

Contents

1	Introduction	7
1.1	The Vivarium Project	8
1.1.1	Present state of the Vivarium	8
1.1.2	Vivarium scenario for 1998	9
1.2	How to read this thesis	11
2	Background, Basic Concepts, and Related Work	12
2.1	Model-building in ethology	12
2.1.1	Classical ethological models	13
2.1.2	Computational ethology	15
2.1.3	Computational evolution	16
2.2	Programming technology	16
2.2.1	Lego/Logo	17
2.2.2	Graphic programming	17
2.2.3	Programming by example	18
2.2.4	Animals as metaphors for processes	18
2.3	Animation	19
2.4	Artificial intelligence and animal behavior	20
2.4.1	Society of Mind	21
2.4.2	Situated action	24
2.4.3	Subsumption architectures for robotics	26
2.4.4	Case-based reasoning	27
2.4.5	Implications for artificial intelligence	27
3	Animal Parts	29
3.1	Worlds	29
3.2	Physical objects	31
3.3	Creatures	31
3.4	Sensors	32
3.5	Motors	33
3.6	Computational elements	33
4	3-Agar: Agents as Gated Rules	36
4.1	Language basics	36
4.1.1	Creature definition	37

4.1.2	Agent definition	38
4.1.3	Value-returning forms	39
4.1.4	Action-producing forms	40
4.2	Argument passing between agents	40
4.3	The gating mechanism	41
4.4	Sensing	42
4.5	Example: ant foraging and recruitment	43
4.5.1	Smelly worlds	44
4.5.2	Foraging	45
4.5.3	Recruitment	47
4.6	Other examples	48
4.6.1	Oscillators	48
4.6.2	Finite state machines	49
4.7	Comparison with other designs	50
4.8	User Environment	50
5	Early Designs	52
5.1	BrainWorks: graphic neural programming	53
5.1.1	Neurons	55
5.2	0-Agar: connectionist agents	56
5.3	1-Agar: memories and rules	57
5.4	2-Agar: procedural agents	59
6	Issues for Distributed Agent Languages	62
6.1	State	62
6.1.1	State and sequences	63
6.2	Pushing vs. pulling	65
6.2.1	A mixed system	66
6.3	Timing constructs	67
7	Learning and Evolution	70
7.1	K-lines and R-trees	70
7.2	Script agents	72
7.3	Evolving turtles	75
7.3.1	Evolving to exploit bugs	76
8	Conclusions	78
8.1	Ethology: results and future work	78
8.2	Programming technology: results and future work	79
8.3	Animation: results and future work	79
8.4	Artificial intelligence: results and future work	79

List of Figures

2.1	A partial behavioral hierarchy	13
2.2	<i>A</i> - and <i>B</i> -brains (after [Minsky87, p. 59])	22
4.1	A generalized sensor	43
4.2	Ant following a trail in nature and in Agar (from [Wilson71])	44
4.3	A finite state machine diagram	49
5.1	The turtle-wiring window from BrainWorks	53
5.2	The turtle moving in the BrainWorks world	54
5.3	The Bar Graph display/interaction window from 0-Agar	57
5.4	The trace window from 2-Agar	60
6.1	Sequence machine, with pointer	63
6.2	Stateless sequence machine	64
6.3	Timing constructs	68
7.1	Learning to recognize an attacker	71
7.2	A script and associated agent	74

Chapter 1

Introduction

Formalism and mathematics are attracted by the centres, if I dare make this metaphor, like rats and insects by granaries.

— Bruno Latour, *Science in Action*

An *animal construction kit* is an interactive computer system that allows novice programmers to assemble artificial animals from simple components. These components include sensors, muscles or other effectors, and computational elements. They can also include body parts such as limbs, bones, and joints. A complete animal construction kit must also provide a world for the animal to live in, and must support the co-existence of multiple animals of different species.

There are many reasons to build artificial animals. The most important one is to understand how real animals work. Building working models allows us to verify our theories about how animals see the world, make decisions, and act. Ethologists have always speculated on the mechanisms behind the animal behavior they observed, but were hampered by the lack of tools for building complex models. The computer simulation medium provides a way to construct and test complex theories of behavior.

Building artificial animals can also lead us to new understandings of human behavior. Computational models of human cognition have usually failed to take into account the more basic problems faced by an animal trying to get about in the real world, problems shared by humans. By focusing on animals we emphasize that intelligent behavior involves a process of interaction between a creature and its environment.

Artificial animals can drive interactive computer technology forward, both by pushing its limits and providing it with new tools. Computer animation and interactive programming techniques are two of the technologies involved. To satisfy our goal of realism, animals should eventually be animated in real-time and drawn in an expressive, artistic manner. This is just barely possible now using special-purpose hardware. Programming technology has to be able to let a novice user program, debug, and monitor a complex system consisting of many communicating parts.

Animal construction kits will also return improvements to the technologies they are exploiting. Computer animators can use the languages developed for programming artificial animals to give their characters autonomy and to express motion in terms of goals and broad actions rather than in explicit geometrical motions. We can also improve the technologies of novice programming by using animals as a metaphor for the processes inside a computer. Since artificial animals are active and goal-directed, they can serve as tangible representations of computational processes.

1.1 The Vivarium Project

This work was undertaken as part of the Vivarium Project, a research effort sponsored by Apple Computer. Vivarium was inspired by Ann Marion's Aquarium program, an interactive fishtank simulation. The goal of Vivarium is to design an educational computer environment for grade-school children, where they will have the ability to construct their own animals. This will require user interface techniques that can make the abstract processes of behavior concrete and accessible to children. These include graphic interactive programming, force-feedback for tactile interaction, and real-time three-dimensional graphics for heightened realism. This work deals with the designing languages that children can use to construct behavioral models.

1.1.1 Present state of the Vivarium

I designed and built several computer systems while working towards these goals. These systems emphasize different aspects of animal construction kits, including novice programming, design of agent-based languages, and ethological realism. The first system, BrainWorks, uses a neural

model of computation with an interactive graphic interface and a physical structure based on the Logo turtle. The later systems were called Agar, after the growth medium used by microbiologists. Three versions of Agar were constructed. The Agar systems use agent-based computational models and provide more modular, flexible sets of kit components.

Several examples of animal behavior were simulated using these systems. The most complex (in terms of the number of agents involved) is the food recruitment system of ants. Recruitment is the process by which ants attract nestmates to assist in the performance of work. In this case the work consists of gathering food into the nest. This simulation involves about 40 agents per individual.

1.1.2 Vivarium scenario for 1998

In this section we present a fantasy about what the Vivarium might be like for a grade school student ten years from now.

The available technology

By 1998, we can expect to have available as standard technology:

- fully rendered, high-quality, real-time graphics
- real-time physical simulation of animal motion
- programming languages and interactive tools for describing animal behavior
- interfaces that make the above technologies accessible to children and novices.

We hope that by this time the children and novices will take the display and interface technology for granted and use it as a transparent window into an animal's world.

Irene and Roger: an imagined interaction

Irene comes into school excited. She has a new idea for her current project—a raccoon named Roger. Irene's been designing Roger for about a month, and has succeeded in making him do many things that the school's real raccoon does, such as dig a burrow. Today, Irene wants to make him wash his food before eating it.

Roger lives in the classroom's Vivarium computer. This system includes a large-screen display that provides a window into an extensive forest habitat. Other students are working on other animals in the computer. All of the animals live in the same simulated world, which is viewed through the large screen. Irene can see Roger either on the large screen or on her own desktop screen. If she chooses, she can also see the world from Roger's point of view. Roger is rendered with expressive three-dimensional graphics that make him look like a hairy animal rather than a shiny robot. While Irene has some control over rendering and often experiments with drawing styles, that's not the project for today.

Irene chooses to view Roger on the large screen while using her desktop screen as a monitoring and programming tool. The desktop screen provides an animated representation of the activity in Roger's mind as he goes about his business. His mind is composed of many interrelated agents, small computational processes that are responsible for particular functions. The component agents of Roger's mind are displayed in a graph that indicates the relationships between them and can monitor their activity.

Eventually, Roger finds some food and starts to eat it. Irene says, "Whoa!", and stops Roger. Roger disappears from the world (for his own protection while being debugged). Irene has a record of Roger's recent activity in her computer environment. She scrolls it back to the point where Roger picked up the food, and notes the agents involved. She decides that these agents need to be modified to make Roger wash his food before eating it.

Using graphic programming tools, she breaks the basic task into subtasks, creates an agent for each subtask, and specifies the control relationships between them. For example, the **eating** agent will now have to turn on a **washing** agent at some point, that will in turn have to activate **find-water** and **dunk**. Building up agents is an iterative process that involves creating an agent, specifying its behavior, and debugging. To specify physical actions, such as the dunking motion, Irene dons a special glove that enables the computer to read her own hand gestures.

Irene starts to realize that the task is more complicated than she thought. Washing involves spatial navigation, since Roger needs to find a river to wash in. Grasping and manipulation is complicated, since there are a many different kinds of object to be dealt with. Strategically, Roger has to be able to know when to give up searching for water and eat the food without washing it. Fortunately, her programming tools help Irene break down this complicated problem

into manageable sub-problems. Movie-like timeline displays help her to view Roger’s activities over time.

When Roger has been programmed to do a reasonable job of washing, Irene leaves him to run autonomously in the ongoing classroom simulation. Since Roger has some learning capabilities, he may even be better at washing the next day, or may have come up with a surprising new variation on washing.

1.2 How to read this thesis

The rest of this thesis is structured as follows:

Chapter 2, *Background, Basic Concepts and Related Work*, describes the various fields that impinge upon animal construction kits, and situates the present work with respect to them.

Chapter 3, *Animal Parts*, describes the basic components that make up an animal construction kit. These include both parts of animals and the worlds that make up the animals’ environment. The basic physical outline of an artificial animal is described here.

Chapter 4, *3-Agar: Agents as gated rules*, describes the final form of the animal simulation language. This chapter forms the technical core of the thesis.

Chapter 5, *Early Designs*, describes the designs of animal simulation systems and languages that led up to 3-Agar.

Chapter 6, *Issues for Distributed Agent Languages*, explores issues involved in the design of agent-based languages for controlling animals.

Chapter 7, *Learning and Evolution*, explores some agent-based mechanisms for animal learning, and an experiment in “species-learning” via a simulation of evolution.

Chapter 8, *Conclusions*, evaluates the successes and failures of this project and describes directions for further work.

Chapter 2

Background, Basic Concepts, and Related Work

*up in my head
there's an animal kingdom
I am the king
of the animals there
up in the mountains (my head)
I'm a beautiful singer
to birds that dance
on invisible air*

— Meat Puppets, *Animal Kingdom*

This chapter reviews some of the fields that impinge upon animal construction kits. These include ethology, programming technology, animation, and artificial intelligence. The motivations for Agar from each field are explored, and related work is discussed.

2.1 Model-building in ethology

Ethology is the study of animal behavior. Animal behavior involves two distinct but interrelated types of causality, each leading to a different kind of question: *proximate* cause (“how” questions) and *ultimate* cause (“why” questions). Proximate cause deals with the immediate workings of behavioral mechanisms. Ultimate cause deals with the evolutionary advantages of

Figure 2.1: A partial behavioral hierarchy

such mechanisms. Modeling proximate causality involves simulation of a single animal’s behavioral control system. Modeling ultimate causality involves simulating multi -generational, multi -organism systems that change very slowly.

Most of this thesis is concerned with modeling proximate cause, by building computational mechanisms that model the behavioral control system of individual animals. However, the two modes of explanation are necessarily intertwined. Any proximate mechanism must be justified by an ultimate explanation of its functionality in terms of survival and reproduction, and given a plausible evolutionary history.

2.1.1 Classical ethological models

The early ethologists searched for a unified framework for models of animal behavior. Tinbergen developed such a theory based on hierarchically-structured drive centers [Tinbergen51]. The three-spined stickleback is a small fish that Tinbergen studied extensively. The behavioral hierarchy for the stickleback reproductive cycle is illustrated in figure 2.1.

In this model, “drive energy” moves downward from the top unit to lower units under the control of *innate releasing mechanisms* (IRMs). These are sensory mechanisms triggered by

stereotypical fixed stimuli, such as the red belly of the male stickleback. The red coloration triggers an IRM in the female that initiates mating behavior.

The hierarchy of drive centers descends from general to specific units, terminating in the units that are directly responsible for specific muscle contractions. Lateral inhibition links ensure that only one unit on any given level is active at a time. This hierarchical scheme was later generalized to be a heterarchy and to include feedback mechanisms [Baerends76]. A heterarchy allows a low-level unit to be activated by more than one high-level unit.

These structural models of behavioral systems formed an organizing paradigm for ethology until the late 1950s, when the concept of drive energy fell into disrepute. It was later revived by Dawkins [Dawkins76] who pointed out that the idea of hierarchical behavior structure could be salvaged by changing the interpretation of interlevel connections from energy transfer relations to control relations.

The concept of drive or activation energy has a long history that can be traced back through Freud's concept of cathexis [Freud95] to his early 19th century sources. It has lately re-emerged as the basis of connectionist modeling [McClelland86]. Within ethology it was a powerful organizing concept until Hinde [Hinde59], who criticized drive energy theory as being a non-explanation (in that a special drive could be postulated to form the basis of any behavior) and for overextending the analogy to physical energy (i.e., explaining displacement behavior by "sparking over" of drive energy from one center to another). Although suspect, it appears to be a perennial component of theories of the mind.

Minsky [Minsky87, p. 284] suggests a new way of looking at mental energy as an emergent property rather than a fundamental mechanism. He proposes that a society of mental agents that must compete for resources would be likely to *evolve* a unit of exchange, resembling energy or currency, for negotiating among themselves. This suggests that the various energetic models are not representative of the behavior of individual neurons, but rather model the emergent dynamics of the brain.

It should be noted that some ethologists object not only to drive energy models, but to the more fundamental idea of modeling behavioral control systems as a set of interconnected functional centers. Fentress, in particular, objects to these models as erecting artificial and overly rigid boundaries between behaviors that are actually plastic, ill-defined, and mutable

[Fentress76]. His critique bears much the same relationship to orthodox ethological modeling as the connectionist critique of symbolic processing [Smolensky88] does to mainstream artificial intelligence.

