PETWORLD: An Animal Behavior System Using Rules

by

William H. Coderre

Submitted to the Department of

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in Partial Fulfillment of the Requirements

for the Degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1986

Signature of author Signature redacted
Department of Electrical Engineering and Computer Science
May 23, 1986

Certified by Signature redacted
Marvin Minsky
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by Signature redacted
David Adler
Chairman, Department Committee

PETWORLD: An Animal Behavior System Using Rules

William H. Coderre

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering

ABSTRACT

The use of computers to model animal behavior has long been a favorite of both AI research and commercial educational software. Yet AI approaches typically model behavior in a very complex way, and commercial products aren't sophisticated enough to model behavior. In this thesis I study using a hierarchical approach to animal behavior simulation. I group production–style rules in a hierarchical structure of rule clumps called *experts*. By allowing rankings of action preferences to flow up from the bottom of the hierarchy, conflicting intentions can be resolved in clever ways with decision knowledge explicitly stated in the hierarchy. Although my system is not as powerful as others, I believe it can simulate realistic behavior patterns with little hassle. Due to the inevitable time constraints, none of this simulation has actually been implemented, but I believe the ideas simple enough to be quickly coded. I offer an example animal which exhibits interesting behavior, yet did not require intricate programming. Equivalent animals in other systems would either not be possible or would require much more complex programs.

Acknowledgements

The author gratefully acknowledges the following, without whom this thesis would not have been possible:

Jim Davis, who wrote the program that should've inspired me, and whose criticism made this thesis reasonable;

Tom Trobaugh, who kept pointing me back to the right track;

the **librarians** at the LCS reading room, who are cool beyond belief;

Phil Agre, who rubs off good ideas more than a cat sheds fur, and

Steve Strassmann, who'll be a good mother someday.

And of course all the neat people at the MIT Media Lab, who made it that much more interesting.

Rules of Life

Since this thesis is all about creating rules to live by, here are mine:

1. Simplify.
2. You can't win;
   You can't break even;
   You can't even quit the game.
   (The Laws of Thermodynamics)
3. Time is incompressible.
   (Fernando J. Corbató)
4. You can be right for the wrong reasons;
   You can be wrong for the right reasons.
   (Principle of Non-Corelation)
5. You cannot prevent a loser from losing;
   Do not try.
   (Why structured programming languages are a bad idea)
6. You have the right of way;
   You cannot argue with a cement mixer.
   (The Principles of Boston Driving)
7. When all is said and done, there's a lot to be said for being done.
   (First Principle of Theses)
8. "You've got to put it somewhere, why not leave it where it is?
   (That's what they say about water.)" *or*
   "Wherever you go, there you are" *or*
   Climbing the ladder does not change the color of the banana.
   (The Frame Axiom)

INTRODUCTION

The use of computers to model animal behavior has been a favorite topic of Artificial Intelligence research, where different programming methods have been used to create simulations of real animals. Neural nets, goal systems, and theorem–proving methods have all been used to imitate behavior. These systems tend to be very powerful — and very complex.

Animal behavior has also been the concern of several commercial software products (RobotWars, Robot Odyssey, ChipWits, and Vehicles, for example), ostensibly to teach programming skills to grade– and high–school students. These systems are usually very limited at mimicking behavior, and concentrate on robot control and problem–solving issues instead.

This thesis explores using a hierarchical system to simulate animal behavior. This approach has several positive aspects: programming is intuitive, adding knowledge incrementally is easy, and realistic behavior can be simulated with small programs. I chose to make my system run incrementally: each action is small, so each decision is used only for a short time. This allows animals to change strategies quickly (a crucial deficiency with sequential programming approaches). By using a high–level method such as rules, animal programs are more compact than low–level systems such as neural nets.

I claim that Petworld has a critical mass of sophistication to produce realistic, non–trivial behavior, without being so complex as to make creation of that behavior difficult. I'll show an example animal to back up that claim.

I expect my research to be a "working paper" for Vivarium, a five–year research project between Apple Computer Corporation, MIT, and the Open School in Los Angeles, California. The aim of Vivarium is to teach grade– and high–school students animal behavior as an approach to learning about thinking. I temper my discussion in that direction, and mention how Vivarium intends to use animal behavior simulation. I'll try to address such issues as complexity, intuitiveness, and user interface. It's not the purpose of this thesis to build a better mousetrap, but rather to make a simpler way to create a mouse.

CHAPTER ONE:

About Petworld

Petworld is a world of animals, rocks, and plants which live in a two-dimensional, limitless cartesian plane. The users create the behavior of new species of animals (what I'll call *pets*) by writing rules of behavior, clumped into behavior *experts* grouped into a hierarchy. Several animals are usually living in the world at once. Several species of animals are already available. In general, different species are mutually antagonistic. Plants are food sources which appear, last for some random number of meals, then vanish. Every time an animal eats, the plant loses one meal count and vanishes when the count reaches zero. When this occurs, a new plant is created somewhere else on the board, with care to be not too close or too far from the current plants.

Animals can move around, and plants and rocks cannot. Animals cannot push or throw things, but can carry one stone at a time. Plants and rocks are obstacles. An animal might build a nest out of rocks, and hide in it. Animals can move in any direction without reorienting themselves, and can see for a limited distance all around themselves. Animals do not reproduce.

