which allowed carpet pixels to be easily classified, then we could use that property as the basis of a new transformation.

3 Preview

The main point of this paper is that we can usefully analyze specialized systems by deriving them from general systems using a chain of conditional optimizations.

Implicit in these claims is the promise that such an analysis can be made formal and precise. However, the "analysis" in the previous section was handwavy, to say the least. Most of the rest of the paper is devoted to an extended example showing one way of making the analysis precise. This entails a great deal of formalism which is otherwise uninteresting. The reader may want to skim the most formal sections or skip them entirely.

The paper should not be interpreted as arguing for any particular choice of notation. The notations used here, while quite servicable, were chosen largely because they were given to a more compact exposition than the alternatives in the literature. Nonetheless, alternatives do exist (see section 4).

Section 4 surveys the literature on environmental analysis. The rest of the paper is devoted to a detailed analysis of the navigational systems of the Polly robot. The navigation system was not cooked up to suit the needs of the formalism. Polly is a real, working, vision-based robot in day-to-day use at the MIT AI lab. Section 5 fleshes out the notions of environment, transformation, and so on. Section 6 then fully formalizes these notions for the purpose of analyzing the blob coloring algorithm, the basis of Polly's low level navigation system. Section 7 extends the formalism to encompass state changes and actions. Section 8 uses this extended formalism to analyze Polly's high level navigation system. Section 9 then gives concluding remarks.

4 Related work

Relatively little attention has been devoted to environmental specialization in computer science, mostly likely because it is only recently that we have begun to construct computational systems that are closely coupled to natural environments.

In biology, a great deal of attention has been given to the specialization of complete agents to their environments. Cybernetics, the progenitor of artificial intelligence, also focused on agent/environment interactions, although not necessarily on the properties of specific, complex environments [45]. Ideas from these areas are now being applied to artificial intelligence and robotics (see McFarland [29], Paton *et al.* [34]. Meyer and Guillot [30]).

In perceptual psychology, Gibson proposed an "ecological" theory of perception that stressed the role of the environment in forming an agent's perceptions. Gibson argued that the structure of the environment determines a set of invariants in the energy flowing through the environment and that these invariants can be directly picked up by the perceptual apparatus of the organism via a process akin to resonance.

Marr [28] argued that in order to properly understand the operation of a perceptual system (or more generally, of any intelligent system), we must understand the problem it solves at the level of a *computational theory.*[1] The computational theory defines the desired input-output behavior of the perceptual system, along with a set of *constraints* on the possible interpretations of a given input. The constraints were necessary because a single stimulus can usually be generated by an infinite number of possible situations. The virtue of a computational theory is that it abstracts away from the details of an individual mechanism. A single computational theory can be used to explain and unify many different mechanisms that instantiate it. To Marr, the role of the constraints within computational theories was to show how the structure of the environment made interpretation possible at all, not how to make it more efficient. Marr believed that the human visual system was a general mechanism for constructing three dimensional descriptions of the environment and so was relatively unconcerned with understanding how a system could be specialized to take advantage of useful, but unnecessary,

[1]Marr's actual story is more complicated than this, and used three levels of explanation, not two. See Marr [28].

properties of the environment. This work extends Marr's ideas by using constraints to explain optimizations at the implementation level.

Most formal models of environments use state-space descriptions of the environment, usually finite-state machines. Rosenschein and Kaelbling used finite state machines to represent both agent and environment (see Rosenschein [36][37], and Rosenschein and Kaelbling [38]). Their formalization allowed specialized mechanisms to be directly synthesized from descriptions of desired behavior and a formalization of the behavior of the environment. The formalization was powerful enough to form the basis of a programming language used to program a real robot. Later, Rosenschein developed a method for synthesizing automata whose internal states had provable correlations to the state of the environment given a set of temporal logic assertions about the dynamics of the environment. Donald and Jennings [12] use a geometric, but similar, approach for constructing virtual sensors.

Wilson [46] has specifically proposed the classification of simulated environments based on the types of mechanisms which can operate successfully within them. Wilson also used a finite state formalization of the environment. He divided environments into three classes based on properties such as determinacy. Todd and Wilson [43] used finite state machines to taxonomize grid worlds for a class of artificial agents created by a genetic algorithm. Littman [25] used FSM models to classify environments for reinforcement learning algorithms. Littman parameterized the complexity of RL agents in terms of the amount of local storage they use and how far into the future the RL algorithm looks. He then empirically classified environments by the the minimal parameters that still allowed an optimal control policy to be learned.

There is also an extensive literature on discrete-event dynamic systems (see Košecká [23] for a readable introduction), which also model the environment as a finite state machine, but which assume that transition information (rather than state information) is visible to the agents.

Several researchers have discussed how time-extended patterns of interaction with the environment (called "dynamics" by Agre [2]) can be used to reduce the computational burden on an agent. Lyons and Hendricks have discussed how to derive and exploit useful dynamics from a formal specification of the environment [27]. They use a uniform formalization of both agent and environment based on process algebra. Using temporal logic, they are able to identify useful dynamics and design reactive behaviors to exploit them. Hammond, Converse, and Grass discuss how new dynamics can be designed into an agent to improve the stability of the agent/environment system [16].

5 Analyzing specialized agents

We will assume that we can reasonably separate the world into agent and environment. The world here need not mean the entire physical universe, only that portion of it which is relevant to our analysis. Let $\mathcal{A}$ denote some set of possible agents and $\mathcal{E}$ a set of environments. Each agent/environment pair will form a dynamic system with some behavior. We will also assume some task-specific notion of equivalence over possible behaviors. We will write $(a_1, e_1) \equiv (a_2, e_2)$ to mean that the behavior of a_1 operating in e_1 is equivalent to the behavior of a_2 in e_2. We can then say that two agents are equivalent if they are equivalent in all environments:

$$a_1 \equiv a_2 \quad \text{iff} \quad \forall e_1, e_2 . (a_1, e_1) \equiv (a_2, e_2)$$

We will call them *conditionally equivalent* given some environmental constraint C if they are equivalent in all environments satisfying C. We will write this $a_1 \overset{\text{C}}{\equiv} a_2$. Thus

$$a_1 \overset{\text{C}}{\equiv} a_2 \quad \text{iff} \quad \forall e_1, e_2 . C(e_1) \wedge C(e_2) \Rightarrow (a_1, e_1) \equiv (a_2, e_2)$$

Often, the designer has a particular behavior that they want the agent to achieve. Then the only useful behavioral distinction is whether the agent "works" or not, and so the $\equiv$ relation will divide the possible behaviors into only two classes, working and not working. Let the *habitat* H_A of agent A be set of environments in which it works. We will often refer to environment constraints as *habitat constraints*, since the habitat can be described as a constraint or conjunction of constraints.

5.1 Specialization as optimization

Suppose we want to understand an agent s that is somehow specialized to its environment. Although s might be more efficient than some more general system g, it may also have a smaller habitat, *i.e.* $H_s \subseteq H_g$. If we can find a sequence of of mechanisms s_i and domain constraints C_i, such that

$$g \overset{\mathrm{C_1}}{\equiv} s_1 \overset{\mathrm{C_2}}{\equiv} s_2 ... \overset{\mathrm{C_n}}{\equiv} s$$

then we have that $g \overset{\mathrm{C_1 \cap ... \cap C_n}}{\equiv} s$. We can phrase this latter statement in English as: *within the environments that satisfy $C_1 ... C_n$, g and s are behaviorally equivalent–they will work in exactly the same cases.* This lets us express the habitat of s in terms of the habitat of g:

$$H_s \supseteq H_g \cap C_1 \cap ... \cap C_n$$

Note that the left- and right-hand sides are not necessarily equal because there may be situations where S works but g does not. One of the constraints on the right hand side might also be overly strong.

I will call such a sequence of equivalences, in which g is gradually transformed into s, a derivation of s from g, in analogy to the derivations of equations. We will restrict our attention to the case where each derivation step $s_{i-1} \overset{\mathrm{C_i}}{\equiv} s_i$ can be seen as the result of applying some general optimizating transformation O_i that preserves equivalence given C_i (*i.e.* for which $s_i = O_i(s_{i-1})$ and for which $a \overset{\mathrm{C_i}}{\equiv} O_i(a)$ whenever $O_i(a)$ is defined). Exhibiting such a derivation breaks s's specialization into smaller pieces that are easier to understand. It also places the constraints C_i in correspondence with their optimizations O_i, making the computational value of each constraint explicit. Teasing these constraints apart helps predict the performance of the agent in novel environments. If an environment satisfies all the constraints, the agent will work. If it does not, then we know which optimizations will fail, and consequently, which parts of the design to modify. In addition, if we can write a general lemma to the effect that $a \overset{\mathrm{C_i}}{\equiv} O_i(a)$, then we can reuse O_i in the design of future systems. Such lemmas may be of greater interest than the actual agents that inspired them.