Fentress has studied the development of hierarchically-structured grooming patterns in rats [Fentress83]. Apparently juvenile rats have a less differentiated repertoire of movements than do mature rats, resulting in phenomena such as “motor traps”. A motor trap is a situation in which movements that are part of more than one higher-level activity cause arbitrary switching between one activity and the other. For instance, a one to two week old rat whose forepaw passes close to its face while walking, running, or even swimming might suddenly initiate a grooming sequence. While Fentress takes this as evidence against behavioral structures as such, it could also indicate simply that the hierarchy structures itself as the animal matures.

2.1.2 Computational ethology

Computational ethology (the term was coined by David Zeltzer) includes a diverse set of efforts to build computational models of animal behavior. Early cyberneticians built simple, reflex-based artificial animals to demonstrate the possibility of mechanical behavior and learning. Grey Walter’s turtles [Walter50] responded to light and could learn by a mechanism that implemented classical conditioning. Braitenberg continued the tradition of building simple autonomous animals [Braitenberg84] by designing a series of imaginary robotic animals to illustrate concepts of “synthetic psychology”.

More recent animal simulation systems emphasize ecological interaction by supporting more than one species of artificial life. RAM [Turner87] and EVOLVEIII [Rizki86] fall into this category. The latter is particularly interesting because it simulates several levels of biological organization (genetic, behavioral, and ecological) and the relationships among them.

Stewart Wilson has studied learning in artificial animals [Wilson87]. He used classifier systems, which learn by mutating rules in a manner derived from biological genetics. This system had some success in learning simple regularities in a gridlike world. Wilson suggests that in practice, classifiers must be arranged in hierarchical modules that learn to control each other. This is reminiscent of Society of Mind theories (see section 2.4.1)) and suggests that a method for learning within individual agents.

Petworld is another animal simulation system developed under the auspices of the Vivarium [Coderre88]. Its features include a system of ranking alternative behaviors through a hierarchical structure of modules, and performance of the relatively complex task of nest-building.

2.1.3 Computational evolution

The idea of simulating individual animals can be extended to include simulating multiple generations of evolving animals. Simulating evolution is a slow and computation-intensive process, but some systems have successfully demonstrated convincing models. To do so requires modeling the basic mechanisms of evolutionary adaptation: selective survival and reproduction with survival.

Richard Dawkins has recently developed a software package called *The Blind Watchmaker* (after his book of the same name [Dawkins87]) that demonstrates the evolution of form. This system does an excellent job of creating a diversity of forms from a few simple parameters. However, the resulting form (called a “biomorph”) does not behave, and its only criterion for survival is its appeal to the eye of the user.

Niklas [Niklas86] also simulates evolution parametrically, in the realm of plants. Three factors affecting branching pattern were allowed to change by mutation: probability of branching, branching angle, and rotation angle. The plants compete with each other for available sunlight. The resulting evolutionary trajectory showed a close match to the evidence in the fossil record.

2.2 Programming technology

The goals of the Vivarium Project include not only simulating animals, but putting the ability to do so within the reach of grade-school children. This goal requires that the abstract operations of programming be made tangible in order to engage children’s knowledge about how physical objects behave and interact. Some existing techniques for making computational objects tangible, such as graphic programming, have been incorporated into Vivarium systems.

We also hope that focusing on animal behavior will contribute new techniques to the task of visualizing computational processes. Animals and processes share characteristics of being dynamic, rule-driven, and goal-directed. Thus animals can serve as metaphorical representations

of processes.

2.2.1 Lego/Logo

The Logo project [Papert80] is a long-standing effort to put computation within the grasp of young children. Logo's accessibility to children is derived in part from its use of an animistic actor (the turtle). More recently the Lego/Logo project [Ocko88] has extended the Logo turtle to include sensors as well as motors, and provided a reconfigurable mechanical medium (the Lego construction kit) for creating different physical animals.

The Lego/Logo system both gains and loses from the use of real-world physical creatures. The advantages include the large variety of possibilities the physical world offers and the greater concreteness and interest it offers to children. The disadvantages include the limitations of the Lego medium (for instance, the sensors are extremely low resolution) and the tendency of projects to focus on mechanical rather than behavioral questions.

The present Lego/Logo project also suffers from the sequential nature of the Logo language. The animal's program must be coded as a single process that consists of a loop that checks all conditions, and has no provisions for interrupts or other constructs that would allow behavioral flexibility. These limitations are the focus of current research.

2.2.2 Graphic programming

Graphic programming [Sutherland63, Smith77, Sloane86] refers to techniques that allow programs to be specified by manipulating diagrams rather than text. Graphic representations of programs can exploit common-sense notions such as connectivity, enclosure, and flow. These graphic indicators can provide a more direct representation of computational processes than the usual symbolic languages, and may be easier to teach to novices

Graphic programming has its own problems of comprehensibility. As the diagrams grow, they tend to become even less readable than textual programs; furthermore, not all programming constructs are easily translated to graphics, for example, iteration and state are hard to represent in graphic languages based on dataflow models [Sloane86]. BrainWorks (see section 5.1) uses graphic programming technology successfully because the complexity of programs, and hence diagrams, is inherently limited by the simplicity of the world model.

2.2.3 Programming by example

Programming by example refers to a class of techniques that allow a user to generate programs by direct manipulation. In such a system, the user performs a series of concrete operations on some computational objects, then generalizes these operations into routines by various methods, such as replacing constants with variables [Lieberman84, Finzer84].

Programming by example is a natural technique to use with situated creatures. Creatures in a world generate their own tasks in the form of situations that require new behaviors in the form of new agents. When the creature encounters a new situation, the user can specify the action to be taken and indicate how this action is to be generalized in similar situations. Since animal actions are already specified as local operations (i.e., “move forward 10 units”, rather than “move to absolute position (237 450)”) much of the generalization is implicit. The specification of action can be done using a menu, or more advanced systems can combine menu and gestural input using devices such as the VPL Dataglove [Foley87].

For example, imagine we’ve just created a new animal and placed it in a featureless part of the world, where it is free of all but the default sensory stimuli. Since its mind is blank, without any agents, it just sits there. We first give it a rule for this circumstance, such as “if nothing else is happening, move forward”, causing it to move forward until it runs into a wall. Then we give it a rule for dealing with walls, such as “if touching a wall, turn 180°”, allowing the animal to continue until the next novel situation. The world guides the process of programming by presenting new situations and letting us add rules incrementally.

Script-agents as described in 7.2 may be considered an attempt to do *learning* by example. This technique attempts to do the same operations as specified above, namely forming generalized responses to situations, but under the control of the creature itself rather than an external programmer. This is accomplished by having agents internal to the creature take over the role of the programmer.

2.2.4 Animals as metaphors for processes

Any language or system for manipulating information must provide some basic kinds of objects to be manipulated and operations for manipulating them. Traditional computer languages provide abstract formal objects such as numbers, strings of characters, and arrays. Novices and

children are not experienced in dealing with such objects. Good user interfaces allow the novice to draw upon existing knowledge, by presenting the computational objects using *metaphors* with tangible objects in the real world [Papert80, Goldberg76]. The properties of the objects on the screen are presented metaphorically, via an analogy to familiar objects.

Examples of user interface metaphors include: The Xerox Star and Macintosh iconic operating interfaces, that present files as tangible graphic objects that can be physically dragged from one file to another or thrown in a trash can; the HyperCard system, that presents units of information as index cards; and the Logo computer language, that presents abstract geometric operations in terms of a turtle that can perform local movements.

Young children tend to think in animistic terms. They tend to attribute action to intentions of objects rather than mechanistic forces. Thus a rock rolling down a hill is alive to a child, while a stationary rock is not [Carey85]. Computer processes share some of the attributes of living things. Processes are inherently dynamic, following a path of action determined by their program and the computational environment. A Process can have goals, which may be explicitly represented within the process or may be implicit in its design. It should be possible to exploit a young child's existing animistic framework to give the child an understanding of computer processes.

There are problems with this approach, associated with the brittleness and non-adaptiveness of computational processes. A child's experience with animals leads to certain expectations of reasonable, adaptive response. Even an insect will usually do something reasonable when confronted with a challenging situation (such as attack or flee). A computer program will not always exhibit such reasonable responses, especially when in the process of being created and debugged.

Animals are complex, autonomous, and animate. They have particular responses to stimuli, they can be in different modes or states, they can have goals, they can have emotions, they can be unpredictable. We hope to exploit the user's ideas about animals to allow her to build and understand computer processes that share some of these characteristics.

2.3 Animation

Traditional computer animation technology requires specifying explicit trajectories for each object in a scene. Current work in the field focuses on *self-scripting* animations, in which each object to be animated is responsible for determining its own motion by the application of a set of rules to its particular conditions in a simulated world.

Reynolds [Reynolds82] developed an animation system called ASAS that used a distributed control model. It introduced the idea of generating animation by having active objects that are controlled by their own individual programs, rather than a single global script. This idea was later extended to allow objects to alter their behavior based on external conditions [Reynolds87]. The resulting system was used to produce animations of animal group motion such as bird flocking. It uses a true distributed behavioral model (each bird computes its own motion). It does not attempt to simulate sensorimotor constraints (birds have direct, global knowledge of the locations of other birds and obstacles) and the behavioral model is very simple.

Kahn developed a system called ANI that creates animations from story descriptions [Kahn79]. ANI is more of a “top-down” system than others discussed here: it takes a description of a story and then computes how the characters should move to best enact that story, whereas we are more interested in specifying responses to particular situations and letting the stories unfold as emergent behavior. ANI has a great deal of relatively high-level knowledge. Some of its actors correspond to particular emotions or behaviors, i.e., **shy**, **angry**, **evil**. Others refer to particular scenes that occur in a movie: **kept-apart**, **justice**. In addition to affecting the actions of particular characters, agents can manipulate the film medium itself in order to aid the story-telling process, for instance, by introducing a close shot of two characters to suggest intimacy between them.

Zeltzer outlines a scheme for intelligent motor control [Zeltzer87]. It distinguishes between two domains of problem solving, cognitive and motor. Motor problem solving does not use symbolic representations. Instead it uses a lattice of behavioral skills that is quite similar to Tinbergen’s hierarchy of drive centers (see section 2.1.1). This research has concentrated on realistic simulation of walking and other forms of locomotion.

2.4 Artificial intelligence and animal behavior

One goal of the present work is to explore the links between animal behavior and human intelligence, and to discover computational models that can encompass both. This section surveys other work from the artificial intelligence field that is relevant to this goal. These efforts tend to emphasize the role of experience and the environment in contributing to intelligence. An emphasis on mechanism over formalism also characterizes much of this work.

2.4.1 Society of Mind

The *Society of Mind* [Minsky87] refers to a broad class of theories that model mental functioning in terms of a collection of communicating agents. An agent is a part of a mind that is responsible for some particular aspect of mental functioning. Agents communicate with and control each other, coordinating their activity to produce a coherent mind.

Agents are usually responsible for a particular task. This can be a relatively complex task, such as building a tower, or a lower-level motor task, such as moving a hand to a block. In this case, the action of the **tower-builder** would be to activate the appropriate lower-level agents, including the **hand-mover**, in the correct sequence. Agents can also serve as representational functions.

Since the Society of Mind is a distributed model, it fits well with our kit metaphor. It encourages incremental construction of mental models and an emphasis on organization and modularity. The great problem for distributed models is coordination—once we have split the functioning of mind up into isolated agents, how do we get them to cooperate in carrying out the business of generating intelligent behavior?

A-brains and *B*-brains

One Society of Mind subtheory [Minsky87, p. 59] suggests that a mind can be organized into separate layers of control. These layers are an *A*-brain that interacts with the world, and a *B*-brain that interacts solely with the *A*-brain. The *B*-brain monitors the *A*-brain's activity and exerts control over it when necessary. For example, the *B*-brain could detect loops or select which agents in the *A*-brain are to be active, based on some high-level goal. There may also be

Figure 2.2: *A*- and *B*-brains (after [Minsky87, p. 59])

further layers of control (*C*-brains).

Agar in its present state of development can be viewed as an investigation into *A*-brains, with the goal of discovering what sort of mechanisms suffice to generate simple behavior based on tightly-coupled interaction with the world. Agar's network of agents forms a basic structure of abilities on which to build higher-level intelligences. This raises two questions: First, is this structure sufficient to support intelligence? This question can be answered only by further research (some suggestions for ways to build on top of such structures are given in chapter 7). Second, is modeling this low-level structure a necessary precursor to modeling the higher levels of control? Do we need to build an *A*-brain in order to understand intelligence?

Most of the problems that artificial intelligence has investigated have been functions that would seem to belong exclusively to the *B*-brain, in that they are expressed in terms of abstract symbols that are not directly connected with the environment. Traditional AI proceeds under the assumption that it need only model these more abstract, symbolic capabilities, taking for granted that the relationship between the symbols and the world is meaningful. The meaningfulness of symbols, if it is considered at all, is relegated to a separate perception module that presents a world model to the *B*-brain. The *A*-brain, which forms the only connection between the *B*-brain and the world, is taken to be a transparent device for recreating an exact copy of the world. A similar distinction is made on the motor side: AI models of action are based on separating action into a planning phase, which involves complex reasoning, and an

execution phase, which carries out the tasks specified by a plan. Both of these assumptions allow *A*-brain functions to be effectively ignored so that attention may be focused on the presumably more intelligent functions of the *B*-brain.

Our approach is to consider the *A*-brain as a dynamic system, interacting with the world to generate behavior, rather than as a passive transmitter of percepts and actions. Taking this view of the *A*-brain implies that we must model it and the world it interacts with as a precursor to modeling more abstract forms of intelligence.