About Animals

Each animal has a mood–state: a set of variables and a flag which indicate the condition of the animal. These are the variables `HUNGER`, `FEAR`, and `INJURY`, which have as values numbers between 0 and 100,

and the flag `PAYLOAD`. A hunger of 0 indicates being full, and a hunger of 100 indicates death by starvation. Eating resets the value to 0, and each action increments the count. Fear is determined by the distance of other animals. A low fear indicates that others are far away, and a high fear tells proximity. Injury is caused by attacks. Each attack increases the animal's injury count by 10, and with time the count goes back down. If an animal's injury count exceeds 100, the animal dies. If the payload flag is true, the pet is carrying a rock. If the payload flag is false, the pet is not.

Pets' actions are *atomic*. This means that each action is designed to take a very short time to execute. Pets' movements are small ("take a step"), eating and attacking happen very quickly, and calculation is instantaneous. Thus pets never get "stuck," say, in the middle of a lengthy movement when they should be avoiding an antagonistic animal.

About Species

When a student programs a pet, she gives it a species name. During a simulation, one or more animals of one or more species are placed in the world, as well as "appropriate" amounts of food and rocks, and the student observes the interactions that result.

Here are some prototypical species, which are "standard equipment" for Petworld.

Myopi

The Myopi (pronounced "my-OH-pee") are vicious warriors who are cursed with bad eyesight. They can only see half as far as average pets. However, once they have spotted a pet, they will follow it ruthlessly until they confront it, or lose sight of it. Yet Myopi, like any bully, can be rid of by standing ground against them. If the pet attacks a Myopi, the Myopi becomes scared of the pet and will run away from it.

Pacifia

Animals of the species Pacifia ("pass-IF-ee-yuh") are proud and basically peace–loving. If left alone, they will not harm any other animals. However, Pacifia are distance–sensitive. If an animal of another species approaches a Pacifia, it will become vengeful and will follow it for some distance to attack it. If it is attacked, it will become enraged and pursue until it finds revenge.

Vrlla

Perhaps the most fearsome animal in Petworld is the Vrlla ("VRIL-luh"). It can see further and run faster than pets. Due to heritage, Vrlla are afraid of rotted meat, and so will not consider attacking anything that appears dead. Their convictions are so strong that once they have decided that an animal is dead (by seeing it not move for five rounds in a row), they will lose any interest in ever attacking it, even if it begins moving again.

CHAPTER TWO:
HYRO: Pet Programming

HYRO is the language in which animals are programmed. It is currently envisioned to be a package of extensions to Common Lisp. I use Lisp because it is the *lingua franca* of the AI business, yet it's easy to imagine a pre–processor which converts a more user–friendly form of rules into Lisp (for example, see Chapter Four). Therefore, I won't concentrate on the mundane aspects of rulebase maintenance or language syntax, but instead on the ideas inside.

In HYRO, there will be most of the common ideas in programming languages, adapted for Petworld and the hierarchical structure of programs in it.

Actions

An action is an externally noticeable activity. Eating, attacking, moving, and getting and putting payload are all actions available to animals. I'll describe them in some detail.

Animals have to eat to stay alive. Eating is caused by the command `(EAT food-coordinate)`. The animal must be within two units of distance of a food source for this command to have effect.

The command `(ATTACK animal-coordinate)` attacks the specified animal if it is less than two units away. It does not affect rocks or food. Attacks are always successful, and add 10 to the injury count of the victim.

Animals move by executing the commands `MOVE-TOWARDS` and `MOVE-AWAY`. These commands take a coordinate as a single argument and move the animal in the direction which will bring them closest (or furthest away) from the mentioned coordinate. Attempting to move into an obstacle will have no effect. If the animal is within one unit of the destination, the movement places the animal at the destination, otherwise the animal only moves one unit in that direction. There will also be `MOVE-TOWARD-DIRECTION` and `MOVE-AWAY-DIRECTION`, which will take a vector as an argument. It is possible (and occasionally necessary) for an animal to hold still by moving toward the place that it's at.

Payload commands are `GET` and `PUT`. Each takes a coordinate as an argument, and gets or puts the rock closest to the desired coordinate, within two distance units of the pet.

Functions and Data Structures

Petworld has some simple mathematical functions for calculation, and boolean functions and mathematical relations for inference. They are the same as the ones in Common Lisp.

A coordinate in the world is a data structure in HYRO, a single entity. `(COORD thing)` returns the coordinate structure of the named `thing`, *e.g.* `(COORD ME)` always returns the coordinate of the pet. `COORD_X` and `COORD_Y` will return the x and y parts of the coordinate. There's a `DISTANCE` function, which returns the length of a straight line between two specified coordinates. `(RESULTING-COORDINATE coordinate movement)` (shorthand `RC`) returns the result of moving from `coordinate` with the specified `movement`. Coordinates can also be processed mathematically, to be used as vectors.

Example: This tells where an animal would be if it took a step toward the closest food:

```
(RESULTING-COORDINATE (COORD ME)
                      (MOVE-TOWARD (NEAREST (FOOD))))
```

The Ranking

Since HYRO is a hierarchical programming system, different program modules must communicate with each other. The method of their communication is *rankings*, which are used to convey information about preferences of actions. These preferences are used to make decisions about actions. The hierarchical aspect of HYRO is the subject of Chapter 3. For now I'll concentrate on the philosophy and mechanics of rankings.