Note that we can equally well perform a derivation of one subsystem of an agent from another possible subsystem. For that reason, I will often use the term "mechanism" to mean either an agent or one of its subsystems.

6 Analysis of simple perceptual systems

In this section, we will perform a more detailed analysis of the coloring algorithm given in section 2.1. To do this, we need to flesh out the notions of environment, behavior, and behavioral equivalence. Throughout the paper, we will use a state space formalization of the environment. In this section, we will only be concerned with the environment states themselves, not with the possible transitions between them. We will also ignore the internal state of the agent. In section 7, we will add dynamics and internal state.

Let W be the set of possible world states. We will model environments as subsets of W (we will consider other state spaces in section 7.1). Thus $\mathcal{E} = 2^W$. Habitats, which we have defined as sets of environments, will then effectively just be (larger) regions of the state-space themselves. Habitat constraints, constraints over possible habitats, are then also effectively just subsets of W.

Since we are ignoring dynamics and internal state, we will consider only those perceptual systems that give information about the instantaneous world state. Thus a perceptual system is a mechanism that has an identifiable output with an identifiable set of possible states S such that the state of the output is causally determined by the state of the world. Effectively, the perceptual system computes a function from W to S. We will call that function the *information type* that the perceptual system computes. We will say that two perceptual systems are behaviorally equivalent if they compute the same information type. An information type is finite if its range is finite. Note that information types should not be confused with the concept of information as inverse probability used in classical information theory (see Hamming [15]). While the two are certainly compatible, classical information theory is concerned with measuring quantities of information, whereas our concern here is with distinguishing among different kinds of information.

6.1 Derivability and equivalence

Often what is interesting about an information type is what other information types can be computed from it. We will say that one information type $I': W \rightarrow S'$ is *derivable* from another, $I: W \rightarrow S$, if there exists a *derivation function* f for which $I' = f \circ I$. I_1 and I_2 are *equivalent* (written $I_1 \equiv I_2$) if they are interderivable.

The range of an information type is irrelevant to derivability; We can arbitrarily rename the elements of its range without changing what can be derived from it. Thus what really matters is the partition P_I it induces on the world states:

$$P_I = \{A \subseteq W \mid x, y \in A \Leftrightarrow I(x) = I(y)\}$$

The elements of the partition are the maximal sets of world states that are indistinguishable given only I. One can easily show that[2]

Lemma 1 *The following statements are equivalent:*

1. *I_1 and I_2 are equivalent, that is, interderivable.*
2. *X is derivable from I_1 iff it is derivable from I_2, for all X.*
3. *The partitions P_{I_1} and P_{I_2} are identical.*
4. *I_1 and I_2 differ only by a bijection (a* 1:1 *onto mapping).*

We will say that I and I' are *conditionally identical given C* (written $I \overset{\mathrm{C}}{=} I'$) if $I(w) = I'(w)$ for all $w \in C$. Note that $I \overset{\mathrm{W}}{=} I$ and that $I_1 \overset{\mathrm{C_1}}{=} I_2$ and $I_2 \overset{\mathrm{C_2}}{=} I_3$ implies $I_1 \overset{\mathrm{C_1 \cap C_2}}{=} I_3$. Finally, we will say that two perceptual systems are behaviorally equivalent if they compute the same information type and conditionally equivalent given C if their information types are conditionally identical given C.

6.2 Unconditional equivalence transformations

We will use a single box labeled with an information type I

$$\Rightarrow \boxed{I} \rightarrow$$

to represent a perceptual system that (somehow) computes the I. The double arrow is meant to represent a connection to the evironment. When we want to expose the internal structure in the system, we will use single arrows to represent connections wholely within the system. Thus

$$\Rightarrow \boxed{I} \rightarrow \boxed{f} \rightarrow \boxed{g} \rightarrow \ldots$$

[2]See [19] for a proof.

represents a system which first computes I and then applies the transformations f, g, ... to it. Finally, we will denote predicates with a "?", thus

$$\Rightarrow\boxed{I}\mapsto\boxed{> T?}\mapsto$$

denotes a system which outputs true when $I(w) > T$, and false otherwise. These diagrams inherit the associativity of function composition:

$$\Rightarrow\boxed{f \circ I}\mapsto\boxed{g}\mapsto \;\equiv\; \Rightarrow\boxed{I}\mapsto\boxed{f}\mapsto\boxed{g}\mapsto \;\equiv\; \Rightarrow\boxed{I}\mapsto\boxed{g \circ f}\mapsto$$

and so a simple optimization, which we might call "folding" (after constant-folding in compiler optimization), is the replacement of a series of computations with a single computation:

$$\Rightarrow\boxed{I}\mapsto\boxed{f}\mapsto \;\equiv\; \Rightarrow\boxed{f \circ I}\mapsto$$

One example of an optimizing transformation is what might be called "decalibration." Estimating precise parameters such as depth can be difficult and can require precise sensor calibration. Often all that is done with this information is to compare it to some empirical threshold. For example, we might estimate the distance to an obstacle to decide whether we should swerve around it, or whether it is too late and we must brake to avoid collision. Generally, the designer arbitrarily chooses a threshold or determines it experimentally. In such situations, we can use *any* mechanism that computes distance in any units, provided that we correct the threshold.

Lemma 2 *(Decalibration) For any information type $I{:}W{\rightarrow}\mathcal{R}$ ($\mathcal{R}$ is the set of real numbers) and any strictly increasing function $f{:}\mathcal{R}{\rightarrow}\mathcal{R}$,*

$$\Rightarrow\boxed{I}\mapsto\boxed{> T?}\mapsto \;\equiv\; \Rightarrow\boxed{f \circ I}\mapsto\boxed{> f(T)?}\mapsto$$

Proof: By associativity, the right hand side is equivalent to

$$\Rightarrow\boxed{I}\mapsto\boxed{(> f(T)?) \circ f}\mapsto$$

and for all x, $f(x) > f(T)$ iff $x > T$, thus $(> f(T)?) \circ f = (> T?)$. $\Box$

Decalibration allows a calibrated mechanism to be replaced with an uncalibrated mechanism, in certain cases.

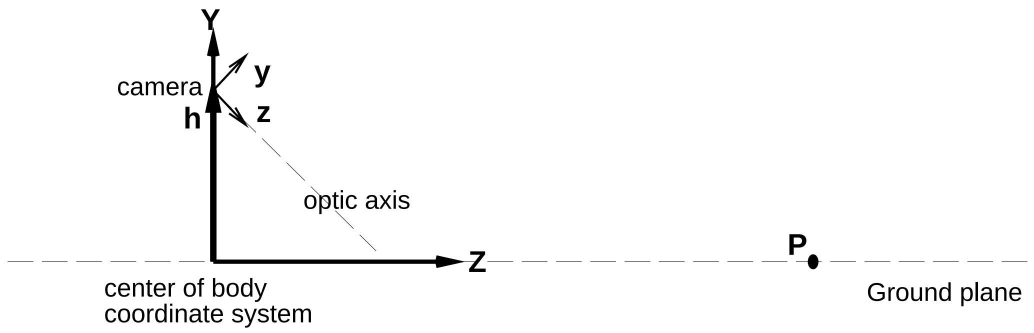

Figure 4: A camera viewing a ground plane. The **X** axis (not shown) comes out of the page and is shared by the the camera and body coordinate frames. The body coordinate frame is formed by **X**, **Y** and **Z**, the camera frame, by **X**, **y** and **z**. **z** is also the axis of projection, or optic axis, of the camera. **h** is the height of the camera and **P** is an arbitrary point on the ground plane.

6.3 Transformations over simple vision systems

The coloring algorithm used image plane height to discriminate depth and a texture detector to find obstacles. In this section, we will derive sufficient conditions for the validity of these techniques. We will show that image plane height is a strictly increasing function of object depth, provided the object rests on the floor and its projection into the floor is contained within its region of contact with the floor. We will also show that for floors whose surface markings have no spatial frequencies below ω and which are viewed from a distance of at least d, any low pass filter with a passband in the region $(0, d\omega)$ can be used to discriminate between objects and floor. The proof is not terribly interesting in itself. The reader may wish to skip to section 6.4.