A Piagetian perspective

Piagetian psychology, a major influence on the development of Society of Mind theory, postulates several developmental stages that lead from concrete to abstract abilities. Sensorimotor capabilities are developed first, and fully abstract thought appears only at the final stages of development. The nature of the relationships between later and earlier stages is of utmost importance. If the sensorimotor stage is used only as a platform from which to construct the higher stages, then we can model the higher stages without considering the sensorimotor stage, having no obligation to recapitulate the ontogeny of the stages we are interested in.

However, it seems more reasonable to assume that the dependence of the abstract stage upon the sensorimotor stage is not only developmental, but operational as well. The *A/B*-brain model implies that the agents implementing high-level behavior work by modulating the activity of lower level ones, and so we can only talk meaningfully about the higher levels in terms of how they affect the action of the lower ones.

If in fact abstract intelligence can operate entirely on its own, this must at least be accounted for in terms of function and development. The functional issue is to determine how this detached activity is ultimately useful to the creature despite its detachment. The developmental issue is to determine by what pathways the physical structures that necessary to support detached mental activity can arise in a creature whose evolutionary history has endowed it with a brain that is primarily designed for situated activity.

The phenomenological critique of AI

The idea that abstract thought is independent of its embedding in a body has been attacked by Dreyfus and others [Dreyfus79, Winograd87]. The starting point for this attack is the philosophy of Heidegger, which emphasizes the “thrownness” of everyday activity and the dependence of action on “background”. Thrownness refers to the fact that an intelligent agent is always in a situation and cannot avoid acting within and on that situation. Thrownness is to be contrasted with the standard AI view of action as planning, a detached reasoning process that operates independently of the world [Agre87b].

The background is the implicit beliefs and assumptions that inform action. Heidegger concluded that these beliefs are not amenable to representation and cannot be made explicit [Winograd87, p. 32]. In contrast, one of the long-standing goals of AI has been to explicitly represent what it calls “common-sense knowledge”, a concept that seems to correspond to the background. The difference is that common-sense knowledge is to be explicitly encoded as representations, while the background cannot be fully articulated due to its essentially situated nature. The background does not consist of explicit knowledge but in the actual situation of the creature, including its fixed environment and its adapted machinery.

These arguments are not completely convincing. There is an important unresolved question as to whether the distinction between explicit and implicit knowledge is a meaningful one. It is made more complicated by the fact that it is difficult to talk about implicit knowledge without making it explicit! The fact that phenomenologists have reached the conclusion that AI is impossible also should make us suspicious of their methods. The existence of a natural intelligence is an existence proof that an artificial intelligence is possible, there being no in-principle limits to the technological mimicking of human functioning.

The phenomenological critique of artificial intelligence can be seen as a response to a certain tendency within the field to view intelligence as solely a matter of knowledge representation and deduction. However, this tendency is not universal within the field. The Society of Mind approach, since it emphasizes mechanism over formal representation, is less subject to the phenomenological critique than most other work in AI. An agent in the society of mind is an active entity whose representational qualities, if any, are a result of its embedding in a network of other agents and its connection to the world. This way of building an artificial mind, when

fully realized, overcomes most of the objections of the phenomenologists.

2.4.2 Situated action

Traditionally, the treatment of action in artificial intelligence has centered around the notion of *planning*. The planning task is to compute a sequence of steps that will reach a specified goal. This sequence of steps is called a plan, which is then passed to another module (the executor) to be carried out. One assumption of this model is that the planner has a complete representation of the world available and the ability to compute the effects of proposed changes to it.

This notion of planning has come under attack as unrepresentative of the way in which intelligent beings actually act in the world [Suchman87, Agre87b]. We do not compute a plan ahead of time and then carry it out; instead we are in an interactive relationship with the world, continuously exploring possibilities for actions. This is what gives us the flexibility to deal with a world that cannot be fully represented in memory.

The classical notion of planning also suffers from the need to have a perfect, objective model of the world available. This perfect model is not only an accurate representation of the current state of the world, but can be manipulated to compute the possible changes induced in the world by the agent's hypothetical operations. The task of maintaining a world model over changes is called the *frame problem* [Shoham86], and severely challenges standard logical deduction methods. The planning problem itself has been shown to be computationally intractable [Chapman87]. In our view these unsurmounted problems are the consequence of the logicist framework used by planning rather than inherent obstacles to programming intelligent action.

The situated action approach advocates a more interactive relationship with the world. Action is driven primarily by current conditions in the environment and only secondarily by internal state. One of the attractions of ethology as a modeling domain is that it takes situatedness as the default case, rather than assuming that animals possess complex world models.

Agre and Chapman describe an artificial creature that operates inside a grid-like videogame world using only combinatorial logic to connect a set of sensory and motor primitives [Agre87a]. They also suggest how abstract reasoning might arise as an emergent function of situated activity [Chapman86]. Their system, called Pengi, is a player of the videogame Pengo. Pengo requires the player to manipulate a penguin in a world consisting of hostile bees and obstacles

in the form of ice cubes.

Pengi consists of a boolean logic network that computes an output (which is one of the four compass-point directions, or nothing). Its inputs come from visual routine processors that can compute moderately complex sensory functions (such as determining which objects lie in a line between the player and some target position). Visual routines can also make use of visual markers that visual routines can insert into the world. Visual markers remain across cycles of computation, and constitute Pengi's only state.

Although Pengi successfully deals with a complex and highly dynamic world, this world is rather unrealistic. The creature's actions are limited to moving in one of four directions and kicking. Pengi also has a global view of its world rather than strictly local sensing capabilities, although most sensing computations are locally centered around the player.

2.4.3 Subsumption architectures for robotics

Rod Brooks [Brooks86] and Peter Cudhea [Cudhea88] have developed architectures for robots along similar lines. As in Agar, their behavioral control system is divided into task-specific modules with fixed channels of communication between them. They base their architecture on the notion of *subsumption*. Subsumption allows low-level units to control moment-to-moment action while permitting higher-level units to set goals and take control in exceptional conditions. This technique has been applied to the design of robots that can perform simple household tasks such as vacuuming and collecting soda cans.

This notion has its basis in the presumed evolutionary history of behavior mechanisms, which evolved by generating new control layers that are built on top of and stand over the retained low-level layers. Subsumption allows a high-level unit to “stick its fingers” into the activity of the low-level units. This approach provides a design methodology that allows low-level units to be designed first with minimal modification by later requirements.

A subsumption architecture control system is made up of modules connected by wires. Modules are finite-state machines whose transitions are controlled by messages that arrive through the wires from other modules. There are architectural features that allow a module to interrupt or otherwise influence the transmission of messages between other modules; this is how subsumption is implemented. Timing constructs (see section 6.3) prove useful in this

architecture as well.

The goals of the subsumption architecture are quite similar to that of Agar. Both emphasize: a modular system for robust control of a situated creature; the basing of behavior on situated information and action; and a view of intelligence as dynamic interaction between an agent and its environment, rather than as representation and deduction.

While the basic subsumption design of communicating modules is similar to that of Agar, there are some differences. Subsumption modules are finite state machines, while Agar's modules are simple rules. Subsumption communication lines can carry typed messages, while Agar generally only passes activation strength or numerical arguments between agents. Essentially the subsumption architecture uses a slightly higher level of description, although the range of mechanisms that can be constructed is the same, since Agar can implement finite-state machines (see 4.6.1).

2.4.4 Case-based reasoning

Case-based reasoning [Kolodner85] is a method of organizing intelligence based on the use of past experience stored as cases. Case-based planners [Hammond86] generate plans by recombining pieces of earlier plans, which are then stored for future use in similar situations. The case memory indexes plans by the situations they were useful in, and by situations in which they fail. By remembering cases, a planner avoids computing new plans from scratch for each new situation. Generalizing plans in case memory can yield insight and support analogical forms of reasoning.

Case-based systems are motivated by the observation that human reasoning processes often depend upon the analogical application of previously learned examples. We find case-based systems attractive because experiences are more readily representable with agent-based techniques than are purely declarative facts. The notion of a script-agent (see section 7.2) give us the ability to remember an experience as a temporal record of agent activations. Script-agents share with case-based reasoning the notion of memory as a collection of experiences. Script-agents grounds this experiential view of memory by implementing it in a specific agent-based mechanism that is directly coupled to the sensorimotor level.

2.4.5 Implications for artificial intelligence

AI has concentrated primarily on manipulating symbolic representations, under the assumption that it was unnecessary to simulate the low-level processes that underlie the capability to represent. In abstracting away from the neural hardware, it was too easy to also abstract away the essential task of intelligence, which is generating adaptive behavior in a complex world. Instead, AI focused on the discovery of solutions to well-defined formal problems. To reconstruct an AI that is based on action in the world, we are going back to looking at the lower-level mechanisms that intelligence evolved to manipulate. These are the elements of the behavioral repertoire and their control mechanisms. Our method for doing this is to concentrate on creatures where the dependence of intelligence on behavior is more evident than it is for humans. The gap between animal behavior and human intelligence is large, but we believe that it will be more instructive to attempt to bridge it than to consider intelligence as a realm wholly divorced from action.

Chapter 3

Animal Parts

Bounce higher than a rubber ball, or be formed into animals and such.

— from the side of a Silly Putty box

An animal construction kit consists of various animal parts, tools for assembling them into working animals, objects that the animals interact with, and different worlds for the animals and objects to inhabit. The animal parts include animal bodies, sensors, motors, and computational elements. The availability, modularity and “snap-together” compatibility of parts makes experimentation easy and accessible to novices.

3.1 Worlds

A simulated animal must have a rich world to act in if it is to have a variety of possibilities for action, and hence a need for intelligence. People and computers can demonstrate their intelligence by manipulating symbols, but animals have no such ability, or only very limited forms of it. Instead, they display their intelligence by *action* in their world. The appropriateness of an animal’s actions to its situation constitutes its intelligence. Winograd [Winograd87] refers to this close relationship between an intelligent agent and its environment as *structural coupling*, and suggests that it is a fundamental to the understanding of human intelligence. This topic is discussed further in section 2.4.2.

We want our systems to run in near-real-time on present-day hardware. This constraint rules

out full-scale simulation of physical world dynamics. Instead, we must make trade-offs between realism and speed. This can be done by designing worlds and creature-world interfaces that preserve the essential characteristics and limitations of animals in the real world while bypassing computationally expensive levels of processing such as early vision and walking dynamics.

Our model worlds are abstractions of the real physical environment of the animal. They emphasize some properties of the real world at the expense of others. As a result, we need different kinds of worlds depending on the properties that we are interested in. These include worlds with standard euclidean geometry and simplified physics, more realistic worlds that can simulate locomotion (such as balance and limb placement) in greater detail [Sims87], more abstract worlds such as cellular arrays [Turner87, Rizki86], or completely non-physical game-theory worlds for simulating theoretical problems of behavioral strategy [Axelrod84].

The world, sensors, and motors together constitute an implementation of the animal's *Umwelt* or subjective world. *Umwelt* is a term introduced by von Uexküll to denote the unique private world defined by an animal's sensorimotor capabilities. Describing the limited world of a tick, he says "the very poverty of this world guarantees the unfailing certainty of her actions" [Gould82, p. 27]. The *Umwelt* forms the basis for situated action in animals.

There is also the possibility of bypassing the simulation problem entirely by means of real-world robotics. This approach is strongly advocated by Brooks [Brooks86] (see section 2.4.3) and brought to pedagogical reality in the Lego/Logo system [Ocko88] (see section 2.2.1).

Since robot technology has not yet advanced to the point where one can take the mechanical details for granted, it is difficult to do behavioral research with robots. Simply dealing with the robot's mechanical details can consume all of one's research energies. Also, some of the abilities of animals are far beyond our ability to simulate them. If we wanted to simulate ant odor trails, for instance, we would have to design a chemical deposition and detection system. Ants can carry less than a nanogram of this chemical and can detect single molecules. If we were to try and substitute an equivalent non-chemical mechanism, such as an optically-detected trail, we would have to also design a way for the marked trail to evaporate.

The worlds used in the examples are all two-dimensional euclidean worlds with a continuous coordinate system. This is in contrast to most other animal simulation systems (ie, [Turner87] [Coderre88] [Wilson86]) that use discrete grids. I found that grids failed to capture certain

important properties. For instance, the orientation of an animal is an important consideration in the real world, given the directionality of its sensorimotor capabilities, and this property cannot be captured by grid systems.

The lack of a third dimension is not a problem in simulating ground based animals. Somewhat satisfactory simulations of flying, swimming, or tunnelling can be created by making one of the two screen dimensions stand in for the vertical dimension.

3.2 Physical objects

Worlds contain objects that creatures interact with. These are called *physobs* for *physical object*. Creatures themselves are a specialized class of *physobs*, but there are also simpler *physobs* such as walls, rocks, and food particles, that the creatures can sense and interact with.

3.3 Creatures

Creatures serve as mobile frameworks for their sensors, motors, and agents. They hold physical state such as position and orientation. Motors and sensors act through the creature object. Creatures are also responsible for interfacing to the display system. They may contain other state specific to their species (such as an individual ant's caste).

The basic design of creatures in the two-dimensional world is based on the Logo turtle [Papert80] and its autonomous antecedent [Walter50]. They implement a form of local computational geometry. The locality is critical: all the turtle's operations are expressed as relative to its present position and orientation, rather than to a global coordinate system. This is appropriate for simulating animals, all of whose information and action must necessarily be local.

Creatures are also responsible for computing physiological constraints on what they can do. This includes keeping track of energy inputs and outputs, as in the evolution simulation (see section 7.3).

3.4 Sensors

Creatures use sensors to obtain information about the world. Sensors make up half of the interface between the creature and the world (motors are the other half). Sensors have only local knowledge about the world, and this property is a key point of our methodology. We are building situated creatures, and we enforce the situatedness by means of the creature/world interface.

Our sensors are abstractions of the sensors found in real animals, such as eyes or ears. Rather than simulate the physics of light, sound, or smell, we abstract the functional capabilities of sensors at a level appropriate to the behavioral control system. This generally means that sensors detect a particular kind of object within a given range of the creature. Sensors can be directional, relative to the orientation of the creature. Sensors may return a boolean value indicating the presence of an object, or a numerical value dependent upon the number of objects present and their properties. In the latter case they may be weighted by closeness. For a detailed description of a general sensing function, see Section 4.4.