A ranking is a sorted list of elements, which may be objects, coordinates, or actions. Elements may appear in a ranking at most once. Attached to each element is a weight, a real number between 0.0 and 1.0. Rankings are sorted on weights.

Certain rankings are predefined. For each of these rankings, the elements are unsorted; the user program may sort the elements for its purposes. `ROCKS`, `FOODS`, `ANIMALS` and `MONSTERS` (all animals not of the same species as the pet) are rankings that return the set of objects of a given type. Another is a subset of the eight compass–adjacent positions to the pet and the pet's current position: `NEIGHBORING-POSITIONS`. Only the positions to which the pet may move (i.e. are unoccupied) are returned. The rankings `EAT` and `ATTACK` return the ranking of available plants or animals, ones which are too far away are not included. `MOVE` returns a uniformly–weighted ranking of movement commands that would deposit the pet at the available compass-adjacent coordinates, with one more that leaves the animal where it is, to "play dead." (Note that the positioning of the pet is not restricted to a certain grid, and that indeed other of the move commands will move the pet at some angle other than the eight standard compass directions. I chose eight compass points for simplicity.)

Here are some of the functions on rankings we'll need:

`(NTH number ranking)` is a function to return a single element of the ranking.

`(LENGTH ranking)` tells how many elements are contained in the ranking.

`(FIRST ranking)` and `(LAST ranking)` return the elements with the highest and lowest weight, respectively.

`(WEIGHT element ranking)` tells a specified element's associated numeric weight.

`(FILTER keeping_test ranking)` returns a ranking which contains the pairs for which the `keeping_test` is true. `Keeping_test` is a specified function which returns a boolean value when presented with an element of the ranking.

`(MERGE ranking1 ranking2)` `(REMOVE ranking1 ranking2)` these functions create new rankings from old; `MERGE` will have each of the elements of `ranking1` and `ranking2`; `REMOVE` will have all the elements of `ranking1` which are not in `ranking2`.

`(SCORE function ranking)` associates with each element the value of function applied to that element.

`(MAP function ranking)` creates a new ranking. `Function`, given an element of the ranking, should produce an element for the new

ranking. The weights of each of the elements of the new ranking are initialized uniformly.

`(REDUCE-ELEMENTS function ranking)` `(REDUCE-WEIGHTS function ranking)` These two functions recursively apply function to elements or weights of the ranking, and return a single object.

Examples:

• Consider the definition of `NEAREST`: It shows a functional abstraction that will be used in later programs.

```
(DEFUN NEAREST (towhat thingtype)
   (LAST (SCORE (lambda (thing)
                        (DISTANCE (COORD thing)
                                  (COORD towhat)))
                (thingtype))
```

• To move a step toward the nearest food,

```
(MOVE-TOWARDS (COORD (NEAREST ME (FOOD))))
```

• Move toward the nearest food to which no monster is closer than you:

```
(MOVE-TOWARDS
  (COORD
    (NEAREST
      (FILTER (lambda (food)
                (< (DISTANCE (COORD ME) (COORD food))
                   (DISTANCE (NEAREST food (MONSTER))
                             (COORD food))))
              (FOOD)))))
```

• This function generates a ranking of possible movements scored on how well they take the animal away from the "center of mass" of the monsters:

```
(SCORE (lambda (move)
         (COORD-DOT
           (COORD-SUBTRACT (RC (COORD ME) move)
                           (COORD ME))
           (COORD-SUBTRACT
             (COORD-SCALE (/ 1 (LENGTH MONSTERS))
                          (REDUCE VECTOR-SUM MONSTERS))
                   (COORD ME))))
       MOVES)
```

Problems with rankings

Rankings are not perfect. They have a tendency to become tricky to use. This is because many of the functions that operate on rankings take functions as arguments. The traditional way to fill those arguments in Lisp is to use `lambda` statements, and those can be ugly. I argue that by avoiding lambda constructs, things won't get nearly as messy as fast. Additionally, this unpleasantness is probably an artifact of using Lisp, and might go away when or if a better way of writing HYRO exists. (I explore one possibility in Chapter Four.)

There are two ways to avoid using `lambda`s with Lisp: either provide a large library of useful functions, or provide functional abstraction as a feature of HYRO. I recommend doing both. Currently functional abstraction is simply performed with a Lisp `DEFUN`.

About Rules

Animals are programmed with rules, similar to the class of programs known as production systems. Each rule has an `IF` clause and a `THEN` clause. The `IF` clause is a single conditional statement whose truth determines if the `THEN` clause will be used in determining the animal's action. Of course, that conditional might be a binary combination of other conditions. The `THEN` clause should be a ranking recommending actions to take with preference weights attached. Default rules are allowed, as shown below.