First, we need to define our coordinate systems, one camera centered, in which the forward direction direction (**z**) is the axis of projection, and the other body-centered, in which the forward (**Z**) direction is the direction of motion (see figure 4). We will assume that the camera faces forward, but somewhat down, and so the camera- and body-centered frames share their left/right axis, which we will call **X**. We will call the up/down axes for the camera- and body-centered systems **y** and **Y**, respectively. We will assume that the ground plane lies at $Y = 0$. We will denote the image with range set X by $\mathcal{I}(X)$ so the b/w images are $\mathcal{I}(\mathcal{R})$ and the color images are $\mathcal{I}(\mathcal{R}^3)$.

The projection process can be specified in either of these coordinate frames. In camera-centered coordinates, the projection process maps a point (X, y, z) in the world to a point $(fX/z, fy/z)$ on the image plane, where f is the focal length of the lens. In the body-centered coordinate system, projection is best expressed with vector algebra. A point $\mathbf{P}$ in the world will be projected to the image plane point

$$\mathbf{p} = \frac{f(\mathbf{P} \Leftrightarrow \mathbf{h})}{\mathbf{z} \cdot (\mathbf{P} \Leftrightarrow \mathbf{h})}$$

(These are 3D coordinates; the 2D coordinates are obtained by projecting it onto the image plane axes $\mathbf{X}$ and $\mathbf{y}$, yielding the coordinates $(\mathbf{X} \cdot \mathbf{p}, \mathbf{y} \cdot \mathbf{p})$).

6.3.1 Salience functions and figure/ground separation

Let O be a set of objects and $FG_O{:}W \rightarrow \mathcal{I}(\{T, F\})$ ("figure/ground") be the unique information type that, for all world states, returns an image in which pixels are marked "T" if they were imaged in that world state from one of the objects O, otherwise "F." A perceptual system that can compute FG_O within its habitat can distinguish O from the background. FG_O can be arbitrarily difficult (consider the case where O is the set of chameleons or snipers). Fortunately, there are often specific cues that allow objects to be recognized in specific contexts. We will call these cues *salience functions*. An information type is a salience function if it is conditionally equivalent to FG_O given some constraint (a "salience constraint"). The use of such simple, easily computed functions to find particular classes of objects is common both in AI (see Swain [42], Turk *et al.* [44], [11], Horswill and Brooks [20], Woodfill and Zabih [47]) and in the biological world (see Roitblat [35] for a good introduction).

The coloring algorithm uses the texture detector as a salience function. We want to determine what salience constraint is required for a given texture detector. For simplicity, we will restrict ourselves to Fourier-based measures of texture. Effectively, a texture detector examines a small patch of the image. We can approximate the projection of a small patch with

$$(X, y, z) \mapsto (fX/z_0, fy/z_0)$$

where z_0 is the distance to the center of the surface patch. A sufficently small patch can be treated as a plane with a some local coordinate system (x', y').

Suppose the patch's reflectance varies as a sinusoid with frequency vector $\vec{\omega}$. Then its reflectance R at a point (x', y') on the patch is given by:

$$R(x', y') = \frac{1}{2}\left(\sin\frac{x'}{\omega_x} + \sin\frac{y'}{\omega_y}\right) + \frac{1}{2}$$

If we view the patch:

- from a unit distance,
- through a lens of unit focal length,
- from a direction normal to the patch,
- with the X axis aligned with the x' axis, and
- with even illumination of unit intensity

then the image intensity will simply be

$$I(x, y) = R(x, y)$$

Now consider the effect of changing the viewing conditions. Doubling the distance or halving the focal length halves the size of the image.

$$I(x, y) = R(\frac{x}{2}, \frac{y}{2}) = \frac{1}{2}\left(\sin\frac{x}{2\omega_x} + \sin\frac{y}{2\omega_y}\right) + \frac{1}{2}$$

The image is still a sine wave grating, but its projected frequency is doubled. Rotating the patch by and angle θ around the X axis shrinks the image along the Y axis by a factor of $\cos\theta$, producing a sine wave of frequency $(\omega_x, \frac{\omega_y}{\cos\theta})$:

$$I(x, y) = R(x, y\cos\theta) = \frac{1}{2}\left(\sin\frac{x}{\omega_x} + \sin\frac{y\cos\theta}{\omega_y}\right) + \frac{1}{2}$$

Rotating about the Y shrinks the X axis. Rotating about the optic axis simply rotates the frequency vector.

Thus a sine wave grating viewed from any position appears as a grating with identical amplitude but with a frequency vector modified by a scaling of its components and possibly a rotation. Since the projection process is linear, we can extend this to arbitrary power spectra: the power spectrum of the patch's projection will be the power spectrum of the patch, rotated and stretched along each axis (see figure 5). Frequency bands of the patch are

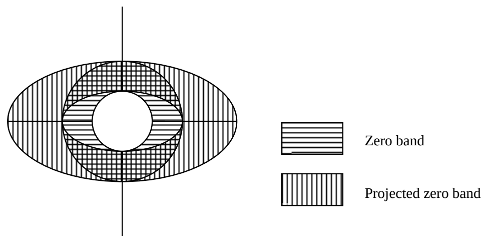

Figure 5: The effect of perspective projection on local frequency distributions.

transformed into elliptical regions of the frequency domain of its projection. Bounds on the possible viewing conditions yield bounds on how much the frequency bands can be deformed.

The *background texture constraint* (BTC) requires that all surface patches of the background have surface markings whose power spectra are bounded below by ω, that all objects have surface markings with energy below ω, and that no surface in view is closer than d focal lengths, and that the scene is uniformly lit. We have that

Lemma 3 *Any thresholded linear filtering of the image with a passband in the interval* $(0, d\omega)$ *is a salience function given the background texture constraint.*

Proof: By assumption, no patch of the background has energy in the band $(0, \omega)$, but all objects do. By the reasoning above, when any patch, either object or background, is viewed fronto-parallel from distance d, the band $(0, \omega)$ projects to the band $(0, d\omega)$. Thus a patch was imaged from an object iff its projection has energy in this band. But note that increasing the distance or changing the viewing orientation can only increase the size of the projected frequency ellipse. Thus for any distance greater than d and any viewing orientation, a patch will have energy in $(0, d\omega)$ iff it was imaged from an object. Thus a thresholded linear filter is a salience function given BTC. $\square$

The corollary to this is that any thresholded linear filter with passband in $(0, d\omega)$ is conditionally equivalent to a figure/ground system given the background texture constraint.

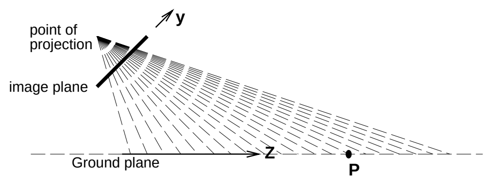

Figure 6: Monotonicity of image plane height in body depth. Rays projected from the point of projection to points on the ground plane pass through successively higher points on the image plane as they move to more distant points on the ground plane.

6.3.2 Depth recovery

Depth can be measured in either a camera-centered or a body-centered coordinate system. We will call these "camera depth" and "body depth," respectively. The camera depth of a point $\mathbf{P}$ is its distance to the image plane, $\mathbf{z} \cdot (\mathbf{P} \Leftrightarrow \mathbf{h})$. Body depth, in the other hand, is how far forward the robot can drive before it collides with the point, $\mathbf{Z} \cdot \mathbf{P}$. We will concern ourselves with body depth.

Consider a world of flat objects lying on a ground plane. Then both object points and ground plane points have zero Y coordinates. The points must be linear combinations of $\mathbf{X}$ and $\mathbf{Z}$. Since both $\mathbf{z}$ and $\mathbf{Z}$ are perpendicular to $\mathbf{X}$, the X component of the point will make no contribution either to camera depth or to body depth and we can restrict our attention to the one dimensional case, shown in figure 4, of a point $\mathbf{P} = n\mathbf{Z}$. Its body depth is simply n, while its camera depth $\mathbf{z} \cdot (n\mathbf{Z} \Leftrightarrow \mathbf{h})$ depends on camera placement. We can see by inspection, however, that the camera depth is linear in n and so camera depth and body depth are related by a linear mapping. More surprisingly, image plane height is a strictly increasing function of body depth. This can be seen from figure 6. It can also be shown analytically. The image plane height of $\mathbf{P}$ is

$$\mathbf{y} \cdot \left(\frac{f(n\mathbf{Z} \Leftrightarrow \mathbf{h})}{\mathbf{z} \cdot (n\mathbf{Z} \Leftrightarrow \mathbf{h})} \right) = \frac{\mathbf{y} \cdot (fn\mathbf{Z} \Leftrightarrow f\mathbf{h})}{n\mathbf{z} \cdot \mathbf{Z} \Leftrightarrow \mathbf{z} \cdot \mathbf{h}} = \frac{n\alpha \Leftrightarrow \delta}{n\beta \Leftrightarrow \gamma}$$

for $\alpha = f\mathbf{Z} \cdot \mathbf{y}$, $\beta = \mathbf{z} \cdot \mathbf{Z}$, $\gamma = \mathbf{z} \cdot \mathbf{h}$, and $\delta = f\mathbf{h} \cdot \mathbf{y}$. Differentiating with respect to n, we obtain

$$\frac{\alpha(n\beta \Leftrightarrow \gamma) \Leftrightarrow \beta(n\alpha \Leftrightarrow \delta)}{(n\beta \Leftrightarrow \gamma)^2} = \frac{\beta\delta \Leftrightarrow \alpha\gamma}{(n\beta \Leftrightarrow \gamma)^2}$$

When the camera looks forward and $\mathbf{P}$ is in front of the agent, we have that $n, \beta, \delta > 0$, and $\gamma\alpha < 0$, so the derivative is strictly positive.