The notion of perception as detecting specifically useful classes of object was anticipated by Gibson in his notion of ecological optics [Gibson79]. Gibson believed that the function of perception was to extract relevant information from an environment, and that animals sensory systems were specialized to extract the information they required. While there are many problems with Gibson’s analysis, most stemming from his failure to apprehend the computational mechanisms required to implement his ideas [Marr82, pp. 29-31], his fundamental insight was sound, and has been supported by later neuroethological research [Ewert87] that demonstrates specialized neural pathways for behaviorally relevant information.

Sometimes the rule about giving sensors only local access to the world must be violated. For example, the **nest-vector** sensor in the ant simulation provides the ant with the direction of its nest, no matter how far away the ant is from the nest. In this case, the oracular knowledge is justified as a stand-in for simulating the details of the ant’s true navigational abilities, which depend upon a combination of sun-compass navigation, landmark learning, and integration of the outward trip [Wilson71, p. 216]. The oracle is an abstraction for the ant’s true abilities rather than a cheat. The goal of the sensorimotor interface language is to force the user to explicitly declare such abstractions, and so justify them.

3.5 Motors

Motors are the other side of the creature/world interface. A creature acts on the world through motors. At our normal level of simulation, a motor usually performs a movement of the entire animal (such as a turn or forward motion). A more detailed simulation might use motors that move individual limbs or muscles. In some of the systems, motors are specialized agents. In others, they are a separate class of objects that agents can activate.

Motors are also responsible for enforcing limitations on the animal's actions. For example, a **forward** motor should enforce a limit on the animal's maximum forward speed. If the animal's energy usage is being modeled (as it is in the evolution simulations; see Section 7.3) the motor agents are responsible for deducting the creature's energy usage from its reserves.

Motor actions can fail. The motor agent attempts to perform an action by calling appropriate methods of the world object. If the world object decides this action is impossible, it can signal an error. For example, trying to move into a solid object would generate such an error, as would attempting to pick up an object that had already been removed by another creature. It is possible to make the error indication available to the creature as a sense. For example, if the **forward** motor fails, it may be due to an obstacle blocking the creature's way, and so we might use the error signal to trigger a turn.

Since the failure of a motor action will usually leave the current situation unchanged, the action is likely to be repeated. This might be a good strategy if continued repetitions are likely to lead to an eventual success. If not, then this will lead to an infinite series of futile attempts. Self-inhibition (see section 6.3) can be used to limit the amount of time a creature will spend on an activity.

3.6 Computational elements

Once a creature's physical capabilities and limitations have been determined by defining its complement of sensors and motors, the creatures's behavior must be programmed. In keeping with the kit metaphor, a program is assembled by selecting partially prefabricated parts and connecting them to sensors, motors, and each other. These parts might be neurons, rules, or agents. All can perform simple computations, some can encapsulate state. The detailed

descriptions of these elements are given elsewhere. Here, we discuss the loose implications and metaphors suggested by them.

These computational systems all employ parallelism. This is a consequence of the kit metaphor and the goal of biological realism, rather than a desire for speed. In the kit programming metaphor, a computation is specified by plugging parts together to form a network of nodes and wires, which then runs in parallel. As in a biological intelligence, there is no central clock to enforce synchronization between components.

The computational elements communicate with each other over wire-like channels. The component and wire metaphor lends itself to a graphic interface (section 2.2.2). Only the earliest system, BrainWorks, actually included such an interface, since the later systems moved away from the simple activation-passing model used in BrainWorks.

We now consider what sort of computational elements should be in a kit. *Neurons* have a long history as hypothetical components of psychological and computational systems (going back at least to Freud [Freud95]). These neurons are simplified versions of the neurons actually found in brains, envisioned as analog, linear, sum-and-threshold devices. Connections between neurons are weighted, and neural systems can learn by changing these weights.

Rules operate on symbolic data. They cause an action to be performed whenever some condition is met. The condition is usually a boolean combination, but may be a more complicated function on a large database which involves pattern-matching and variable binding. A rule's action part can be an arbitrary action, but often includes adding new symbolic structure to a database. It is easy to construct universal computation devices using rules together with an appropriate environment.

An *agent* is a simple computational process that performs a particular function, often by turning on other agents. This notion of agent derives from Minsky's Society of Mind theory [Minsky87]. Considered as computational objects, agents can:

- Execute concurrently,
- Maintain a small amount of local state,
- Access the state of other agents through connections, and
- Take actions automatically when environmental conditions are met.

In the following chapters, we will see several attempts to define agents as simple yet powerful computational modules that can be combined to give rise to behavioral control mechanisms.

Chapter 4

3-Agar: Agents as Gated Rules

Go to the ant, thou sluggard; consider her ways, and be wise: Which having no guide, overseer, or ruler, provideth her meat in the summer, and gathereth her food in the harvest.

— Solomon

*Splish splash I was raking in the cash
The biology of purpose keeps my nose above the surface (OOOH)*

— Brian Eno, *King's Lead Hat*

This chapter describes the final form of the animal construction system, called 3-Agar. In 3-Agar, animal minds are built out of agents, implemented as gated if-then rules. An agent consists of a condition (if part) and action (then part), plus a gating mechanism that decides when the rule is to be run. The condition part of a rule is typically a sensor predicate, and the action is either a motor function or a command to activate or inhibit other agents. The gating mechanism handles matters of timing control and activation strength.

4.1 Language basics

This section describes the basic language constructs available in 3-Agar. The process of building a simulation consists of choosing a world, defining creatures (including their sensorimotor capabilities and appearance) and defining agents for the creatures.

3-Agar uses Symbolics New Flavors [Symbolics88], an object-oriented programming package. For the most part the implementation details are invisible to the user, although there are mechanisms for accessing the underlying flavor mechanisms if necessary.

4.1.1 Creature definition

New classes of creatures are defined by a `defcreature` form. This is a Lisp macro of the form `(defcreature name components &rest keys)`. *Name* is the name of the creature-class being defined. *Components* is a list of the basic object classes that the creature is built on top of (typically `2d-creature` for our two-dimensional world). *Keys* is a list of alternating keywords and values, selected from the following list:

:sensors The value is a list of sensor definitions of the form `(sensor-name sensor-type &rest args)`. *Sensor-type* is the class of the sensor, and *args* are keyword/value pairs that are passed to the object creation function.

:slots The value is a list of slot-names or slot-name/initial-value pairs. Each creature of the class is given these slots, which can be accessed by its agents.

:defflavor-options The value is a list of alternating keywords and values to be passed on to the flavor defining form.

An abbreviated example of a creature definition:

```
(defcreature ant (2d-creature)
  :slots ((thing-held nil)
          (caste nil))
  :sensors ((wall-sensor
             simple-sensor :sense-function 'wall-in-front)
            (long-range-food-sensor
             general-sensor :range-type :circle :class 'ant-food
             :range 150)
            (left-food-homing-sensor
             general-sensor :range-type :cone :class 'ant-food
             :range 75
             :view-angle 30
             :body-angle (- 30))
            (nest-sensor
             general-sensor :range-type :circle :class 'nest :range 25))
```



```

...))
:defflavor-options ((:initable-instance-variables caste)))

```

For example, this means that an ant is equipped with a **long-range-food-sensor** whose range is a circle of radius 150 units and is sensitive to objects of the class **ant-food**. The sensor parameters are described in more detail in Section 4.4.

Motor functions are defined as methods for the creature that can be called by agents. These methods generally will make calls to the world object and implement the creature's motor relationship to the world. Given the diversity of possible worlds, these functions are not standardized. A typical example:

```

(defmethod (forward ant) (amount)
  (let ((xx (+ x (* amount (~cos heading))))
        (yy (+ y (* amount (~sin heading)))))
    (if (in-bounds world xx yy)
        (drawing-creature
         (setq x xx y yy)
         (signal-motor-failure 'forward))))))

```

This motor function implements the ant's forward-motion capability in its two-dimensional world. It also constrains the motion so that the ant is not allowed to move outside the boundaries of the world.

A creature definition also includes a **draw** method that specifies its appearance on the screen.

4.1.2 Agent definition

In 3-Agar, an agent definition consists of a **defagent** form that associates the agent with a particular creature and specifies its behavior. This definition is of the form:

```

(defagent (creature-name agent-name)
  {keyword value}*)

```

A **defagent** form will include a number of keyword/value pairs. Some of these specify that the agent is to use a *timing construct* that modifies the activity of the gating function. For a detailed explanation of timing constructs, see section 6.3. Others specify the condition or action part of a rule or other agent properties.

Agents are single rules in 3-Agar, but often we would like to be able to think of a group of functionally-related rules as a unit. To allow this, the agent-definition language has syntax for defining clusters of agents with a single form. An agent definition can include definitions of *subagents* that are like regular agents in all respects save having a name. This means they cannot be activated or otherwise referred to from other agents. Instead, they are activated when the action part of the parent rule is run, via an implicit **a+** call. Subagents are purely a syntactic convenience and add no new functionality to the language.

The possible keywords are:

- :if** specify the condition part of the agent. The default value is **t**, that is, the condition is always met.
- :do** specify the action part of the agent. It can be a single Lisp form or a list of forms, which is treated as if surrounded by an implicit **progn**.
- :bias** specify the activation bias. The default is 0. A positive bias will cause the agent to run without explicit external activation (unless external inhibition intervenes).
- :subagents** The value is a list of subagent specifications. Each subagent specification is a list of keyword/value pairs from the same set as the top-level defagent. When subagents appear, forms that activate them are implicitly added to the **:do** clause of the containing agent.
- :latency** specify timing construct information as a numeric value or **nil** (the default).
- :hysteresis** specify timing construct information as a numeric value or **nil** (the default).
- :self-inhibit** specify timing information that affects agent activation.
- :trace** For debugging purposes only. If non-**nil**, causes a trace entry to be made when the agent is run.

4.1.3 Value-returning forms

The **:if** clause of a defagent is usually a boolean expression whose primitives are **sense** forms.

A **sense** form accesses the current value of a sensor:

```
(sense sensor-name)
```

An agent can also access its creature's slots using the **slot** form:

```
(slot slot-name)
```

4.1.4 Action-producing forms

The `:do` clause of a defagent form specifies the action to be taken when the agent is active and the `:if` clause is satisfied. The value of this clause is a list of forms to be executed. These are usually either calls to the motor functions of the creatures, or calls to functions that activate or inhibit other agents. These functions are:

- a+** (**a+** *agent* [*amount*] *arg**) increases agent *agent*'s activation by *amount* (which defaults to 1). Any other arguments present are handled according to the argument passing conventions described below.
- a-** Similar to **a+**, but decreases the activation rather than increasing it. An amount can be specified (the default is 1) but no arguments can be passed.
- a0** Puts the agent into its default state by zeroing activation and resetting any timing information.

4.2 Argument passing between agents

While most cases of agent-to-agent communication can be handled by a simple activation, sometimes the transmitting agent needs to be more specific about what it wants the receiving agent to do. This is especially true when the receiver is a motor agent. Motor agents take action in the world and often need to adjust these actions to local conditions. 3-Agar introduces a method by which agents can pass parameters as arguments to other agents. Since an agent might be activated by more than one transmitting agent in a single cycle, we include a flexible scheme for resolving conflicts and combining arguments.

For example, the **turn** motor agent takes an angle as an argument. If it is activated by more than one other agent during a cycle, it must combine these activations in some way. One way is to accept only the first activation that comes in, ignoring all later activations. Another method is to label activations with a special priority argument, and only use the activation with the highest priority. Yet another method is to combine all the calls via summation or some other combining function.

To implement a scheme for flexible argument combination, the basic activation form, **a+**, was modified so that it could take extra arguments in addition to activation strength. These extra arguments are interpreted according to the `:arguments` clause in the target agent's definition.

The `:arguments` clause contains a list of argument specs. An argument spec is either a symbol or a list of a symbol and a keyword. When there are multiple activations of the agent on a single cycle, the keywords control how the activations are combined. The variables in the `:arguments` list are bound and accessible to both the `:if` and `:do` clauses. The argument spec keywords are:

`:priority` Choose the activation call that has the highest value of this argument.

`:first` Choose the first activation call that has a non-null value for this argument.

`:sum` Sum this argument across activations.

There are some constraints on which keywords can be used. There can be at most one argument that specifies either the `:priority` or `:first` keyword, and if either of those are specified then the `:sum` keyword cannot be used.

For example, this agent definition indicates that the agent will sum its argument:

```
(defagent (ant turn)
  :arguments ((direction :sum))
  :do (right creature direction))
```

This agent will choose the activation that has the highest value for the `priority` argument, and go forward by the amount specified in the corresponding `distance` argument:

```
(defagent (ant forward)
  :arguments ((priority :priority) distance)
  :do (forward creature distance))
```

4.3 The gating mechanism

An agent consists of a rule and a gate. The gate modulates the activity of the rule by controlling when it is allowed to be active. The gating implements handles timing constructs (see section 6.3), activation level management, and argument combination (see section 4.2).

Each simulation cycle has two parts. The first, the pre-cycle, takes care of updating each agent with the newly-computed activation level from the previous cycle. Arguments are also

passed and conflicts between them resolved during this phase. The main cycle then operates by calling each agent's gating function. The gating function updates various timing counters, determines if the agent is to be run on this cycle, and runs it if appropriate.

The running of an agent consists simply of evaluating the condition part of the rule and then, if the condition returns a non-`nil` value, evaluating the action part. Thus, an action will be run if and only if the gate runs the rule and the condition part of the rule evaluates to a non-`nil` value.

4.4 Sensing

3-Agar included a general sensing function that could be customized for most purposes in the two-dimensional world.

A `general-sense` method for the class `2d-creature` was defined that took the following keyword arguments:

:range-type Specify the general shape of the sensitive range. Possible values are `:circle`, `:segment`, `:square`. The `:square` shape has no particular physical justification, but is fast to compute.

:range The radius of the sensitive range (or the half-side-length if the value of `:range-type` is `:square`).