Example:

Suppose our critter is hungry. We want it to move toward food, in the hopes of eating it. We'll have it use the naïve strategy of moving toward the nearest food.

```
IF (< (DISTANCE (COORD ME) (COORD (NEAREST FOOD))) 2)
THEN    EAT

IF *DEFAULT*
THEN (SCORE (lambda (movement)
              (DISTANCE (COORD (NEAREST FOOD))
                        (RC (COORD ME) movement)))
             MOVES)
```

Memory

I think a simple form of memory will be sufficiently powerful for HYRO. Things are remembered under headings, simple strings of text. Associated with the heading is a value, which may be an object, a movement, a coordinate, or a ranking. Any expert in the animal will be able to access these strings. Here are the commands for utilizing memory:

```
(LEARN string value)
(FORGET string)
(REPLACE string value)
(RECALL string)
```

CHAPTER THREE: STRUCTURING THE SYSTEM

One of the traditional problems with rule–driven systems is complexity. There are often hundreds or thousands of rules in the rulebase. Some sort of structuring on the rulebase is clearly needed, not only for efficiency in running the system, but also for clarity in maintaining it. Instead of writing a rule for every state an animal could possibly be in, one could easily imagine writing fewer, more general rules and providing a mechanism for resolving conflicts between them.

Another classic problem of rules is that often enough either too many or not enough rules fire. Too few rules firing can easily be solved by adding a special kind of rule known as a default, which fires only when nothing else does. Too many rules firing is another story, though. An animal can't perform two incompatible actions at the same time, yet it would be good to allow compromises between different strategies. If the animal has decided to "run away" in this round, and there were two directions of running that were almost equally valid in light of the criterion for running, it might wish to choose the direction that was better, say, for obtaining food. This sort of subtle compromise between rules is very useful for making sophisticated pets.

I examine several ways of dividing up the database: grouping, meta–rules, and hierarchies. While grouping and meta–rules can be useful, hierarchies have clear advantages.

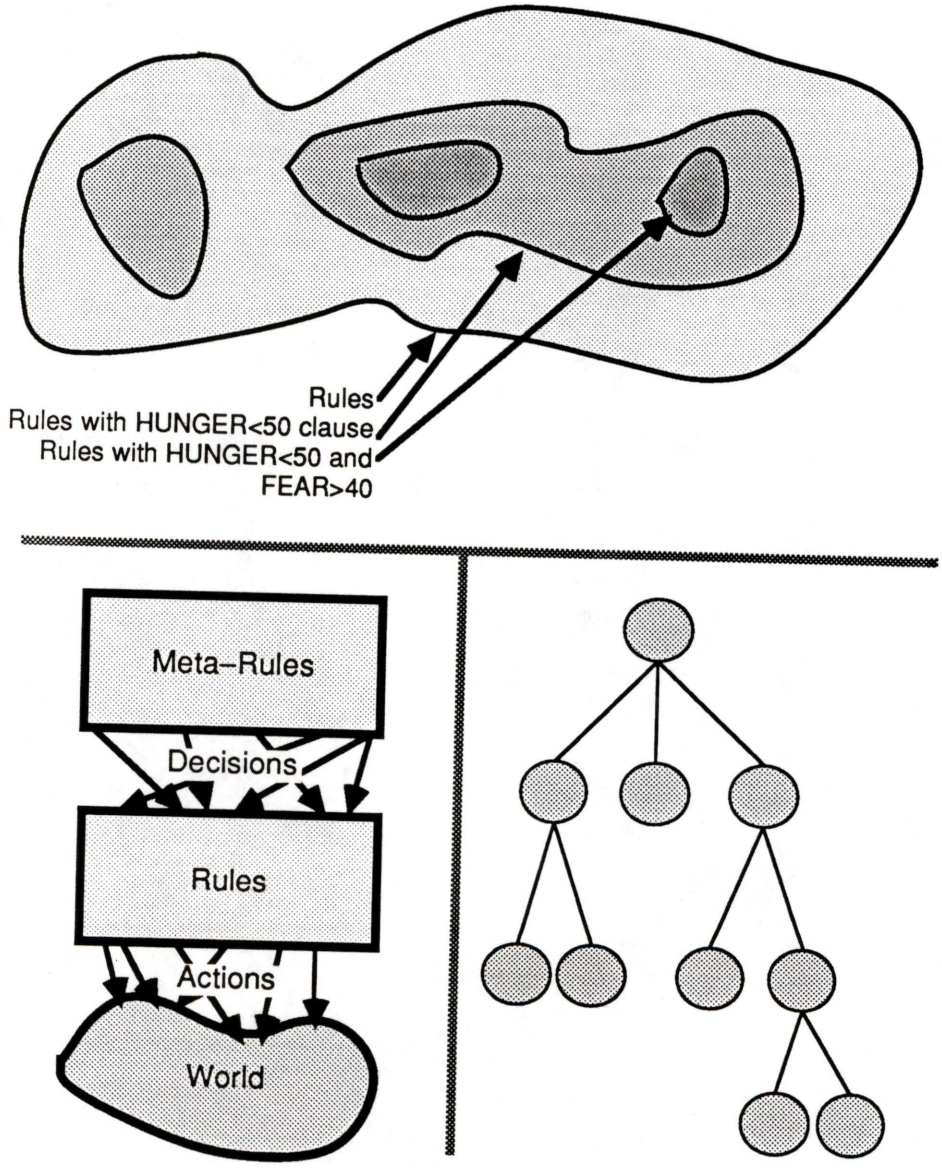

Figure 1: Grouping, Meta–Rules, and Hierarchies

• If there were no structure to our rulebase, many of the rules would have common clauses. It might seem useful to group rules that have these common clauses, to reduce the amount of typing and pattern-matching during execution, and to keep them organized. One might imagine wrapping these rules up in some form of structure. While this helps somewhat with managing complexity, every decision will still have to be *explicitly represented.* And there is no chance of reconciliation between strategies other than explicit coding of all the alternatives.