The *ground plane constraint* (GPC) requires that the camera view a set of the objects O resting on a ground plane G, and that for each $o \in O$, o is completely in view and o's projection in G is its set of points of contact with G.[3] Thus pyramids resting on their bases would satisfy the restriction, but not pyramids resting on their points. Given GPC, we can use least y coordinate as a measure of the depth of the closest object. Let Body-Depth$_O$ be the information type which gives the correct body depth for pixels generated by one of the objects O, or ∞ for pixels generated by the background.

Lemma 4 *Let R be a region of the image. Then* $\min_R \circ$Body-Depth$_O$ *is conditionally equivalent to* $\min\{y : FG_O(x, y)$ *for some* $(x, y) \in R\}$ *given GPC, modulo a strictly increasing function.*

Proof: Note that there can only be one minimal depth, but there can be many minimal-depth object points. However, it must be the case that some contact point (an object point touching the floor) has minimal depth, otherwise there would be an object point whose ground plane projection was not a contact point, a contradiction. Let p be a minimal-depth contact point. We want to show that no object point can have a smaller projected y coordinate than p. Since the y coordinate is invariant with respect to changes in the X coordinate, a point which projects to a lesser y coordinate than p must have either a smaller Z coordinate or a smaller Y coordinate. The first would contradict p's being a minimal-depth point while the latter would place the point below the ground plane. Thus p must have a minimal y projection. We have already shown that for contact points the y projection is strictly increasing in body depth. $\square$

A trivial corollary to this lemma is that the height of the lowest figure pixel in an image column gives the distance to the nearest object in the direction corresponding to the column.

[3]Formalizing the notion of "touches" can be difficult (see for example Fleck [13], chapter 8), but we will treat the notion as primitive, since the particular formalization is unimportant for our purposes.

6.4 Derivation of the coloring algorithm

We can now derive the coloring algorithm from the stereo algorithm. Recall the stereo system:

$$\begin{tikzcd}{} \arrow[r, Rightarrow] & \boxed{\text{stereo}} \arrow[r, mapsto] & \boxed{\text{project}} \arrow[r, mapsto] & {}\end{tikzcd}$$

By lemma 4, the stereo system is conditionally equivalent given GPC (modulo a monotonic function) to any system of the form

$$\begin{tikzcd}{} \arrow[r, Rightarrow] & \boxed{\text{FG}} \arrow[r, mapsto] & \boxed{\text{column heights}} \arrow[r, mapsto] & {}\end{tikzcd}$$

where “FG” is some computation that performs figure/ground separation. By lemma 3, this is conditionally equivalent given BTC to

$$\begin{tikzcd}{} \arrow[r, Rightarrow] & \boxed{\text{filter}} \arrow[r, mapsto] & \boxed{\text{threshold}} \arrow[r, mapsto] & \boxed{\text{column heights}} \arrow[r, mapsto] & {}\end{tikzcd}$$

where “filter” is any linear filter restricted to the frequency band $(0, d\omega)$, such as an edge detector operating at a scale larger than the floor texture:

$$\begin{tikzcd}{} \arrow[r, Rightarrow] & \boxed{\text{edges}} \arrow[r, mapsto] & \boxed{\text{column heights}} \arrow[r, mapsto] & {}\end{tikzcd}$$

Since the coloring system and the stereo system yield outputs which differ by monotonic functions, it remains to be shown that substituting one for the other leads to the same motor beahvior. It can be shown that for a steering system based on balancing left and right distances the attractor and repellor basins in the robot’s configuration space are invariant with respect to this substitution, provided that we can model the steering motor as a first order system. Doing so, however, requires the introduction of still more math, so the interested reader is directed to [19].

Even this is a restricted derivation, since it assumes fully textured objects. The derivation can be extended to untextured objects with different reflectances than the background. While space precludes a full derivation, the argument goes as follows. Untextured objects still trigger the texture detector at their boundaries. We can then compute the correct figure/ground map by filling the interiors of closed contours in the texture image. However, the column heights are invariant with respect to the interior filling operation, provided that the full object is in view (if part of it runs off the bottom of the screen, the coloring of the filled and unfilled versions will differ). Thus the raw coloring algorithm will still work on untextured objects, provided they have different reflectances than the background and they are in full view.

The derivation shows that the the background texture constraint is used to to simplify figure/ground separation. More importantly, it shows that it is not used for anything else. If we wish to run the system in an environment that does not satisfy the background texture constraint, but which does satisfy the ground plane constraint, then we can substitute any salience constraint that holds in the new environment. For example, if the background has a distinctive color or set of colors, then we might use a color system such as that of Turk *et al.* [44], or Crisman [11], to find the floor:

$$\Rightarrow \boxed{\text{color}} \mapsto \boxed{\text{column heights}} \mapsto$$

If we wanted to build a system that worked in both environments, then we could implement both systems and switch between them opportunistically, provided there was sufficient information to determine which one to use. We could even implement the original stereo system in parallel with these systems and add another switch.

7 Analysis of action selection

In this section we will apply transformational techniques to action-selection tasks with the goal of demonstrating a number of formal conditions under which we can reduce deliberative planning systems to reactive systems. We will continue to model the environment as a dynamic system with a known set of possible states. First, we will add actions (state transitions) to the environment, making it a full state-machine. We will then model both deliberative planning systems and reactive systems as variants of the control policies of classical control theory (see Luenberger [26] or Beer [7]). This gives us a uniform vocabulary for expressing both types of systems. We can then examine various formal conditions on the environment that allow simplifications of the control policy (*e.g.* substitution of a reactive policy for a deliberative one)

Again, the focus of this paper is the use of transformational analysis, not the specifics of the notation used below. The notation is needed to establish

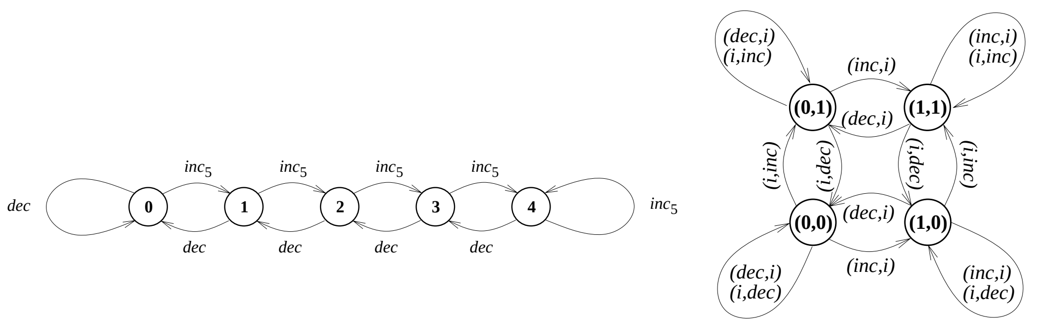

Figure 7: The environment Z_5 (left) and and the serial product of $\mathcal{Z}_2$ with itself, expressed as graphs. Function products have been written as pairs, *i.e.* $inc \times i$ is written as (inc, i). Identity actions (i and $i \times i$) have been omitted to reduce clutter.

a framework within which to apply the transformations. The notation used here is largely equivalent to those used by Rosenschein and Kaelbling [38], and by Donald and Jennings [12]. It was chosen for largely for compactness of presentation. The formal trick of externalizing the agent's internal state also turns out to be useful.