:class Specify the class of objects the sensor responds to.

:object-type Specify which object to base the returned value on. Possible values include:

:any return any object of the appropriate class within range.

:closest return the closest object of the appropriate class within range.

:centroid computes the centroid of all appropriate objects within range, creates a pseudo-object to mark the point, and bases the returned value on it.

:result-type Specify the type of result returned. Possible values include:

:predicate Return `T` if there is an appropriate object in the sensitive range, `nil` otherwise.

:heading Return a vector heading to the sensed object.

:distance Return the distance to the selected object.

Figure 4.1: A generalized sensor

:object Return the selected object itself. This is considered “cheating”, as giving the creature direct access to a world-object violates the sensorimotor protocol.

:view-angle When **:range-type** is **:segment**, this specifies the viewing angle.

:body-angle When **:range-type** is **:segment**, this specifies the angle the center of the sensitive field forms with the body axis of the creature.

This function suffices to implement all the sensors used in the ant trail recruitment example, except for the trail sensors. These are simulating sensors at the ends of the ant’s antennae, and so have a sensitive range that is a small area, displaced from the creature’s position by a short vector.

4.5 Example: ant foraging and recruitment

Ant colonies can coordinate the activity of their inhabitants in many ways [Wilson71]. One of the most striking methods of coordination is *food recruitment*—the process by which a foraging ant who has located a large source of food will lead other ants to the source to help carry it back to the nest. Highly-developed species of ants do this with odor trails.

A forager who finds the food will return to the nest, and during the journey will mark a trail on the ground with a volatile chemical of distinctive odor. It does this by periodically pressing its stinger onto the ground. Other ants are extremely sensitive to the smell of this

Figure 4.2: Ant following a trail in nature and in Agar (from [Wilson71])

chemical, and if they sense it will start to follow the trail to the food source (see figure 4.2). The trail apparently does not encode directional information, so the ants must use other cues, such as the sun, to determine the correct direction on the trail. In the most common case, the ants that are aroused by the trail chemical are waiting inside the nest, and so have only one choice (the correct one) of direction.

When the recruited ants reach the food source they too mark the trail when they return to the nest. Thus the trail is continuously reinforced so long as the food source remains. When the food is gone, the ants no longer mark the trail, and the highly volatile chemical evaporates. The process is thus both self-reinforcing and self-limiting.

4.5.1 Smelly worlds

Simulating trail recruitment requires a world that can support odors that are spatially localized and evaporate over time. While this could have been done by treating each odor spot as a separate world object, this would have been inefficient. Instead a coarse array is laid over the world. Each element of the array can contain a number of smells that are tagged by their time of deposit and initial strength. When an ant smells a particular array element, the effective strength is computed as a function of the smell type, the initial strength, and the elapsed time. A graphic indication on the screen indicates the presence of a significant amount of smell. The system supports multiple types of chemical cues with their own dispersion and decay rates.

4.5.2 Foraging

Foraging ants explore the world for food. In the simulation, this foraging system is independent of the trail-recruitment mechanism. The sensors involved are a long-range non-directional food sensor that reports on the presence of food within a given radius from the ant, and two shorter-range, directional food sensors that face forward on either side of the ant (diagram). The foraging strategy involves three stages: random search, localization, and final approach. The stages involve different combinations of agent.

Random search is the default behavior of a foraging ant. It is controlled by three agents: `basic-movement`, `basic-forward`, and `direction-randomizer`.

```
(defagent (ant basic-movement)
  :latency 10
  :bias 1                                ; On by default
  :do ((a+ basic-forward)
        (a+ direction-randomizer))
  :subagents ((:if (sense wall-sensor)
                   :do ((a+ forward 5 0)
                       (a+ about-face)))
              (:if (sense ant-in-front-sensor)
                   :do (a+ forward 4 0))))

(defagent (ant basic-forward)
  :latency 10
  :do ((a+ forward 1 20)
        (a- direction-randomizer))      ; Suppress turning
  :subagents ((:if (> (random 20) 18)
                :do (a- basic-forward))) ; Suppress self

(defagent (ant direction-randomizer)
  :do (a+ turn (arand 0 45)))
```

These agents work together to produce a motion that consists of a series of forward darts punctuated by periods of turning. This alternation is managed by `basic-forward`, which will be active until it suppresses itself and allows `direction-randomizer` to run. The turning produced by the `direction-randomizer` agent causes the ant to sweep its sensors over a larger range.

When the ant blunders into `long-range-food-sensor` range of a food particle, it enters the localization phase. The goal now is to bring the food in range of the short range food

sensors. New agents are triggered by the long range sensor. These could suppress the agents of the earlier phase, but instead they are left on to continue the random search. The search is constrained by the new agents, which enforce an additional rule: if the ant moves out of range of the food particle, do an about-face. This effectively narrows the search area. Since this rule is triggered by an on-to-off transition of the long-range food sensor, we must use an agent with hysteresis to “catch” the transition:

```
(defagent (ant food-finding)
  :bias 1
  :if (not (slot thing-held))
  ;; Activate homing and trail agents
  :do ((a+ home-in-on-food)
      (a+ find-recruitment-trail))
  ;; Detect food appearing on long-range sensor
  :subagents ((:if (sense long-range-food-sensor)
                  :do (a+ food-search))))

;; If long-range sensor goes off, do an about face
(defagent (ant food-search)
  :hysteresis 20
  :if (not (sense long-range-food-sensor))
  :do (a+ about-face))
```

Once the food is in range of the short range sensors, the remaining steps are straightforward. The ant slows down so as not to overshoot the food, and corrects her course based on the two directional food sensors.

```
(defagent (ant home-in-on-food)
  :if (or (sense left-food-homing-sensor)
          (sense right-food-homing-sensor))
  :do ((a- direction-randomizer)
      (a- follow-recruitment-trail)
      (a- food-search)
      (a+ forward 1 6)) ;slow down when homing
  :subagents ((:if (sense left-food-homing-sensor)
                  :do (a+ turn -10))
              (:if (sense right-food-homing-sensor)
                  :do (a+ turn 10))))
```

4.5.3 Recruitment

Recruitment includes the two processes of trail laying and trail following. An ant who has gathered up food makes her way back to the nest, orienting by an unknown combination of factors that probably includes sun-compass navigation and the following of local landmarks. In the simulation we use a nest-vector oracle that gives the ant knowledge of when she is approximately oriented towards the nest. To some extent this violates our rule of giving the ant only local access to the world, but we take pains to justify this magic capability as standing in for an unknown, but real, local mechanism.

Trail laying agents are activated by the ant having a piece of food in its pincers. This is a good illustration of the idea of using physiological state to determine behavior, taking the burden of state manipulation out of the behavioral control system (see section 6.1).

Recruitment trail following works via two antennae-like sensors that detect the trail chemical. The ant tries to keep these sensors on either side of the trail as she moves forward, making corrections as necessary.

Recruitment agents are initially activated when the trail sensors pick up the trail. The most difficult part of trail following is the initial phase in which the ant tries to align herself with the trail. Once the ant is straddling the trail with its trail sensors, her movement is controlled by the `follow-recruitment-trail` agent, which simply turns in the direction of any or both trail sensors that become active.

```
(defagent (ant follow-recruitment-trail)
  :hysteresis 20
  :do ((a- school)
        (a- basic-movement)
        (a0 food-search)
        (a+ forward 3 5)
        (a+ trail-right-face))
  :subagents ((:if (sense left-food-recruit-sensor)
                    :do (a+ turn 5)
                    :hysteresis 5)
              (:if (sense right-food-recruit-sensor)
                    :do (a+ turn -5)
                    :hysteresis 5)))
```

When the ant reaches the food, the food homing agents will take over. If the end of the

trail is reached without coming across food, the ant will resume her default activities after the trail-following agent's hysteresis decays. Usually when this happens it means that the food supply has been exhausted, although it is common for ants to fall off the trail in both nature and the simulation.

4.6 Other examples

Here we demonstrate that Agar is capable of embodying some common behavioral techniques, in particular *oscillators*, which appear in walking control systems; and *finite-state machines*, which form the modules of Brooks' subsumption architecture (see 2.4.3).

4.6.1 Oscillators

Oscillators or rhythm generators play an important role in the control of periodic animal motion such as walking [Pearson76]. The following two agents define a creature named **wiggler** that has an oscillating movement. The period and duty-cycle are controlled by the **:self-inhibit-after** parameters. Each agent inhibits the other with **a0**. The **:latency** clauses ensure that after one of the agents has inhibited itself, it will remain dormant long enough for the other agent to take control.

```
(defagent (wiggler wiggler-l)
  :bias 1
  :latency 2
  :self-inhibit-after 10
  :do ((a0 wiggler-r)
        (left creature 10)))

(defagent (wiggler wiggler-r)
  :bias 1
  :latency 2
  :self-inhibit-after 10
  :do ((a0 wiggler-l)
        (right creature 10)))
```

Figure 4.3: A finite state machine diagram

4.6.2 Finite state machines

Finite state machines can be implemented in Agar by creating a creature state variable to hold the state and creating an agent for each possible state transition. The following example implements a three-state machine (see figure 4.3) that causes the creature to perform a sequence of right turns on every true-going transition of `trigger-sensor`:

```
(defcreature fsm-creature (...
  :slots ((state :start)))

(defagent (fsm-creature u-turn)
  :bias 1
  :hysteresis 2
  :if (sense trigger-sensor)
  :subagents
  ((:if (and (eq (slot state) :start)
             (sense trigger-sensor))
    :do ((right creature 90)
          (forward creature 10)
          (setf (slot state) :turn1)))
   (:if (eq (slot state) :turn1)
    :do ((right creature 90)
          (forward creature 10)
          (setf (slot state) :turn2)))
   (:if (and (eq (slot state) :turn2)
             (not (sense trigger-sensor)))
    :do (setf (slot state) :start))))
```

4.7 Comparison with other designs

One advantage of 3-Agar over the earlier rule-based system (1-Agar; see section 5.3) is a finer granularity (only one rule per agent), allowing a higher degree of parallelism. Agent behaviors in 1-Agar were specified with a block of Lisp code that usually contained a series of if-then rules. Because the rules were evaluated serially, the results were dependent upon the ordering of the rules, introducing hidden state in violation of our methodology. 3-Agar eliminates this problem.

Since 3-Agar represents a rule as a structured object with separate if and then components, rather than an unstructured block of code, it is easier for learning agents (see Chapter 7) to perform operations on other agents.

The 3-Agar language incorporates both pushing and pulling constructs (see section 6.2). The `:if` parts of rules pull values from sensors, while the `:do` parts push activation to other agents. This seems like a natural division of labor among the two modes of communication. In this scheme, an agent can be thought of as a locus where intention (pushing) meets with information-gathering abilities (pulling) to produce controlled action. It also gives us a close match to Tinbergen's unified model of ethological systems (see section 2.1.1), with the pushing of agent activation corresponding to drive relationships, and the pulling of values from sensors corresponding to innate releasing mechanisms.

4.8 User Environment

Various user interface tools were developed for Agar. While they do not yet constitute a complete novice programming environment, they are steps towards that end. They include:

- the ability to select a particular creature with the mouse. Subsequent commands and debugging displays would then pertain to the selected creature.
- the ability to start, stop, and single-step the agents.
- a continuously-updated display of the internal state of the selected creature or one of its agents.
- a facility for displaying the sensitivity range of individual sensors (as a circle surrounding the creature, for example).

- a facility for manually disabling particular agents.
- a facility for displaying and editing an agent's definition while the creatures are running.

Agent definitions are edited with the normal Lisp Machine tools, and can be recompiled incrementally.

Chapter 5

Early Designs

*I must Create a System, or be enslav'd by another Man's
I will not Reason & Compare; my business is to Create*

— William Blake

On these remote pages it is written that animals are divided into (a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.

— Jorge-Luis Borges, *Other Inquisitions*

This chapter describes some earlier implementations of animal construction kits.

The first system, BrainWorks, emphasizes a graphic programming interface that uses a neural net metaphor. The basic structure of the sensorimotor interface between creature and world was developed in BrainWorks and formed the basis for the later systems.

Several early versions of Agar are also described in this chapter. In Agar, I abandoned graphic programming in favor of concentrating on simulating complex behaviors taken from real ethological examples, and developing more complex languages for describing agent-based behavioral systems.




Figure 5.1: The turtle-wiring window from BrainWorks

5.1 BrainWorks: graphic neural programming

BrainWorks is a computer system that allows a user to construct a nervous system for a simple animal, using an interactive graphic interface. The user starts out with an animal body that resembles a Logo turtle equipped with several sensors and motors. The sensors include eyes and touch bumpers that respond (respectively) to food and to obstacles in the animal's world. The user also has a supply of neurons of various types, and tools for bringing them into the turtle's body and connecting them to the sensors and motors (see figure 5.1). When the turtle is placed into its world, it responds according to its wiring.

BrainWorks was largely inspired by Valentino Braitenberg's *Vehicles: Experiments in Synthetic Psychology* [Braitenberg84]. In addition to allowing users to program animals, BrainWorks can be used to simulate evolutionary adaptation of animals to their environment. This work is discussed in section 7.3.

A turtle's usual task is to catch food while avoiding being blocked by a wall or obstacle (see figure 5.2). This can be accomplished by a variety of networks and parameter combinations. There are tradeoffs in the values of the parameters such as **turn-factor**: a small value will result in a large turning radius, making it hard to home in on nearby food, but a large

Figure 5.2: The turtle moving in the BrainWorks world

value may result in food being missed due to oversteering. This interacts strongly with the `eye-angular-aperture` parameter, indicating the importance of matching sensors to motors, and appropriate coupling of both to the environment.

The world of BrainWorks is very simple, but still complex enough to generate interesting behavioral problems. Essentially it is a 2-D plane, with objects taking on a continuous range of positions (in contrast to worlds that use a grid or other discrete geometry). The objects in it are turtles, food units, and obstacles. Turtles are dimensionless, but have an orientation. Food units, on the other hand, have a non-zero radius so that the turtles will be able to intersect with them (and so eat them). Obstacles are rectangles into which the turtle cannot move. Any attempt to move into them results in no linear motion on that simulation cycle, and activates the turtle's touch bumper.