• Meta–rules can solve the firing problem. Meta–rules are rules about rules, rules which sort the set of rules that would fire. The drawback of strict meta–rules is that still the reconciliation problem remains. There is no chance for a compromise between experts without explicit coding.

• In a hierarchical system the rulebase is divided into segments called *behavior experts*. Each expert is a collection of rules which performs some small task (a low–level expert might be called `MOVE-TOWARD-NEST`, and a higher–level one might be `RUN-AWAY`). Inside each expert there should be few enough rules to prevent conflicts. Default rules, but not meta-rules, are allowed in each expert.

The organization of the hierarchy of experts controls the flow of decisions. There are several approaches to this form of hierarchical structuring. We examine them below.

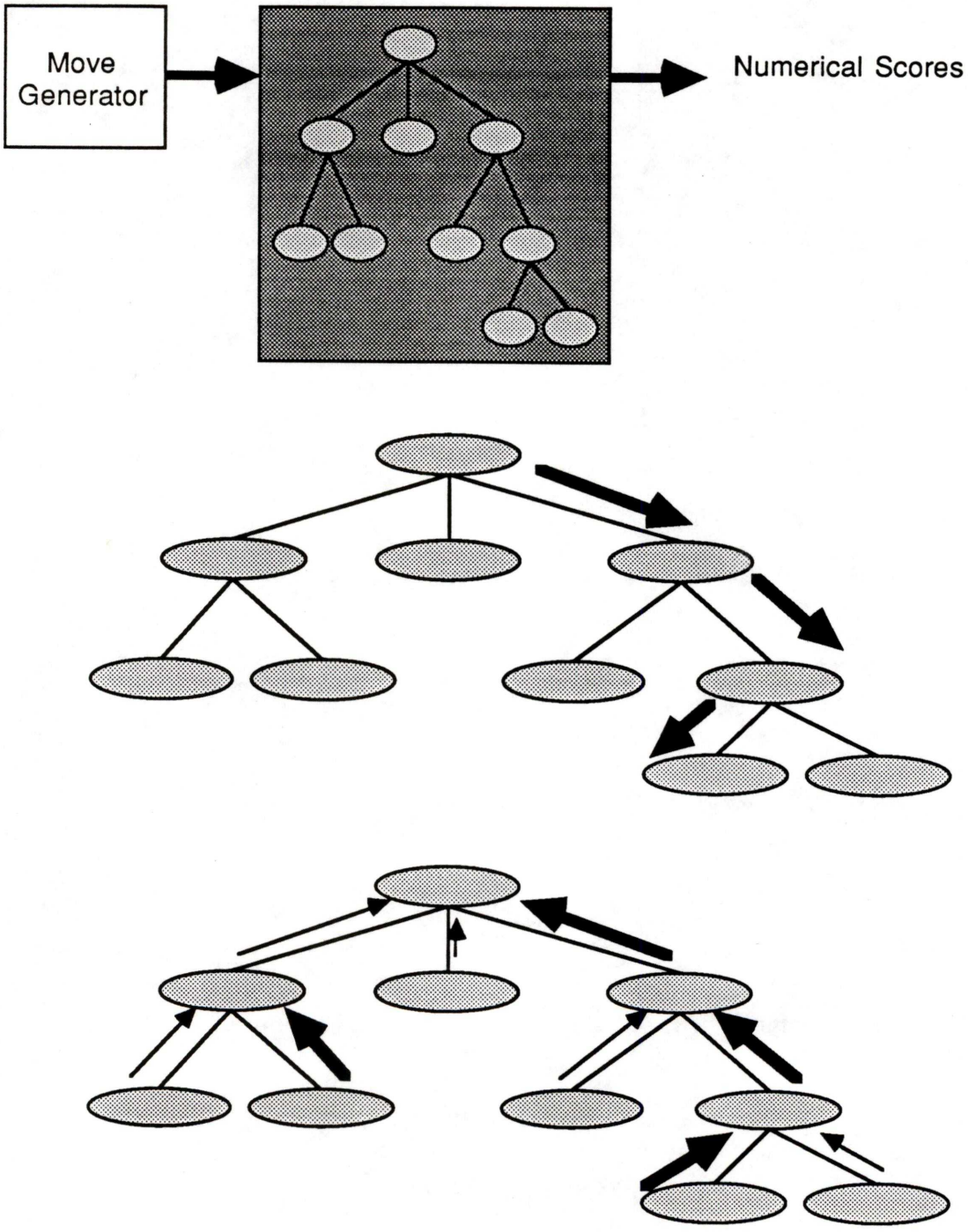

Figure 2: Decision Trees, Branching Down, and Burbling Up

• A simple approach, called a *decision tree,* is based on a science known as operations research, and is similar to how many computer games work. A next–move generator feeds the hierarchy. Each expert passes to its superior a numeric rating for each of the moves based upon world state, pet mood–state, and the opinions of its subordinate experts. The action that gets the best top–level rating executes.