7.1 Environments

We will now allow different environments to have different state spaces and will treat actions as mappings from states to states. An environment will then be a state machine $E = (S, A)$ formed of a state space S and a set of actions A which are mappings from S to S.

For example, consider a robot moving along a corridor with n equally spaced offices labeled 1, 2, and so on. We can formalize this as the environment $\mathcal{Z}_n = (\{0, 1, ..., n \Leftrightarrow 1\}, \{inc_n, dec, i\})$, where i is the identity function, and where inc_n and dec map an integer i to $i + 1$ and $i \Leftrightarrow 1$, respectively, with the proviso that $dec(0) = 0$ and $inc_n(n \Leftrightarrow 1) = n \Leftrightarrow 1$ (see figure 7). Note that the effect of performing the identity action is to stay in the same state.

7.1.1 Discrete control problems

We will say that a *discrete control problem*, or DCP, is a pair $D = (E, G)$ where E is an environment and G, the goal, is a region of E's state space. The problem of getting to the beginning of the corridor for our robot would be the DCP $(\mathcal{Z}_n, \{0\})$. By abuse of notation, we will also write a DCP as a triple (S, A, G). A finite sequence of actions $a = (a_1, a_2, ..., a_n)$ *solves D from initial state s* if $a_n(a_{n-1}(...a_1(s))) \in G$. D em is solvable from s if such a sequence exists. *D is solvable* (in general) if it is solvable from all $s \in S$.

7.1.2 Cartesian products

Often, the state space of the environment is structured into distinct components that can be acted upon independently. The position of the king on a chess board has row and column components, for example. Thus we would like to think of the king-on-a-chess-board environment as being the "product" of the environment $\mathcal{Z}_8$ with itself (since there are eight rows and eight columns), just as $\mathcal{R}^2$ is the Cartesian product of the reals with themselves. However, consider an environment in which a car drives through an 8×8 grid of city blocks. We would also like to think of this environment as being the product of $\mathcal{Z}_8$ with itself. Both the car and the king have 8×8 grids as their state spaces, but the car can only change one of its state components at a time, whereas the king can change both by moving diagonally.

We will therefore distinguish two different Cartesian products of environments, the *parallel product*, which corresponds to the king case, and the *serial product*, which corresponds to the car case. Let the Cartesian product of two functions f and g be $f \times g\colon (a,\ b) \mapsto (f(a),\ g(b))$, and let i be the identity function. For two environments $E_1 = (S_1, A_1)$ and $E_2 = (S_2, A_2)$, we will define the parallel product to be

$$E_1 \,\|\, E_2 \;=\; (S_1 \times S_2, \{a_1 \times a_2 : a_1 \in A_1, a_2 \in A_2\})$$

and the serial product to be

$$E_1 \rightleftharpoons E_2 \;=\; (S_1 \times S_2, \{a_1 \times i : a_1 \in A_1\} \cup \{i \times a_2 : a_2 \in A_2\})$$

The products of DCPs are defined in the obvious way:

$$(E_1, G_1) \,\|\, (E_2, G_2) \;=\; (E_1 \,\|\, E_2, G_1 \times G_2)$$

$$(E_1, G_1) \rightleftharpoons (E_2, G_2) \;=\; (E_1 \rightleftharpoons E_2, G_1 \times G_2)$$

The state diagram for $\mathcal{Z}_2 \rightleftharpoons \mathcal{Z}_2$ is shown in figure 7.

We will say that a an environment or DCP is parallel (or serial) *separable* if it is isomorphic to a product of environments or DCPs.

7.1.3 Solvability of separable DCPs

The important property of separable DCPs is that their solutions can be constructed from solutions to their components:

Claim 1 *Let D_1 and D_2 be DCPs. Then $D_1 \rightleftharpoons D_2$ is solvable from state (s_1, s_2) iff D_1 is solvable from s_1 and D_2 is solvable from s_2.*

Proof: Consider a sequence S that solves the product from (s_1, s_2). Let S_1 and S_2 be the sequences of actions from D_1 and D_2, respectively, that together form S, so that if S were the sequence

$$(a \times i,\; i \times x,\; i \times y,\; b \times i,\; i \times z,\; c \times i)$$

then S_1 would be (a, b, c) and S_2 would be (x, y, z). S must leave the product in some goal state (g_1, g_2). By definition, g_1 and g_2 must be goal states of D_1 and D_2 and so S_1 and S_2 must be solution sequences to D_1 and D_2, respectively. Conversely, we can construct a solution sequence to the product from solution sequences for the components. $\square$

The parallel product case is more complicated because the agent must always change both state components. This leaves the agent no way of preserving one solved subproblem while solving another. Consider a "flip-flop" environment $F = (\{0, 1\}, \{flip\})$ where $flip(x) = 1 \Leftrightarrow x$. F has the property that every state is accessible from every other state. $F \rightleftharpoons F$ also has this property. $F \,\|\, F$ does not however. $F \,\|\, F$ has only one action, which flips both state components at once. Thus only two states are accessible from any given state in $F \,\|\, F$, the state and its flip. As with the king, the problem is fixed if we add the identity action to F. Then it is possible to leave one component of the product intact, while changing the other. The identity action, while sufficient, is not necessary. A weaker, but still unnecessary, condition is that F have some action that always maps goal states to goal states.

Claim 2 *Let D_1 and D_2 be DCPs. If $D_1 \,\|\, D_2$ is solvable from state (s_1, s_2) then D_1 is solvable from s_1 and D_2 is solvable from s_2. The converse is also true if for every goal state of D_1 and D_2, there is an action that maps to another goal state.*

Again, let S be a solution sequence from (s_1, s_2). Now let S_1 and S_2 be the sequences of obtained by taking the first and second components, respectively, of each element of S. Thus, if S is

$$(a \times x,\ b \times y,\ c \times z)$$

then we again have that S_1 is (a, b, c) and S_2 is (x, y, z). Again, S_1 and S_2 are solution sequences for their respective component problems. Similarly, we can form a solution to the product from solutions to the components by combining them element-wise. To do this, the solutions to the components must be of the same length. Without loss of generality, let S_1 be the shorter solution. Since there is always an action to map a goal state to a goal state, we can pad S_1 with actions that will keep D_1 within its goal region. The combination of S_2 and the padded S_1 must then be a solution to the product. $\square$

7.2 Agents

We will assume an agent uses some *policy* to choose actions. A policy p is a mapping from states to actions. We will say that p:

- *generates a state sequence s_i* when $s_{i+1} = (p(s_i))(s_i)$ for all i.
- *generates an action sequence a_i* when it generates s_i and $a_i = p(s_i)$ for all i.
- *solves D from state s* when p generates a solution sequence from s.
- *solves D* when it solves D from all states.
- *solves D and halts* when it solves D and for all $s \in G$, $(p(s))(s) \in G$.

For example, the constant function $p(s) = dec$ is a policy that solves the DCP $(\mathcal{Z}_n, \{0\})$, and halts.

7.2.1 Hidden state and sensors

A policy uses perfect information about the world to choose an action. In real life, agents only have access to sensory information. Let $T\colon S \to X$ be the information type (see section 6) provided by the agent's sensors. The crucial question about T is what information can be derived from it. We will say that an information type is *observable* if it is derivable from T.

To choose actions, we need a mapping not from world states S to A, but from sensor states X to A. We will call such a mapping a *T-policy.* A function p is a T-policy for a DCP D if $p \circ T$ is a policy for D. We will say that p *T-solves* D from a given state if $p \circ T$ solves it, and that p T-solves D (in general) if it T-solves it from any initial state.

7.2.2 Externalization of internal state

We have also assumed that the agent itself has no internal state–that its actions are determined completely by the state of its sensors. In real life, agents generally have internal state. We will model internal state as a form of external (environmental) state with perfect sensors and effectors. Let the *register environment* R_A over an alphabet A be the environment whose state space is A and whose actions are the constant functions over A. We will write the constant function whose value is always a as $\mathcal{C}_a$. The action $\mathcal{C}_a$ "writes" a into the register. We will call $E \,\|\, R_A$ the *augmentation* of E with the alphabet A. An agent operating in the augmentation can, at each point in time, read the states of E and the register, perform an action in E, and write a new value into the register.

Using external state for internal state is not simply mathematical artifice. Agents can and do use the world as external memory. An agent need only isolate some portion of the world's state (such as the appearance of a sheet of paper) which can be accurately sensed and controled. Humans do this routinely. Appointment books allow people to keep their plans for the day in the world, rather than in their scarce memory. Bartenders use the position of a glass on the bar to encode what type of drink they intend to mix and how far they are into the mixing (see Beach [6]). For an example of a program that uses external state, see Agre and Horswill [1].