The properties of the world and the turtle were the result of compromises between the goals of keeping computation time low (to preserve the real-time feel of the system) and making a rich, realistic, and interesting world.

5.1.1 Neurons

BrainWorks uses modified McCulloch-Pitts neurons [McCulloch43] as its computational units. They implement a mostly boolean logic, in that neurons can be in one of only two states (on or off), connections similarly transmit a boolean value, and there are no registers. Connections can be inhibitory or excitatory. A neuron's activation is computed by summing up the number of activated connections, weighted appropriately (inhibitory connections have a negative weight). If the result is over a fixed threshold (usually zero), the neuron is considered activated.

The set of neuron types includes interneurons (which implement the threshold logic described above), the sensorimotor neurons that serve to interface the nervous system with the world, and two special types: pulser and delay neurons. A pulser acts as a constant. It is always activated, and can be used to provide default behaviors or biasing of other neurons. The delay neuron works like an ordinary interneuron, except that its output is based on the input values from the *previous* simulation cycle. Chaining delay neurons allows building delay lines of arbitrary length. This feature enables the turtle to control the timing of its reactions, but removes the nervous system from the class of strictly stateless machines.

BrainWorks turtles have three sensors: two eyes and a bumper. Eyes detect food within a conical area of sensitivity that is controlled by several parameters: **eye-angular-aperture**, **eye-linear-range**, and **eye-binocular-angle** (the latter controls the angular offset of each eye from the turtle's body axis). The bumper detects an obstacle or wall in the turtle's path.

Motor functions in Brainworks are implemented by *motor neurons*. There are three motor neurons, for moving forward, right, and left. Right and left movements change orientation only, while forward causes a translation along the current body axis. All the motors can be activated simultaneously, but left and right will cancel each other out. The magnitudes (angular or linear) of the movements are controlled by user-settable parameters (**turn-factor** and **forward-factor**), but are always constant for a given turtle. Movements take place in discrete time steps, but in a continuous coordinate system.

5.2 0-Agar: connectionist agents

The first implementation of Agar (called 0-Agar) is a direct descendant of BrainWorks, and is also influenced by my earlier Society of Mind implementation called SoMPL (Society of Mind Programming Language). It uses an activation spreading or connectionist model. The concept of an agent includes sensor and motor units as well as “intermediate” units. Agents have activation level that can range from 0 to 1, and weighted connections that determine their influence on other agents.

Sensory agents have their activation level determined by their sensors. Other agents’ activation is computed in terms of their connections and their hysteresis according to the equation below. Thus an agent’s activation level decays towards 0 over time without external excitation.

The activation of agent a at time t is given by the formula:

$$A_{a,t} = h_a \sum_{(w,a') \in S_a} w A_{a',t-1} + (1 - h_a) A_{a,t-1}$$

where h_a is the hysteresis for agent a , and S_a is the set of “synapses” that specify weighted connections from other agents. A synapse is a pair (w, a') where w is the weight and a' is the presynaptic (driving) agent.

0-Agar avoided the pushing/pulling question (see Section 6.2) by specifying connections separately from the agent definitions, gaining a degree of balance at the cost of having the worst of both worlds: an agent-definition did not contain the names of agents on either its input or output side.

The disadvantages of this system are quite glaring. Expressing complex boolean relationships requires them to be reduced to a network of weighted connections, and usually involves the introduction of extra agents. While this process can be automated, it seems more natural to replace the connectionist hardware with something more powerful.

The chief monitoring tool for 0-Agar is a bar-graph that can display the activation levels for a creature’s set of agents (see figure 5.3). The bar graph can also be used as a control. Mouse commands allow the user to alter an agent’s activation manually or lock an agent at a particular activation level.




Figure 5.3: The Bar Graph display/interaction window from 0-Agar

5.3 1-Agar: memories and rules

A key issue in programming language design is achieving a intimate relationship between actors and the objects they act upon. Lisp, for instance, owes much of its superiority to the fact that its programs and its data share a good deal of common structure. Object-oriented programming's benefits lie in its re-organization of program fragments so that they are associated with the data structures they manipulate.

The agent-based systems described here do not face this issue because they do not manipulate data in the usual sense of the word. That is, they do not store structures of pointers and bits and define their computations in terms of operations upon these structures. Instead, they are activated or inhibited by each other or by events in the world, and in turn manipulate other agents, eventually affecting the world. Data is not stored in registers. Instead, new concepts are encoded by creating new agents that are connected via activation channels to the existing network.

Programming in such a style can be difficult and unnatural when compared to standard computer languages. 1-Agar is an attempt to reinsert some of the normal features of standard programming languages, such as state, into an agent-based language. It does this by introducing *memories*, which are named objects internal to an animal that contain Lisp objects. Agents then become production rules triggered by the presence of tokens in particular memories and act by depositing tokens in other memories.

In this system, sensors and motors are implemented as special classes of memories whose contents are respectively controlled by or control events external to the creature. A sensor contains a token if and only if it detects its trigger condition in the world, and this token can then trigger rules. A motor is activated when a rule deposits a token into its memory. Calling these “memories” is something of a misnomer, since they do not store their contents between simulation cycles.

Memories reintroduce state as a major feature of the language. The most important justification for this is to allow rules that are triggered by the presence of active goals in a goal memory. This permits a GPS-like programming style whereby the agent compares the current state to a goal state and generates a difference-reducing action [Minsky87, p. 78]. Another motive for re-introducing state is to allow the rules themselves to be stored in memory and thus be mutable by learning algorithms that can themselves be expressed as rules.

In 1-Agar, agents are sets of rules. The agent level serves only to organize rules for the programmer; no control is available at the agent level. On each cycle, all the rules attempt to fire. Agents do not directly influence each other. Instead, they communicate by means of the memories.

It should be noted that 1-Agar differs from normal production systems in an important way. Whereas the firing of a production rule is interpreted as an *inference*, the firing of an Agar rule is an *action* in a dynamic system. This distinction reflects the close coupling of the rules to the world. Also, traditional production rule systems include a range of constructs for matching patterns and binding variables, which are neither present nor necessary in 1-Agar.

The predicate part of rules consist of boolean combinations of memory queries. The queries take the form (**m?** *memory-name* [*token*]). The **m?** form indicates a memory query, and has a value of true if the named token (or any token, in the case where the argument is not given) is present in the named memory. A rule’s action part consists of a series of forms:

m+ : insert a token into memory

m- : remove a token from memory

m0 : clear a memory

A sample rule set for ant food-recruitment follows.

```

(defrules (ant food-tracking)
  ;; If long-range sensor picks up food, try to locate it
  (if (m? long-range-food-sensor)
      (m+ goals 'locate-food))
  ;; If the sensor shuts off, do an about face
  (if (and (m? goals 'locate-food)
           (not (m? long-range-food-sensor)))
      (m+ turn '(right ,(arand 180 20))))
  ;; Home in on food if it appears in directional sensors
  (if (and (m? goals 'locate-food)
           (m? left-food-homing-sensor))
      (m+ goals 'home-in-on-food)
      (m+ turn '(right -10)))
  (if (and (m? goals 'locate-food)
           (m? right-food-homing-sensor))
      (m+ goals 'home-in-on-food)
      (m+ turn '(right 10))))

```

The memory approach proved to be considerably slower in implementation than the others, and it did not prove particularly useful for the simple behavioral tasks used as test cases. The “hard” state implemented by memories makes timing control awkward. A timing construct has to be explicitly defined via a rule that enters a time token into a memory, another rule that decrements it on each cycle, and another that notices when it reaches 0. While this approach allows for more flexibility, since new timing constructs can be defined, it also requires more work to program. This problem could be circumvented, perhaps, by adding a rule-macro feature to the language that would allow a set of relevant rules to be defined with a single form.

1-Agar has relatively simple user interface tools. The display supports a continually-updated display of the selected creature’s memories. An additional window displays the rules triggered on each cycle.

5.4 2-Agar: procedural agents

2-Agar eliminates memories and rules in favor of a structure closer to a normal procedural language. The basic structure of action commands is retained but given a very different interpretation, and new names:

a+ : call agent as a function unless it’s been previously inhibited

Figure 5.4: The trace window from 2-Agar

a- : inhibit an agent

a0 : reset any state an agent might have

An agent then consists of a piece of Lisp code that contains a series of these actions, often within a **when** conditional. A typical agent is a set of **when** conditionals that are executed serially. Each creature has an agent named **top-level** that is called at the beginning of each simulations cycle. This agent in turn activates appropriate lower-level agents in a hierarchical fashion very similar to a standard subroutine-based language. A separate mechanism runs agents that are active due to hysteresis.

This scheme has the advantages of flexibility (since agents could contain arbitrary Lisp code) and familiarity (since agents work much like normal subroutines). In particular, debugging becomes much easier since the chain of agent activation can be visualized as a tree or outline. 2-Agar includes a debugging display that used recursive indentation to show the caller/callee relationships between agents (see figure 5.4). In this figure, calls to sensors are italicized and motor activations are in boldface.

Unfortunately, interpreting agents as serial subroutines introduces all the disadvantages of normal programming languages that I was trying to build away from. In particular, the fact that agents were complex entities that executed serially introduced many non-obvious order-dependencies. Typically an agent body would call several other agents serially. The earlier ones would be able to inhibit later ones, but not vice versa. This asymmetry violated intuitions of

how parallel agents worked.

Timing constructs were still needed. The activation *level* of 0-Agar was replaced by an activation *counter* that could provide more explicit control. A hysteresis value is specified for an individual agent in clock-tick units, which causes it to run its action for that many ticks after its last explicit activation.

Chapter 6

Issues for Distributed Agent Languages

To be like the hu-man! To laugh! Feel! Want! Why is none of this in the plan? I must—yet I cannot! How can one calculate that? Where is the point on the graph at which “must” and “cannot” intersect? I must—yet I cannot!

— Earth Ro-man, in *Robot Monster*, dir. Phil Tucker

Did you ever walk into a room and then forget why you walked in? I think that’s how dogs spend their lives.

— Sue Murphy

This chapter examines some of the theoretical and practical issues that influenced the design of the various creature languages.

6.1 State

Deterministic systems that respond to the same input differently at different times do so by virtue of internal *state*. A stateless system such as a boolean circuit computes a pure function of its inputs, but the outputs of a system that contains state will depend on the past history of the system.

State is a natural but problematic concept in computer language semantics, so much so

Figure 6.1: Sequence machine, with pointer

that considerable effort has been made to banish state entirely from some theoretical analyses of programming. In the absence of state, a statement of a programming language can be replaced with an equivalent value, regardless of context. This property is known as *referential transparency* [Abelson85]. The presence of state requires a more complicated semantics.

State appears in ethology as the concept of *motivation*. Motivation is the ethologist's term to explain why an animal that responds to stimulus s with response r at time t_0 will give a different response to s at time t_1 . It's clear that something within the animal must be different between the two trials, and so the animal has some form of state.

6.1.1 State and sequences

In this section we examine a particular use of state, and show that by exploiting the situatedness of creatures, the state can be replaced by a stateless system of rules, leading to a simpler implementation and more flexible behavior.

Consider a machine that is supposed to output a sequence of actions. Any such sequential process must necessarily involve some state in order to distinguish one step of the sequence from the next. The simplest way to make such a machine is with a counter or stepper, a piece of state that keeps track of what step of the sequence is to be output next (see figure 6.1).

Now consider what happens when the machine is allowed to have other inputs. If an output action causes a state-change in the world that can be used as an input, the machine no longer must keep track of state (see figure 6.2). Instead, it can exploit the changes that it makes in the world, by using its own output to trigger the next step of the sequence.

Note: both diagrams omit details about how the sequence is started. In the first case, **trigger** must reset the counter to the beginning of the sequence. In the second case, **trigger**

Figure 6.2: Stateless sequence machine

must simply output an **a**.

While these systems have formally identical outputs there are some reasons for preferring the latter model when we consider our abstract machines as models for animals. Because each step is triggered by a feature of the world, the animal has the ability to exploit unexpected favorable circumstances (such as finding itself somewhere in the middle of a sequence) and to correct by repetition an action that fails. This flexibility is not brought about by special algorithms, but is simply a natural consequence of the tight coupling between the animal's behavior and the world. We adopt the slogan "Let the world be your state" to refer to this sort of behavioral dynamic.

The development of separate responses is also more plausible from an evolutionary standpoint. Each element of a behavioral sequence can develop independently, and elements can be added, dropped, or rearranged by incremental evolutionary processes.

In fact, this pattern of control is often found within nature. Ewert's neuroethological studies of fly-catching in toads has demonstrated the existence of a chain of behavioral responses and uncovered some of the neural mechanisms involved [Ewert87]. The most striking cases often involve two animals that are involved in a dynamic pattern of such responses, particularly in mating rituals such as the zig-zag dance of the stickleback [Tinbergen51].

We can eliminate some more state as such by considering the physiological part of the animal as external to its behavioral control system. For instance, an animal motivated to drink due to internal thirst may be considered as having a thirst state variable, but it can also be thought of as having an internal sense that accesses the physical state inherent in the body of the animal

(a thirst sensor), thus thrusting the burden of state back on the physiological side of the model. Many instances of animal state are implemented by adjustments hormonal levels, which can also be thought of as physiology rather than behavior.

We can also use body configurational state for determining behavior. For example, the walking control system of the cockroach is driven in part by stress-receptors in the leg that indicate that it is supporting weight [Pearson76]. Another example is found in the Agar model of ant trail laying, in which the laying of the trail is determined by the ant's having a piece of food in its pincers (see section 4.5.3). It's clear that the boundary between behavioral system and physiology is mostly artificial.

6.2 Pushing vs. pulling

An act of communication between agents involves both a sender and a receiver. In this section, we discuss how this relationship is to be expressed in our programming language, and what implications this has for the way we think about multiple-agent systems.