The drawbacks of this approach are that it evaluates a lot to find out a little, and it requires that compromises be finagled as weightings of the scoring functions, rather than being written out as rules. Some of the decision knowledge is hidden in the rating scheme.

• Another approach is similar to the meta–rule and grouping approaches extended to their logical extreme: control branches down the hierarchy by running rules in each expert along the path until an expert at the bottom of the tree is chosen. One of its rules then fires and action is finally taken.

This approach, although theoretically as powerful as any other, in practice still has the reconciliation problem. If I wish to effect a compromise between two experts, I will have to construct a third to be the "compromise between *a* and *b*" expert, and will have to add more rules to the superior expert to decide when to choose it.

• The method we chose has low–level experts "burble up" rankings from which higher levels choose actions. Each expert will be concerned

with compromising between actions presented to it, either by chosing one action, or by creating a new action. Knowledge about making decisions is stored explicitly in the experts, and does not rely on the exact nature of the experts' rankings. Perhaps an example will illuminate the matter.

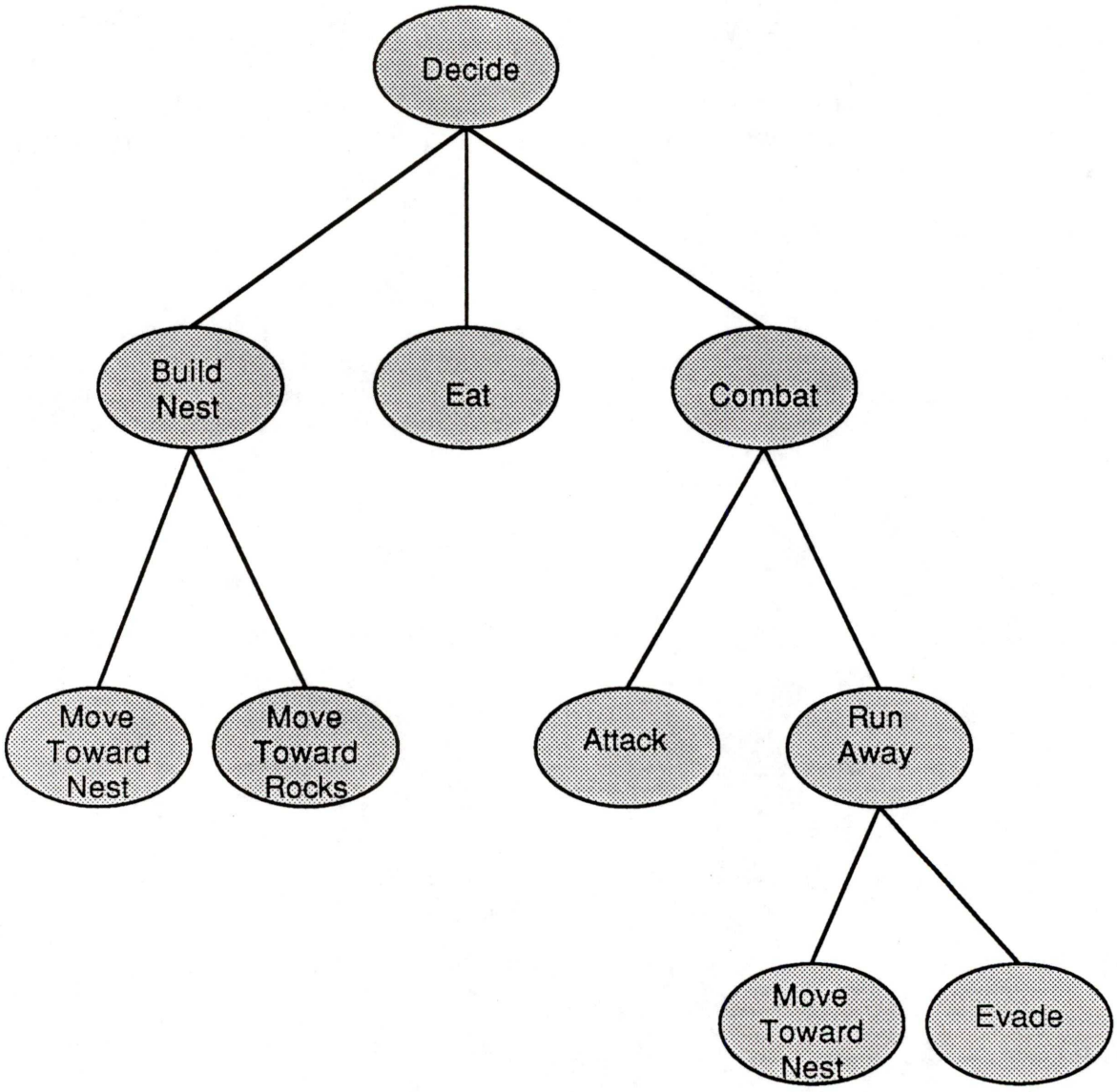

Figure 3: An Example Animal

An example animal

Figure 3 shows the hierarchical structure of a fairly interesting animal. Its major concerns are represented by different experts: `EAT`, `COMBAT`, and `BUILD-NEST`. The expert `DECIDE` will chose the appropriate type of action based on their recommendations. I'll describe the functions of the experts, and their contained reasoning, in pseudo–English for clarity.