7.3 Progress functions

A *progress function* is a measure of distance to a goal. In particular, a progress function Φ for a DCP $D = (S, A, G)$ is a non-negative function from S to the reals for which

1. Φ is nonnegative, *i.e.* $\Phi(s) \geq 0$ for all s.
2. $\Phi(s) = 0$ iff $s \in G$.
3. For any initial state i from which D is solvable, there exists a solution sequence $S = (a_1, ...a_n)$ along which Φ is strictly decreasing (*i.e.* $\Phi(a_j(...(a_1(i)))) > \Phi(a_{j+1}(a_j(...a_1(i))))$ for all j).

The term "progress function" is taken from the program verification literature, where it refers to functions over the internal state of the program that are used to prove termination of loops. Progress functions are also similar to Liapunov functions (see Luenberger [26]), admissible heuristics (see Barr and Feigenbaum [5], volume 1, chapter II), and artificial potential fields (see Khatib [21] or Latombe [24]).

We will say that a policy p *honors* a non-negative function Φ, if Φ steadily decreases it until it reaches zero, *i.e.* for all states s and some $\epsilon > 0$, either $\Phi((p(s))(s)) < \Phi(s) \Leftrightarrow \epsilon$ or else $\Phi(s) = \Phi((p(s))(s)) = 0$. A policy that honors Φ can be thought of as doing hill-climbing on Φ and so will run until it reaches a local minimum of Φ. When Φ also happens to be a progress function for the DCP, that local minimum will be a global minimum corresponding to the goal:

Lemma 5 *Let Φ: $S \to \mathcal{R}$ be non-negative and let p be a policy for a DCP D that honors Φ. Then p solves D and halts exactly when Φ is a progress function on D.*

Proof: Consider the execution of p from an arbitrary initial state i. On each step, the value of Φ decreases by at least ϵ until it reaches 0, after which it must remain zero. Thus Φ must converge to zero within $\frac{\Phi(i)}{\epsilon}$ steps after which the state of the system is confined to the set $\Phi^{-1}(0)$. We need only show that $\Phi^{-1}(0) \subseteq G$ iff Φ is a progress function for D. If Φ is a progress function $\Phi^{-1}(0) \subseteq G$ holds by definition. To see the converse, suppose $\Phi^{-1}(0) \subseteq G$. We want to show that from every state from which D is solvable, there is a solution sequence that monotonically decreases Φ. The sequence generated by p is a such a sequence. $\Box$

Progress functions can be generated directly from policies. The *standard progress function* $\Phi_{p,D}$ on a policy p that solves D is the number of steps in which p solves D from a given state. An important property of product DCPs is that we can construct progress functions for products from progress functions for their components:

Lemma 6 *If Φ_1 is a progress function for D_1 and Φ_2 is a progress function for D_2, then $\Phi\colon (x, y) \mapsto \Phi_1(x) + \Phi_2(y)$ is a progress function for the serial product of the DCPs.*

Proof: Since $\Phi_1 \geq 0$ and $\Phi_2 \geq 0$, we have that $\Phi \geq 0$. Similarly, Φ must be zero for exactly the goal states of the product. Now suppose the product is solvable from (s_1, s_2). Then there must exist solution sequences for the components that monotonically decrease Φ_1 and Φ_2, respectively. Any combination of these sequences to form a solution to the product must then monotonically decrease Φ, and so Φ must be a progress function for the product. $\Box$

Again, the parallel case is more complicated:

Lemma 7 *If Φ_1 is a progress function for D_1 and Φ_2 is a progress function for D_2, and for every goal state of D_1 and D_2 there is an action that maps that state to a goal state, then $\Phi\colon (x, y) \mapsto \Phi_1(x)+\Phi_2(y)$ is a progress function for the parallel product of the two DCPs.*

Proof: Again, we have that $\Phi \geq 0$ and that Φ is zero for exactly the the goal states of the product. Now consider a state (s_1, s_2) from which the product is solvable. There must be solution sequences S_1 and S_2 to the component problems along which Φ_1 and Φ_2, respectively, are strictly decreasing. Without loss of generality, assume that S_1 is the shorter. Of the two solutions. We can pad S_1 and combine the solutions to produce a solution to the product. The padding cannot change the value of Φ_1, and so the value of Φ must be strictly decreasing along the combined solution. $\Box$

7.4 Construction of DCP solutions by decomposition

7.4.1 Product DCPs

We now have the tools to construct solutions to product DCPs from the solutions to their components:

Lemma 8 *Let p_1 be a policy which solves D_1 and halts from all states in some set of initial states I_1, and let p_2 be a policy which solves D_2 and halts from all states in I_2. Then the policy*

$$p(x, y) = p_1(x) \times p_2(y)$$

solves $D_1 \parallel D_2$ and halts from all states in $I_1 \times I_2$. (Note that here we are using the convention of treating p, a function over pairs, as a function over two scalars.)

Lemma 9 *Let p_1 be a policy which solves D_1 from all states in some set of initial states I_1, and let p_2 be a policy which solves D_2 from all states in I_2. Then any policy for which*

$$p(x, y) = p_1(x) \times i \text{ or } i \times p_2(y)$$

and

$$\begin{array}{rcl} y \in G_2, x \notin G_1 & \Rightarrow & p(x, y) = p_1(x) \times i \\ x \in G_1, y \notin G_2 & \Rightarrow & p(x, y) = i \times p_2(y) \end{array}$$

will solve $D_1 \rightleftharpoons D_2$ and halt from all states in $I_1 \times I_2$.

Proof: We can prove both lemmas using progress functions. Let Φ_{p_1,D_1} and Φ_{p_2,D_2} be the standard progress for p_1 and p_2 on D_1 and D_2, respectively. Their sum must be a progress function for the product. This follows directly for the serial case, and from the fact that p_1 and p_2 halt for the parallel case. Since the policies for both products clearly honor the sum, they must solve their respective products. Note that the constraint given in the second lemma is sufficient, but not necessary. $\square$

7.4.2 Reduction

We can often treat one environment as an abstraction of another; The abstract environment retains some of the fundamental structure of the concrete environment but removes unimportant distinctions between states. An abstract state corresponds to a set of concrete states and abstract actions correspond to complicated sequences of concrete actions.

Let a projection of an environment $E = (S, A)$ into an abstract environment $E' = (S', A')$ be a mapping $\pi\colon S \to S' \cup \{\perp\}$. π gives the abstract state for a given concrete state or else $\perp$ if it has no corresponding abstract state. π^{-1} gives the concrete states corresponding to a given abstract state. For sets of states, we will let $\pi^{-1}(S) = \cup_{s \in S}\ \pi^{-1}(s)$.

We define a π-implementation of an abstract action a' to be a policy that reliably moves from states corresponding to an abstract state s' to states corresponding the abstract state $a'(s')$ *without visiting states corresponding to other abstract states.* Thus for any s' for which $a'(s')$ is defined, the implementation solves the DCP

$$(\pi^{-1}(\{s', \perp, a'(s')\}),\ A,\ \pi^{-1}(a'(s')))$$

Note that we do not require p to stay in $\pi^{-1}(a'(s'))$ upon reaching it.

Given π-implementations $p_{a'}$ of each abstract action a', we can use an abstract policy p' to solve problems in the concrete environment by emulating the abstract actions. We need only look up the abstract state corresponding to our current concrete state, look up the abstract action for the abstract state, and run its implementation. This suggests the policy

$$p(s) = p_{p'(\pi(s))}(s)$$

This concrete policy works by taking the state s, looking up its abstract state $\pi(s)$, computing the proper abstract action $p'(\pi(s))$, and then computing and running the next concrete action in its implementation $p_{p'(\pi(s))}$. Note that since this policy has no internal state, it effectively recomputes the abstract action each time it chooses a concrete action. This is no problem when the concrete environment is in a state that corresponds to an abstract state, but the π-implementations are allowed to visit states that have no abstract state. To handle this problem, it is necessary to add a state register to the environment to remember what abstract action is presently being performed. The policy for the augmented environment computes a new abstract action whenever the environment is in a concrete state with a corresponding abstract state. It stores the name of the new abstract action in the register for later use, while also executing it its implementation. When the environment is in a concerete state with no abstract state, it uses the abstract action stored in the register and preserves the value in the register:

Lemma 10 *Let $D = (S, A, G)$, $D' = (S', A', G')$ be DCPs, π be a projection of D into D', and for each action $a' \in A'$, let $p_{a'}$ be a π-implementation of a' in D. If p' is a policy which solves D', then the policy*

$$p(s,a) = \begin{cases} p_a(s) \times \mathcal{C}_a & \text{if } \pi(s) = \perp \\ p_{p'(\pi(s))}(s) \times \mathcal{C}_{p'(\pi(s))} & \text{otherwise} \end{cases}$$

solves the augmentation of D with the alphabet A', from any state in $\pi^{-1}(S')$.