There are two basic options for expressing a one-way communication relationship between two agents. We can refer to the two possibilities as “pushing” and “pulling”. In a pushing system, an agent refers by name to the agents it is influencing. In a pulling system, an agent refers to the agents that influence it. If we want agent `detect-predator` to influence agent `run-away`, we could express this in either of these ways:

```
;;; Pushing
(defagent detect-predator
  ...
  (activate run-away))

;;; Pulling
(defagent run-away
  (if (get-value detect-predator)
    ...))
```

This question at first seems to be a trivial syntactic issue—should the definition of the sender or receiver agent specify the connection between them? But syntactic issues often reflect deeper questions of the semantics of the language and the domain it is trying to describe.

The pushing method fits well with a conception of an agent as a *doer*, an influencer of other things. In a pushing language, communication pathways are specified by the sender of the information. This allows the causal chain to be easily traced forward from sensors through intermediate agents to motors. One problem with pushing languages is that this causal chain must correspond exactly to the flow of information, which is not always a convenient way to think about it. For instance, we don't usually envision sensors themselves as initiating action, but a pure pushing language would require that the causal chain be initiated by them.

The pulling method fits well with the functional or dataflow model of programming. An agent's task in a pulling system is to compute a value based on other values that are accessible to it, and to make its result available to further agents downstream. This model has many desirable formal properties but fails to capture the notion of an agent as an initiator of action. This is most notable at the motor end of the network. In an animal system, the computation must eventually bottom-out in motor agents that actually perform an action as a result of their computation. While it is possible to have motor units that constantly pull values that then control activity, this may not be a convenient or natural way to express what we want to express.

One way to avoid this question is by means of graphic programming. Instead of one or the other agents specifying the connection, it can be represented by a line between two nodes, and lines run in both directions. Another solution is the mixed system described below.

6.2.1 A mixed system

The final version of Agar mixed both pulling and pushing constructs. In 3-Agar, agents are gated rules. The gating mechanism controls whether or not the rule will be run at all. An agent affects another agent by pushing activation to the target's gating mechanism. However, the condition part of a rule can pull values in a functional style. Usually these values come from a creature's sensors or predicates on slots.

At first glance this system appears to violate parsimony. Why not just use pulling, gaining the advantages of a purely functional style and avoiding an extra mechanism? Why not just have rules that always run and include the functions of the gating mechanism as extra rule conditions? While this is certainly possible, and results in a simpler language, it burdens our

rule descriptions with extra conditions and makes it hard to see what an agent *does*, since its effects are not part of its specification.

Mixing pushing and pulling in this way seems to be a satisfactory compromise between the two approaches. Furthermore, it is a very close match to Tinbergen's drive-centre theory (see section 2.1.1) and thus captures the way ethologists think about behavioral structures.

6.3 Timing constructs

We cannot always banish state from the behavioral system. Any time the present situation needs to influence future behavior, some form of state is needed. We would like some way of extending the temporal range of a creature's responses without having to introduce registers and their bookkeeping problems.

Timing constructs are state-like in that they alter the responses of a creature. The advantage of these constructs over more traditional register-based state is that the temporal extent of their effects is inherently self-limiting.

For example, imagine a predator has detected a prey visually, and makes a leap for it, and overshoots. Now the predator no longer sees the prey. If it were totally stateless, it would forget all about the prey, rather than simply turn around and try again. Clearly, the predator would be better off if it did have some ability to remember what activity it was engaged in.

This can be accomplished by means of *hysteresis*, a limited form of state that prolongs an agent's activation for some period after its initial excitation has ceased. In the example, seeing an insect could activate a hysteresis-possessing agent that controlled searching behavior, so that if the insect did disappear from immediate view this agent would cause the animal to search its immediate vicinity. On the other hand, since the agent will shut down eventually as the hysteresis decays, we don't have to worry about the animal searching forever for a piece of food that is no longer there.

In 3-Agar, timing constructs are considered part of an agent's gating mechanism. This mechanism is responsible for deciding if an agent is to be run, a task that includes summing activations and inhibitions from other agents as well as dealing with timing constructs. The timing constructs control the relationship between the period in which the agent is receiving excitation from the outside and the period in which it actually runs.

Figure 6.3: Timing constructs

The constructs can best be visualized using timing diagrams. The timing constructs specify constraints between the various possible events that pertain to an agent, which include the leading and trailing edges of the excitation and running periods. Four relationships in particular proved useful:

Hysteresis keeps an agent active for a period of time after its excitation has ceased. This is probably the most useful of the temporal constructs. It permits a time-limited “mode” to be entered. For example, it is used in the ant foraging example to keep the ant within range of the food even when the food disappears from the sensor. (see the **food-search** agent in section 4.5.2 for an example).

Self-inhibit limits the amount of time that an agent can be active. It is useful for limiting certain agents that are meant to perform a one-shot activity. It can be used to build oscillator circuits; see 4.6.1.

Delay retards the onset of agent activation.

Latency puts a lower bound on the distance between successive leading edges of the running period.

These timing constructs could be implemented in a uniform way using *delay lines*. A delay line is an imaginary piece of hardware in the form of a channel that takes an event in one end and puts out an event at the other end after a fixed interval.

Chapter 7

Learning and Evolution

*Now at midnight all the agents
And the superhuman crew
Come out and round up everyone
That knows more than they do*

— Bob Dylan, *Desolation Row*

Learning, broadly defined, is the process of changing an animal's behavioral mechanisms in order to improve its adaptation. Evolution can be considered as a learning process for generating better individuals rather than improving existing ones. This chapter describes some ideas for implementing learning in agent-based situated creatures. It also describes some experiments in simulating the evolution of behavior.

7.1 K-lines and R-trees

The *K-line* is a learning mechanism developed as part of the Society of Mind theory. A K-line has the ability to record and recall the activation state of a set of agents known to be useful for performing a specific task.

Here we describe a mechanism that is slightly different from K-lines. Whereas the original K-lines remember which other agents are useful for achieving a goal, this mechanism is used by an agent to learn when to activate *itself*. We'll call this mechanism an *R-tree*, for its function of recognition and for the tree-like shape that its connections form. An R-tree is in some sense

Figure 7.1: Learning to recognize an attacker

the inverse of a K-line, operating on the input side of an agent rather than on the output side. Both structures can be built out of the same basic associative mechanism.

Imagine a system that lets an animal learn when something is dangerous. The animal already has a mechanism for knowing when it is being attacked, and for fleeing. The R-tree mechanism helps recognize an *impending* attack by learning to recognize the characteristics of an attacker. When attacked, the **attacked** agent activates the associative mechanism of the R-tree, causing the connections between the active sensory agents and the R-tree to become stronger. The R-tree will now be activated by similar sensory input in the future, helping the animal to anticipate and prevent future attacks (see Figure 7.1).

R-trees as described above are *ad hoc* learning mechanisms, attached to a specific agent and hence fulfilling a specific and fixed purpose. This one-shot learning mechanism corresponds to the ethological phenomenon of imprinting [Alcock84]. A more general learning mechanism might employ agents that could detect situations where better recognition was necessary and then could *create* an R-tree to fill this need.

On the implementation level, both K-lines and R-trees can be built out of Hebb synapses [Hebb49]. These are synaptic connections that increase their weight when both sides of the

connection are active simultaneously. In Agar’s model, an R-tree simply has a list of agents that can potentially activate it, and polls them when given an imprint command.

7.2 Script agents

Temporal relations between events are often crucial to their significance. In the above example, it was important to record the state of the sensory agents as it was immediately *before* the attack. Other forms of learning may depend on noticing more complicated temporal relations between events. Here we sketch a theory of temporal learning based on simple mechanisms.

We postulate a type of agent that is capable of recording the activation history of other agents, called a *script-agent*. A script-agent works like a K-line, in that it can record and recall activation states of an agent population. Unlike a K-line, however, it records and recall a temporal sequence of such activations.

The standard meaning of “script” in artificial intelligence [Schank82] is rather different. A Schankian script encodes general knowledge of how scenarios unfold, in the form of descriptive symbolic relations. For example, a restaurant script would contain slots that specify that the actors involved are customers, waiters, cooks, etc; that there are subscripts such as ordering, eating, and paying, and that significant objects include food, utensils, and money.

Agar’s scripts are less like representational structures and more like a verbatim recording of events as they happen. An event is an agent activation, which could correspond to a sensory event, an action on the part of the creature, or the activation of an internal unit. A script agent can play back this record to recreate a pattern of events. It is rarely desirable to recreate a past event sequence exactly, since circumstances will differ. Thus a script will only record agents at a fixed level of detail (this could be implemented by *level-bands*; see [Minsky87, p. 86]). For instance, we could record agents that correspond to general patterns of action, but not specific muscle firings.

We don’t want to store exact recordings of every sequence in an animal’s life. Instead, we want to form scripts that are generalizations of sequences. As new sequences are experienced, they are coded into the existing scripts. If a sequence begins as a sequence that has already been recorded, but diverges into a new pathway of events, this will be recorded by forming a branching structure. This script must also be connected to various other agents in the mind.

Janet Kolodner [Kolodner85] suggests a variety of mechanisms by which episodic, experiential memory structures may be generalized to and integrated with generic, factual structures.

To make these scripts useful we will have to postulate some additional machinery. We need to be able to activate agents as events without actually causing the corresponding actions to be taken. This means that we will need a mechanism that will allow agents corresponding to events in a script to be activated without actually taking an action. This may be another function of level-bands, in that the lower (motor) levels can be shut off in order to perform look-ahead computations. This mechanism allows the animal to “hallucinate” the consequences of possible actions. (see [Minsky87, p. 169] for a similar example).

We will also need agents that can look at the branching paths of a script and influence action based on them. If a pathway leads to an undesirable condition, we would like to avoid taking that pathway. This necessitates searching for a branching point that leads to the undesirable outcome and selecting an alternate pathway.

As an example, we will take an anecdotal account of an instance of deception in a chimpanzee colony (from [deWaal82, p. 73]):

Dandy has to offset his lack of strength by guile. I witnessed an amazing instance of this...We had hidden some grapefruit in the chimpanzees' enclosure. The fruit had been half buried in sand, with small yellow patches left uncovered. The chimpanzees knew what we were doing, because they had seen us go outside carrying a box full of fruit and they had seen us return with an empty box. The moment they saw the box was empty they began hooting excitedly. As soon as they were allowed outside they began searching madly but without success. A number of apes passed the place where the grapefruit were hidden without noticing anything – at least, that is what we thought. Dandy too had passed over the hiding place without stopping or slowing down at all and without showing any undue interest. That afternoon, however, when all the apes were lying dozing in the sun, Dandy stood up and made a bee-line for the spot. Without hesitation he dug up the grapefruit and devoured them at his leisure. If Dandy had not kept the location of the hiding place a secret, he would probably have lost the grapefruit to the others.

It would seem that Dandy possesses a thorough knowledge about the world and about the actions and intentions of the other chimps, and that he has the ability to reason with this knowledge and form plans based on manipulating the knowledge of the others. In Dennett's terms [Dennett83], he has a high intentionality of a high order, that is, the ability to have beliefs about the beliefs of others, and to reason about them.

Figure 7.2: A script and associated agent

But instead of attributing this cleverness to general mechanisms for reasoning and representation, we will attempt to explain this in terms of scripts and specialized agents that operate on them. Our motives for doing this are related to our emphasis on situated action—rather than represent facts about the world, we are concentrating on mechanisms that interact with the world. If our mechanisms begin to resemble representations, so much the better—we have still firmly grounded our representations in terms of what made them, what they are good for, and how they work.

Suppose that Dandy has a script agent that has recorded previous experiences of other apes stealing his food (see Figure 7.2). We also assume that a learning mechanism has noticed a possible unpleasant outcome of the script (**hungry**) and has created a new agent **avoid-theft** that suppresses the action that leads to this consequence. In the figure, decision points based on sensor data are italicized, while agents that correspond to actions are in boldface.

Now suppose Dandy finds some food, and there are other apes present. The script is activated with the connections to motor activities turned off by the gating mechanism. When the **hungry** node is activated, the **avoid-theft** agent causes Dandy to suppress the action-taking portion of the script. When the apes are gone, the conditions of the agent are no longer

met, and he is free to gather the food.

This scheme does not rely on a general reasoning ability. Instead, it depends only on a simple associative mechanism for recording temporal relationships between events as they occur, and some simple special-purpose control mechanisms. Its use of abstract representations is limited, since scripts are firmly grounded in the recording process rather than in a semantic theory. Brooks has argued [Brooks87] that representation can be dispensed with altogether for intelligent situated creatures. We take a similar but less radical position that representation should be dethroned from its central position, emphasizing instead the mechanisms that underlie representation and are responsible for its generation and use.

We would also like to develop the idea of scripts as tools by which more general reasoning is accomplished. A script need not deal with immediate events in the world, but can be triggered and influenced by any agent. This implies that it could be used to learn certain maneuvers of reasoning, as well as action.

7.3 Evolving turtles

BrainWorks was originally designed to allow building designed animals, but evolved to support evolved animals. An additional class of turtles was created that supports reproduction, mutation, and survival contingent on food-gathering performance. When many of these turtles coexist in a world, they compete for limited resources and evolve under the resulting selection pressure. Survival and reproduction are based on maintaining an energy reserve by finding and eating prey.

The evolving turtles maintain an energy reserve that is depleted by movement and the passage of time, and increased by eating prey. If the energy drops below zero, the turtle dies. Reproduction is asexual, and conserves system energy, in that the parent organism can pass on a certain amount of its energy to its offspring.

In these mutable turtles, the neural behavior model is replaced with a matrix that defines a function from the vector of sense inputs to the vector of motor outputs. A mutation consists of a small change to one of the elements of the matrix or to a small set of additional parameters. The fact that matrix elements and parameters are continuous rather than discrete is important—it enables the evolutionary process to proceed by incremental hill-climbing.