EAT

`EAT` will recommend an appropriate action: either a movement ranking which indicates preferred directions of motion, or a simple `EAT` command when food is sufficiently close. It therefore has two internal rules:

1) If you are close enough to food, recommend eating.

2) Otherwise, recommend moving, ranking the movements on how close they will bring you to food.

Note that `EAT` doesn't care about where potential foes are, or how hungry the animal is. Those concerns will be weighed by `DECIDE`.

BUILD–NEST

`BUILD-NEST` will also be fairly simple: it will require a few subordinates, though. `MOVE-TOWARD-ROCKS` will rank movements on how well they move toward rocks. `MOVE-TOWARD-NEST` will rank movements on how well they move toward the animal's nest. Built–in reasoning involves a few rules:

1) If you are in your nest and you have a rock, put the rock in the nest.

2) If you don't have a rock, recommend what MOVE-TOWARD-ROCKS did.

3) If you do have a rock, recommend what MOVE-TOWARD-NEST did.

COMBAT

COMBAT will be in charge of interactions with other animals. Since Petworld is mostly antagonistic, and since different threats require different strategies, COMBAT will have several subordinates: RUN-AWAY and ATTACK. Each of these will have several subordinates. MOVE-TOWARD-NEST will recommend how to get toward the nest to hide, and EVADE will handle escaping the clutches of a foe. Both will report to RUN-AWAY. ATTACK will recommend either attacking or moving toward an apropriately attackable adversary. COMBAT will have to decide what the nature of the threat is and how to deal with it, independently recommending holding still when necessary. COMBAT might also decide to be more defensive when heavily injured. The heart of the matter will be to decide what the principal threat(s) is(are), and then to pick the appropriate strategy.

1) If no monster can see you, recommend what RUN-AWAY did.

2) If the closest monster is a Vrlla, then recommend holding still.

3) If the closest monster is a Myopi, and it can see you, then recommend attacking it.

4) If the closest monster is a Pacifia, then recommend running away.

DECIDE

DECIDE will resolve the conflicting input. I will give a fairly simple decision algorithm.

1) If FEAR is higher than HUNGER, and greater than 30, then do what COMBAT recommended, compromising somewhat with EAT.

2) If HUNGER is greater than FEAR, and greater than 50, then do what EAT recommended, compromising somewhat with COMBAT.

3) If HUNGER and FEAR are both greater than 70, then do what the higher of the two recommended, but wherever possible compromise with the other.

4) Otherwise, do what BUILD-NEST recommended.

Limitations of hierarchies

A rigid hierarchy is not the most powerful model available. It cannot plan or learn. These two terms are rather complex, ill-defined, and hotly debated these days, so I will state what I mean them to be.

By "plan," I refer to the kind of explicit goal creation and pursuit used in GPS, EURISKO, and similar programs. Pets are improvisers which let the world tell them what to do. This means that sophisticated tasks, such as the classic monkey, banana, and ladder problem, are probably not solvable by a hierarchy.

By "learn," I refer to any kind of learning beyond the simple remembering of facts. A pet cannot change the rules in its database, for example, and it cannot learn a "concept." A pet cannot be "creative."

Nevertheless, a student can create, with only a small amount of code, a rich behavior pattern, similar in complexity and flavor to that of many natural animals. That code is unencumbered by the necessary generality baggage for planning or learning, and so remains simple to use and understand.

From the view of a learning environment, I feel it is a necessity to have animals that do just what they are told. If pets were perfect replacements for rabbits, the student would gain no more insight from watching pets than rabbits. An optimal simulation would "optimize out" the student out of the loop by learning the "right" behavior no matter what. So in order to have the student learn, I require that pets do not.

CHAPTER FOUR:
The Vivarium Project

The Vivarium Project is a five–year cooperative effort between Apple Computer Corporation, MIT, and The Open School at Los Angeles, California. It is the intention of the project to develop new tools — both hardware and software — to adjoin a curriculum of animal behavior study. The target audience is first through sixth grade students, abnormally bright for their age.

Commercial Software

There are several commercial software packages available that bear mention. None of the packages known to this author focuses on behavior. Still, here are brief descriptions some of the better packages.

Robot Odyssey

In Robot Odyssey users program robots to solve problems using simulated logic gates and sensors. Complexity is avoided by simplifying the simulation and by allowing abstraction in the form of "chip burning." Using an inherently parallel programming paradigm and exploiting the naïve users' intuitions about circuitry, the game's interface is quick for new users to learn. One of the problems of the game is that no single robot can be made sophisticated enough to solve all the problems, so users must help out their robots from time to time.

RobotWars

In this game, players program tanks which have realistic motion and sensory apparatus in a programming language similar to BASIC. They write tank programs which run without user intervention, and stage competitions between them in the battle pit. The object of the game is to create a tank which will kill its opponent tanks. The game is a lot of fun, and players tend to get very attached to their creations, constantly extending and refining them.

Since execution of the program is sequential, the tanks can get stuck performing a sequence of actions when they should be changing strategies. This problem is enhanced by the fact that there are no "interrupts" in the language, to, say, jump to a subroutine when hit with enemy fire. Additionally, geometry of the playing surface and program size are significant issues of programming tanks. As a result, the most interesting thing one learns when playing the game is the art of the finely–tuned hack, writing a program which is subtle yet simple.