Proof: Let $\Phi_{p',D'}$ be the standard progress function for p' on D' and let $s \in P^{-1}(S')$. Then $\Phi_{p',D'}(\pi(s))$ is the number of abstract actions need to solve the problem from the concrete state s. If $\Phi_{p',D'}(\pi(s)) = 0$, then the problem is already solved, so suppose that p solves the problem from states s for which $\Phi_{p',D'}(\pi(s)) = n$ and consider an s for which $\Phi_{p',D'}(\pi(s)) = n + 1$. The policy p will immediately compute $p'(\pi(s))$, store it into the register. Call this action a'. The policy p will also immediately begin executing $p_{a'}$. Since this policy is a p-implementation of a', the system must reach a state in $\pi^{-1}(a'(\pi(s)))$ in finite time, which is to say that it will reach the next state in D'. By assumption, p' can solve D' from this high level state in n steps, and so p must be able to solve D from s, and so, by induction p solves D for all $s \in P^{-1}(S')$. □

We will say that D is *reducible* to D' if there exists a projection π of D into D' and π-implementations of all of actions in D'. If D is reducible to D' then we can easily convert any solution to D' into a solution to D.

8 Analysis of a robot navigation system

Consider the problem of piloting a robot about the office environment shown in figure 8. At any given moment, the robot must decide given its destination how fast to turn and how fast to move forward or backward. Polly uses the policy of following corridors except when it reaches intersections. At intersections it compares the coordinates of the intersection to the coordinates of its goal (presumed to be another intersection) and turns north when the goal

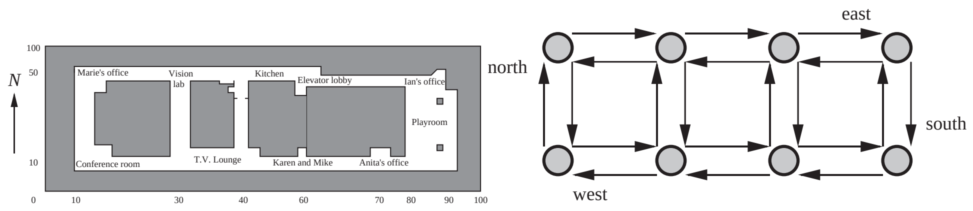

Figure 8: Approximate layout of the 7th floor of the AI lab at MIT (left) and its toplogical structure (right).

is to the north, south when the goal is to the south, and so on:

$$p_{polly}(sensors) = \begin{cases} stop & \text{if at goal} \\ turn\text{-}north & \text{if north of goal and at turn to north} \\ turn\text{-}south & \text{if south of goal and at turn to south} \\ \ldots & \\ turn\text{-}north & \text{if south of goal and pointed south} \\ turn\text{-}south & \text{if north of goal and pointed north} \\ \ldots & \\ follow\text{-}corridor & \text{otherwise} \end{cases}$$

The details of the perception and control systems are given in [19].

8.1 Derivation from a geometric path planner

Geometric path planning is a common technique for solving this type of problem. Given a detailed description of the environment, a start position, and a goal position, a path planner computes a safe path through the environment from start to goal (see Latombe [24]). Once the path has been planned, a separate system follows the path. Geometric planning is versatile and can produce very efficient paths, but is not computationally efficient. It also requires detailed knowledge of the environment which the perceptual system may be unable to deliver.

We can clarify the relationship between a path planning system and Polly's reactive algorithm by deriving Polly's algorithm from the planing system. Let $\mathcal{N}$ be the DCP whose states are (position, orientation) pairs and whose actions are small (translation, rotation) pairs such as the robot might move in one clock tick. Clearly, Polly can be modeled as an $\mathcal{N}$ policy. However, the planner can equally well be modeled as an $\mathcal{N}$ policy. A planner/executive is simply a policy that uses internal state to compute and execute a plan. The planning portion uses scratch memory to gradually compute a plan and store it in a plan register, while the executive reads the finished plan out of the register and executes each segment in turn. Thus a planner/executive architecture has the form:

$$\begin{aligned} p_0(s, plan, scratch) &= \begin{cases} i \times \text{plan}_{\mathcal{N}}(s, scratch) & \text{if } plan \text{ incomplete} \\ \text{execute}(plan) \times i & \text{otherwise} \end{cases} \\ \text{execute}(plan) &= \text{head}(plan) \times \mathcal{C}_{\text{tail}(plan)} \end{aligned}$$

An agent in $\mathcal{N}$ will spend nearly all its time in corridors. The only real choice points in this environment are the corridor intersections. Thus only the graph of corridors and intersections $\mathcal{N}'$, need be searched, rather than the full state space of $\mathcal{N}$ (see figure 8). By lemma 10, we can augment the environment with a register to hold the current north/south/east/west action and replace p_0 with the policy

$$p_1(s, action) = \begin{cases} p_{p'_1(I(s))}(s) \times \mathcal{C}_{p'_1(I(s))} & \text{if at intersection} \\ p_{action}(s) \times \mathcal{C}_{action} & \text{otherwise} \end{cases}$$

where:

- $I(s)$ is the intersection at state s
- the different p_{action} policies implement following north, south, east, and west corridors, respectively, and
- p'_1 is an arbitrary $\mathcal{N}'$ policy.

The lemma requires that the goal always be a corridor intersection and that the robot always be started from a corridor intersection. We could now solve $\mathcal{N}'$ by adding plan and scratch registers and using a plan/execute policy:

$$p'_1(intersec, plan, scratch) = \begin{cases} i \times \text{plan}_{\mathcal{N}'}(intersec, scratch) & \text{if } plan \text{ incomplete} \\ \text{execute}(plan) \times i & \text{otherwise} \end{cases}$$

We can simplify further by noting that $\mathcal{N}'$ is isomorphic to $\mathcal{Z}_4 \rightleftharpoons \mathcal{Z}_2$, that is, the corridor network is a 4×2 grid. By lemma 9, we can replace p'_1 with any policy that interleaves actions to reduce grid coordinate differences between the current location and the goal. We can then remove the plan and scratch registers from p_1 and reduce it to

$$p_2(s, action) = \begin{cases} p_{p'_2(I(s))}(s) \times \mathcal{C}_{p'_2(I(s))} & \text{if at intersection} \\ p_{action}(s) \times \mathcal{C}_{action} & \text{otherwise} \end{cases}$$

where p'_2 is any $\mathcal{N}'$ policy satisfying the constraints that (1) it only stops at the goal, and (2) it only moves north/south/east/west if the goal is north/south/east/west of $I(s)$.

There are still two important differences between p_2 and p_{polly}: Polly uses a different set of actions ("turn north" instead of "go north") and it has no internal state to keep track of its abstract action. While it appears to use a qualitatively different policy than we have derived, it does not. Within a short period of beginning a *north* action, an agent will always be pointed north. Similarly for *east*, *south*, and *west* actions. The orientation of the robot effectively is the *action* register and turn commands effectively write the register. There's no need for internal memory. Polly stores its state in its motor.

We can summarize the transformations used in the derivation as follows (see table 1). The containt that the environment consist of a network of corridors and that the goal be a corridor intersection allows us to replace geometric planning with planning in the corridor graph. The isomorphism of the corridor graph to a grid allows us to replace planning with difference reduction. Finally, the correlation of the robot's orientation with its internal state allows us to store the current action in the orientation.

It is important to note that either, both, or neither of the subproblems (the abstracted environment and corridor following) could be solved using deliberative planning; the two decisions are orthogonal. If both are implemented using planners, then the resulting system is effectively a hierarchical planner (see Sacerdoti [39] or Knoblock *et al.* [22]). Polly's environment happens to allow the use of simple reactive policies for both, so it is a layered reactive system (Brooks [9]). In an environment with a more complicated graph topology, one could reverse the second optimization and use a deliberative planner, leaving the first optimization intact. The result would then be a hybrid system with planning on top and reacting on the bottom (see Spector and Hendler [40], Lyons and Hendriks [27], Bresina and Drummond

Constraint	Optimization
ground plane constraint	use height for depth estimation
background-texture constrint	use texture for obstacle detection
corridor network	replace planning in $\mathcal{N}$ with planning in $\mathcal{N}'$
grid structure	replace planning with difference reduction
orientation correlation	store state in orientation

Table 1: Summary of constraints and optimizations used in Polly's navigation system.