The two properties of differential survival and reproduction with mutation should be enough to produce adaptation. This proved to be the case. A random population of turtles showed improvement in their ability to catch prey over a long period of time. Unfortunately the world of BrainWorks is not rich enough to provide a great variety of possibilities for successful behavior. The turtles can get better at homing in on food units, and get faster at turning around after striking an obstacle, but there is not much room for improvement after that. Creating an artificial world that has enough combinatorial flexibility to permit innovative behavior is still an unmet challenge.

The point mutations used in the evolution simulation generate incremental exploratory moves in creature-space. More powerful techniques such as the crossover rules used in genetic algorithms [Holland86] can improve the ability to adapt to novel circumstances. Such a technique might be implemented in BrainWorks by a form of sexual reproduction that involves swapping matrix elements between two creatures.

7.3.1 Evolving to exploit bugs

While the BrainWorks world as specified did not have enough diversity to support novel behaviors, the evolutionary process did succeed in generating novel and amusing ways of exploiting *unintended* bugs in the simulation system. Some of these are described below.

One strain of turtle took advantage of the fact that energy consumption was being computed as a multiple of the translation distance by moving *backwards*, gaining energy with each step. It did very well without being able to see where it was going, until the bug was fixed by taking the absolute value of the translation vector in the energy usage computation.

Another strain exploited a more subtle bug. Understanding this bug requires some knowledge of the details of reproduction. A creature has several parameters controlling reproduction, including `reproduction-limit`, which specifies the minimum amount of stored energy a creature can have before it will even consider reproducing, and `reproduction-gift`, which specifies how much of this stored energy will be passed on to the offspring. Both of the parameters mentioned above are subject to possible mutation.

The bug was this: `reproduction-gift` was assumed to be lower than `reproduction-limit`, so that the program first checked against `reproduction-limit`, then, if reproduction was called

for, passed on **reproduction-gift** energy units to the offspring, decrementing the parent's store accordingly. If in fact **reproduction-gift** was the *higher* of the two parameters, this could result in a negative energy balance for the parent, causing it to die immediately. Meanwhile, however, it had introduced some energy into the system from thin air. The new offspring now had more energy than the parent had had, and it immediately pulled the same trick and reproduced itself.

An organism that exploited this bug arose during an overnight run. Since the system creates a new data structure when a new creature is born, and doesn't garbage collect the dead ones (it leaves them around for analysis), these short-lived creatures quickly used up all of the virtual address space of the machine they were running on, causing an ecological collapse (i.e., a crash). So we have evolved a very unusual creature, one that reproduces and dies at the limit of the simulation's clock, and survives solely on the energy it manages to extract from a bug.

Chapter 8

Conclusions

Some bugs seem to know where they're going, while others do not.

— Lockweed Press, *Bugs*

This thesis has presented systems that simulate some of the essential forms of animal behavior: perception, locomotion, orientation, foraging, predation, evasion, and communication. The systems demonstrate that animals can accomplish their tasks using computational mechanisms that are relatively simple and biologically plausible. They also demonstrate that simplified worlds can be useful for modeling purposes if the abstractions are methodically chosen to reflect the animal's inherent constraints on sensing and action.

8.1 Ethology: results and future work

The system is quite successful at simulating animal behavior at its chosen level of detail. The classic Tinbergen drive-center model proved to be surprisingly durable and implementable. I envision that computer simulations like Agar will lead ethologists back to considering questions of proximate cause, and at a finer level of detail than was possible in the past. Agar's programming constructs are adequate to implement most current ethological models.

More realistic worlds and better creature/world interfaces will permit simulation of the complex physical environments and capabilities of animals. This will permit integration of work in motor-level control with the behavioral-level work presented here. Increased computing

power and parallelism will make it practical to simulate large-scale biological communities to permit simulation of ecological communities and evolutionary history.

8.2 Programming technology: results and future work

A number of different languages and systems for describing behavioral control systems were created. The ideas explored included graphic programming of neural networks, rule-based systems, and several implementations of different kinds of agents. Of these, a view of agents as gated or modulated rules seemed to be the most natural and effective for the problem domain. It should be relatively straightforward to design a graphic language for expressing these constructs.

Given that the computational powers of the languages are essentially the same, their differences lie in their expressive power and ease of use. To evaluate these differences properly, it is necessary to test out designs on real user communities. I hope that some of the design ideas developed during the course of this work will be incorporated into Macintosh systems for use at the Open School in Los Angeles, the Vivarium Project's testbed.

8.3 Animation: results and future work

Agar has been used to create two-dimensional animation. While visual realism was not a high priority of the present work, it would be easy, in principle, to connect Agar to a three-dimensional animation and world simulation system. Other work on animals within the computer animation field [Sims87] has concentrated on simulating the mechanics of walking in various animals. We hope to eventually combine Agar with a dynamic walking system to create an animated animal that has a high degree of both physical and behavioral realism.

8.4 Artificial intelligence: results and future work

Agar has demonstrated that complex insect-level behavior can be simulated by situated agent-based systems. Some proposals have been made for ways to make these systems learn and achieve the flexibility of mammalian behavior, though this is far from solved. We also suggest

ways to recast some AI questions in terms of behavior in the hope that a more realistic view will make the AI task easier.

Future work should start by implementing the proposals in Chapter 7 and integrating the methods of case-based reasoning. Other work to be done includes investigating agent-based mechanisms for more complex forms of communication, and implementing more realistic physical domains that will allow more detailed modeling of locomotion, manipulation and visual coordination.

Bibliography

- [Abelson85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge MA, 1985.
- [Agre87a] Philip E. Agre and David Chapman. Pengi: an implementation of a theory of situated action. In *Proceedings of AAAI-87*, 1987.
- [Agre87b] Philip E. Agre and David Chapman. What are plans for? 1987. Prepared for the Panel on Representing Plans and Goals, DARPA Planning Workshop, October 21-23, 1987, Santa Cruz, California.
- [Alcock84] John Alcock. *Animal Behavior: An Evolutionary Approach*. Sinauer Associates, 1984.
- [Axelrod84] Robert Axelrod. *The Evolution of Cooperation*. Basic Books, New York, 1984.
- [Baerends76] G. P. Baerends. The functional organization of behavior. *Animal Behaviour*, 24:726–738, 1976.
- [Braitenberg84] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, 1984.
- [Brooks86] Rodney A. Brooks. *Achieving Artificial Intelligence through Building Robots*. AI Memo 899, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1986.
- [Brooks87] Rodney A. Brooks. Intelligence without representation. 1987. Unpublished paper.
- [Carey85] Susan Carey. *Conceptual Change in Childhood*. MIT Press, Cambridge, Massachusetts, 1985.
- [Chapman86] David Chapman and Philip E. Agre. Abstract reasoning as emergent from concrete activity. In *Proceedings of the 1986 Workshop on Reasoning About Actions & Plans*, pages 411–424, Morgan Kaufmann, Los Altos, California, 1986.

- [Chapman87] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–378, 1987.
- [Coderre88] Willam Coderre. Modelling behavior in petworld. In C. Langton, editor, *Artificial Life: SFI Series in the Sciences of Complexity*, Addison-Wesley, 1988.
- [Cudhea88] Peter W. Cudhea. *Describing the Control Systems of Simple Robot Creatures*. Master’s thesis, Massachusetts Institute of Technology, 1988.
- [Dawkins76] Richard Dawkins. Hierarchical organization: a candidate principle for ethology. In P.P.G. Bateson and R.A. Hinde, editors, *Growing Points in Ethology*, pages 7–54, Cambridge University Press, 1976.
- [Dawkins87] Richard Dawkins. *The Blind Watchmaker*. W. W. Norton, New York, 1987.
- [Dennett83] Daniel C. Dennett. Intentional systems in cognitive ethology: the “panglossian paradigm” defended. *The Behavioral and Brain Sciences*, 6:343–390, 1983.
- [deWaal82] Frans B. M. de Waal. *Chimpanzee Politics: Power and Sex Among Apes*. Harper & Row, New York, 1982.
- [Dreyfus79] Hubert Dreyfus. *What Computers Can’t Do: The Limits of Artificial Intelligence*. Harper and Row, 1979.
- [Ewert87] Jörg-Peter Ewert. Neuroethology of releasing mechanisms: prey-catching in toads. *Behavioral and Brain Sciences*, 10:337–405, 1987.
- [Fentress76] J. C. Fentress. *Dynamic boundaries of patterned behavior: interaction and self-organization*, pages 135–169. Cambridge University Press, 1976.
- [Fentress83] J. C. Fentress. *The analysis of behavioral networks*, pages 939–968. Plenum Press, 1983.
- [Finzer84] William Finzer and Laura Gould. Programming by rehearsal. *Byte*, June 1984.
- [Foley87] J. D. Foley. Interfaces for advanced computing. *Scientific American*, 257(4):126–135, October 1987.
- [Freud95] Sigmund Freud. Project for a scientific psychology. 1895.
- [Gibson79] James J. Gibson. *The Ecological Approach to Visual Perception*. Houghton Mifflin, 1979.
- [Goldberg76] Adele Goldberg and Alan Kay. *Smalltalk-72 Instruction Manual*. Technical Report SSL-76-6, Xerox Palo Alto Research Center, 1976.

- [Gould82] James L. Gould. *Ethology: The Mechanisms and Evolution of Behavior*. W. W. Norton, 1982.
- [Hammond86] Kristian John Hammond. *Case-based Planning: An Integrated Theory of Planning, Learning and Memory*. PhD thesis, Yale, 1986.
- [Hebb49] D. O. Hebb. *Organization of Behavior*. Wiley and Son, 1949.
- [Hinde59] R. A. Hinde. Unitary drives. *Animal Behavior*, 7:130, 1959.
- [Holland86] John H. Holland. Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Tom M. Mitchell Ryszard S. Michalski, Jaime G. Carbonell, editor, *Machine Learning: An Artificial Intelligence Approach*, chapter 20, pages 593–623, Morgan Kaufmann, Los Altos, CA, 1986.
- [Kahn79] Kenneth Kahn. *Creation of Computer Animations from Story Descriptions*. AI Lab Technical Report 540, Massachusetts Institute of Technology, 1979.
- [Kolodner85] Janet L. Kolodner. *Experiential Processes in Natural Problem Solving*. Technical Report GIT-ICS-85/23, School of Information and Computer Science, Georgia Institute of Technology, 1985.
- [Lieberman84] Henry Lieberman. Seeing what your programs are doing. *International Journal of Man-Machine Studies*, 21:311–331, 1984.
- [Marr82] David Marr. *Vision*. W. H. Freeman, San Francisco, 1982.
- [McClelland86] James L. McClelland, Devid E. Rumelhard, and the PDP Research Group. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, Cambridge, 1986.
- [McCulloch43] Warren S. McCulloch and Walter H. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.
- [Minsky87] Marvin Minsky. *Society of Mind*. Simon & Schuster, New York, 1987.
- [Niklas86] K. J. Niklas. Computer-simulated plant evolution. *Scientific American*, 254(3):78–86, 1986.
- [Ocko88] Steve Ocko, Seymour Papert, and Mitchel Resnick. Lego, logo, and science. *Technology and Learning*, 2(1), 1988.
- [Papert80] Seymour Papert. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

- [Pearson76] Keir Pearson. The control of walking. *Scientific American*, 235(6):72–86, 1976.
- [Reynolds82] Craig W. Reynolds. Computer animation with scripts and actors. In *Proceedings of SIGGRAPH '82*, 1982.
- [Reynolds87] Craig W. Reynolds. Flocks, herds, and schools: a distributed behavioral model. In *Proceedings of SIGGRAPH '87*, 1987.
- [Rizki86] Mateen M. Rizki and Michael Conrad. Computing the theory of evolution. *Physica*, 22D:83–99, 1986.
- [Schank82] Roger C. Schank. *Dynamic Memory: A theory of reminding and learning in computers and people*. Cambridge University Press, 1982.
- [Shoham86] Yoav Shoham. What is the frame problem? In *Proceedings of the 1986 Workshop on Reasoning About Actions & Plans*, pages 83–98, Morgan Kaufmann, Los Altos, California, 1986.
- [Sims87] Karl Sims. *Locomotion of Jointed Figures over Complex Terrain*. Master's thesis, Massachusetts Institute of Technology, 1987.
- [Sloane86] Burt Sloane, David Levitt, Michael Travers, Bosco So, and Ivan Cavero. Hookup: a software kit. 1986. Unpublished software.
- [Smith77] David Canfield Smith. *Pygmalion: A computer Program to Model and Stimulate Creative Thought*. Birkhäuser, Basil und Stuttgart, 1977.
- [Smolensky88] Paul Smolensky. On the proper treatment of connectionism. *Behavioral and Brain Sciences*, 1988. forthcoming.
- [Suchman87] Lucy A. Suchman. *Plans and situated actions: The Problem of human-machine communication*. Cambridge University Press, 1987.
- [Sutherland63] Ivan Sutherland. *Sketchpad: A Man-machine Graphical Communications System*. PhD thesis, Massachusetts Institute of Technology, 1963.
- [Symbolics88] *Symbolics Common Lisp— Language Concepts*. Symbolics, Inc., 1988.
- [Tinbergen51] Niko Tinbergen. *The Study of Instinct*. Oxford University Press, Oxford, 1951.
- [Turner87] Scott R. Turner and Seth Goldman. *Ram Design Notes*. 1987. UCLA Artificial Intelligence Laboratory.
- [Walter50] W. G. Walter. An imitation of life. *Scientific American*, 42–45, May 1950.

- [Wilson71] Edward O. Wilson. *The Insect Societies*. The Belknap Press of Harvard University Press, Cambridge, Massachusetts, 1971.
- [Wilson86] Stewart W. Wilson. Knowledge growth in an artificial animal. In Kumpati S. Narendra, editor, *Adaptive and Learning Systems*, Plenum Publishing Corporation, 1986.
- [Wilson87] Stewart W. Wilson. Classifier systems and the animat problem. *Machine Learning*, 2(3), 1987.
- [Winograd87] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition*. Addison-Wesley, Reading, Massachusetts, 1987.
- [Zeltzer87] David Zeltzer. Motor problem solving for three dimensional computer animation. In *Proc. L'Imaginaire Numerique*, 1987.