[8], or Gat [14] for other examples). On the other hand, one could imagine an environment where the individual corridors were cluttered but were connected in a grid. In such an environment, the abstract problem could be solved reactively, but corridor following might actually require deliberative planning.

9 Conclusions

Fundamentally, this paper is about explanation. For one reason or another, we are often faced with agents or other mechanisms that operate properly in one type of environment but not in another. In such cases, we want to explain the agent's performance in different environments. Transformational analysis is a way of reverse-engineering ones own programs. It reduces an agent's environmental specialization to a series of lemmas giving the conditions under which different optimizations are possible. The lemmas are often more enlightening than the agents themselves. No one cares what edge detector Polly uses. The constraint (given in lemma 3) which background surface markings place on the choice of edge detector is far more interesting. Once the lemmas have been obtained, they can be used to predict the performance of old agents in new environments or to suggest designs for new agents in old environments. Given a sufficient stock of optimization lemmas, one can imagine developing cookbook methods for designing particular kinds of situated agents, much as cookbook methods are currently used in electrical and mechanical engineering.

A discussion of when specialization is appropriate is outside the scope of this paper. The issue is not so much whether to build specialized systems or general systems as how we can be intelligent consumers of specialization. The literature is full of specialized systems, although they are often not billed as such. We must think carefully about whether an agent works because of the generality of its design or because of serendipitous properties of test data.

We must study the environment not only formally but experimentally. The knowledge we use of the external world to design our agents is necessarily incomplete. I have used the transformational techniques discussed here primarily for *post-hoc* analysis. It is rare that I understand the structure of my sensors and environment well enough for my first guess at an algorithm be robust. Performing a derivation based on plausible constraints that turn out to be empirically false is wasted effort. The environmental constraints we encode within our agents are partial theories of the environment. Theories must be tested. If we take this notion seriously, then we must view artificial intelligence as a form of natural science. To understand intelligence, we must study not only ourselves but our world.

Acknowledgements

Phil Agre, Rod Brooks, Bruce Donald, Eric Grimson, Maja Mataric, David Michael, Ray Paton, Stan Rosenschein, and Lynn Stein all provided much needed feedback during the development of these ideas. The reviewers provided many useful suggestions for improving the presentation. Support for this research was provided in part by the University Research Initiative under Office of Naval Research contract N00014–86–K–0685, and in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014–85–K–0124.

References

[1] Philip Agre and Ian Horswill. Cultural support for improvisation. In *Tenth National Conference on Artificial Intelligence*, Cambridge, MA, 1992. American Assoiciation for Artificial Intelligence, MIT Press.

[2] Philip E. Agre. The dynamic structure of everyday life. Technical Report 1085, Massachusetts Institute of Technology, Artificial Intelligence Lab, October 1988.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[4] Stephen T. Barnard and Martin A. Fischler. Computational and biological models of stereo vision. In *Proc. DARPA Image Understanding Workshop*, September 1990.

[5] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence.* William Kaufmann, Inc., 1981.

[6] King Beach. Becoming a bartender: The role of external memory cues in a work-directed educational activity. *Journal of Applied Cognitive Psychology*, 1992.

[7] Randall Beer. A dynamical systems perspective on autonomous agents. CES 92-11, Case Western Reserve University, Cleveland, Ohio, 1992.

[8] J. Bresina and M. Drummond. Integrating planning and reaction. In J. Hendler, editor, *AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments.* AAAI, March 1990.

[9] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automoation*, 2(1):14–23, March 1986.

[10] Alonzo Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, NJ, 1951.

[11] Jill D. Crisman. Color region tracking for vehicle guidance. In Andrew Blake and Alan Yuille, editors, *Active Vision*, chapter 7. MIT Press, Cambridge, MA, 1992.

[12] Bruce Randall Donald and James Jennings. Constructive recognizability for task-directed robot programming. *Robotics and Autonomous Systems*, 9:41–74, 1992.

[13] Margaret M. Fleck. Boundaries and topological algorithms. TR 1065, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1988.

[14] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings, AAAI-92*, 1992.

[15] Richard W. Hamming. *Coding and Information Theory.* Prentice Hall, Englewood Cliffs, N.J. 07632, 1980.

[16] Kristian J. Hammond and Timothy M. Converse. Stabilizing environments to facilitate planning and activity: An engineering argument. In *Ninth National Conference on Artificial Intelligence*, pages 787–793, Menlo Park, CA, July 1991. American Association for Artificial Intelligence, AAAI Press.

[17] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[18] Ian Horswill. Polly: A vision-based artificial agent. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 824–829. AAAI, MIT Press, 1993.

[19] Ian Horswill. *Specialization of perceptual processes.* PhD thesis, Massachusetts Institute of Technology, Cambridge, May 1993.

[20] Ian Horswill and Rodney Brooks. Situated vision in a dynamic environment: Chasing objects. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, August 1988.

[21] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 5(1):90–98, 1986.

[22] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. A spectrum of abstraction hierarchies for planning. In *Proceedings of AAAI-90*, 1990.

[23] Jana Košecká. Control of discrete event systems. GRASP LAB report 313, University of Pennsylvania Computer and Information Science Department, Philadelphia, PA, April 1992.

[24] Jean-Claude Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.

[25] Michael L. Littman. An optimization-based categorization of reinforcement learning environments. In Meyer and Wilson [32], pages 262–270.

[26] David G. Luenberger. *Introduction to Dynamic Systems: Theory, Models, and Applications.* John Wiley and Sons, 1979.

[27] D. M. Lyons and A. J. Hendriks. Exploiting patterns of interaction to achieve reactive behavior. *in submission*, 1993.

[28] David Marr. *Vision*. W. H. Freeman and Co., 1982.

[29] David McFarland. What it means for robot behavior to be adaptive. In Meyer and Wilson [31], pages 22–28.

[30] Jean-Arcady Meyer and Agnes Guillot. Simulation of adaptive behavior in animats: Review and prospect. In Meyer and Wilson [31], pages 2–14.

[31] Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, Massachusetts, 1991.

[32] Jean-Arcady Meyer and Stewart W. Wilson, editors. *From Animals to Animats: The Second International Conference on Simulation of Adaptive Behavior.* MIT Press, Cambridge, Massachusetts, 1993.

[33] Marvin Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, NJ, 1967.

[34] R. C. Patton, H. S. Nwana, M. J. R. Shave, and T. J. M. Bench-Capon. Computing at the tissue/organ level (with particular reference to the liver). pages 411–420. MIT Press, Cambridge, MA, 1992.

[35] Herbert L. Roitblat. *Introduction to Comparitive Cognition.* W. H. Freeman and Company, 1987.

[36] Stanley J. Rosenschein. Formal theories of knowledge in ai and robotics. report CSLI-87-84, Center for the Study of Language and Information, Stanford, CA, 1987.

[37] Stanley J. Rosenschein. Synthesizing information-tracking automata from environment descriptions. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 386–393, May 1989.

[38] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of machines with provable epistemic properties. In Joseph Halpern, editor, *Proc. Conf. on Theoretical Aspects of Reasoning about Knowledge*, pages 83–98. Morgan Kaufmann, 1986.

[39] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2), 1974.

[40] Lee Spector and James Hendler. The supervenience architecture. In Avi Kak, editor, *Working notes of the AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, pages 93–100. AAAI Press, Asilomar, California, 1991.

[41] Guy L. Steele and Gerald Jay Sussman. The revised report on scheme: an interpreter for the extended lambda calculus. AI Memo 452, MIT Artificial Intelligence Laboratory, Cambridge MA, 1978.

[42] Michael J. Swain. Color indexing. Technical Report 390, University of Rochester Computer Science Department, November 1990.

[43] Peter M. Todd and Stewart W. Wilson. Environment structure and adaptive behavior from the ground up. In Meyer and Wilson [32], pages 11–20.

[44] Matthew A. Turk, David G. Morgenthaler, Keith Gremban, and Martin Marra. Video road following for the autonomous land vehicle. In *1987 IEEE Internation Conference on Robotics and Automation*, pages 273–280. IEEE, March 1987.

[45] Norbert Wiener. *Cybernetics.* MIT Press, Cambridge, 1961.

[46] Stewart W. Wilson. The animat path to ai. In Meyer and Wilson [31], pages 15–21.

[47] John Woodfill and Ramin Zabih. Using motion vision for a simple robotic task. In *AAAI Fall Symposium on Sensory Aspects of Robotic Intelligence*, 1991.