ChipWits

In ChipWits the user builds a robot's program to enable it to survive independently. Survival includes finding food and zapping enemy creatures. It is a one-player game, and has several scenarios. Its graphics and animation are good, but its programming capabilities are severely limited. It, too, has sequential programs, and it also has very limited memory and programming power.

Vehicles

Valentino Braitenburg recently published a book called **Vehicles**, in which he describes how behavior and intelligence could arise from sufficiently complex groupings of simple, deterministic parts. One of his students has recreated some of his scenarios as software. This software is radically different from the other packages mentioned, since the user does not create robots or write programs.

In one simulation, the user can wire sensors to motors on the back of a vehicle's body, much as in numerous other "Turtles and Lights" simulations. Another simulation dealing with growth and decay tries to show a balance between food and population, and a third which illustrates learning by negative feedback is also included.

User Interface

With any system, the user interface is among the most crucial of parts. Vivarium has the extra–tough challenge of interfacing to young children. Allison Druin has been studying an animal design workstation which is a big stuffed animal in which the child sits. The student's direct manipulation of the critter's body parts and interaction on a touch sensitive graphics display create and animate an imaginary animal.

In Petworld, I would like to see a multi-screen environment. In one screen (or window on a screen, for that matter), the users see a graphical representation of the world, showing the actions of the animals with some

form of animation. This screen will be the focus of interest to the user during normal operation. Petworld can easily provide enough information to an animation system to display not only the actions themselves but also the motivations: we could attach animation cues to different experts (for example, a "hungry" cue to the food-gathering expert, a "frightened" cue to the running-away expert, and a "brave" cue to the expert in charge of attacking). These cues, along with the mood-state, could change the portrayal of the animal, both in terms of visible expression (facial, in the case of humans), and motion rendition (a stuttering gate, a happy stride, and so on).

In another screen, a diagram of the hierarchy of the brain of the pet would appear, much as it has in my figures. While the program was running, the pet's decisions could be shown. Starting at the bottom of the hierarchy, different rankings would be color–coded. As rankings found their way up the hierarchy, inheritance lines and nodes would change color to indicate their preference. Also on this screen would be the mood–state of the pet, represented graphically by "gas gauge" displays, and memory, probably displayed as text.

More screens would be available to display the internal operations of selected experts. The rules which make up an expert would be shown, and similarly color-coded as they make choices between inputs. Inputs to an expert (the subordinate experts' decisions) would appear, properly colored, in small windows on the screen.

The last screen in the Petworld simulation would be the Control Panel. It would behave like its counterpart on a video tape machine. Buttons marked FORWARD, REVERSE, STOP, and SLOW MOTION would be available for the user to control the flow of action. REVERSE would actually run the simulation backward, by use of a recorded history. Also available would be "markers" to remember interesting points in time.

At any point in the simulation, the user should be able to stop command flow and change rules and situations by editing the apropriate screens. Thus she will be able to immediately test theories, and try alternate strategies by backing up, marking, modifying, and replaying with new rules. The screens will allow users to quickly localize difficulties when the animal makes the "wrong choice," and will allow the user to detect when an expert fails to make an expected recommendation.

These ideas are not particularly new: several good commercial programming language debuggers have similar features. Yet this strategy allows the debugging heuristic to be fully exploited in a highly visual, highly intuitive format.

Program Creation

Rules in HYRO could be created by using template forms. Users would string together statements (bits of forms) to make a program shell, then fill in the blank parts of the forms to complete the program.These statements would run sequentially. As the user was filling in blank parts of the forms,

a help window would tell him what could be put in the given field, both with a list of probable specific choices, and with a list of general data types that would fit. A sample program written in the HYRO-FORMS prototype is shown in Figure 4.

I've chosen to make the FORMS language linear. This avoids the nested levels of complexity of Lisp. Also, although the forms have a syntax that reads like English, compilation of programs is simple, since the text is merely "syntactic sugar" for the statements. One could even imagine a conciseness switch for frequently used statements.

```
1. Remove the elements of FOODS for which
   MONSTER-CLOSER-THAN-ME is TRUE.

2. Sort the elements of RESULT-OF-STATEMENT-1
   on DISTANCE-FROM applied to ME.

3. Sort the elements of MOVES on
   WEIGHTED-DISTANCE-FROM applied to
   RESULT-OF-STATEMENT-2.
```

Figure 4: An Example HYRO-FORMS program.

Conclusions

A hierarchy is a useful approach to writing animal behavior programs. It has the benefits of simplicity, organization, power, and flexibility. Drawbacks are the lack of planning and learning, two powerful metaphors for sophisticated behavior.

Yet, I feel that the added complexity and sophistication of planning and learning are unnecessary overkill for the purposes of Vivarium. Many natural animals simply do not plan, and do not learn except in the most trivial ways. Petworld can simulate rich behavior with simple code. Pets ought to be able to simulate a fairly complex animal, with only a moderately complex program.

REFERENCES

Agre, Phil, *Routines*, MIT AI Laboratory Memo 828, May 1985.

Alkon, Daniel L., *Learning in a Marine Snail*, **Scientific American**, June 1983, pp 70 – 84.

Amari, Tom, and Druin, Allison, *The Role of Graphics in Expert Systems*, MIT Visible Language Workshop Memo {available through author of paper}, May 1986.

Batali, John, *Computation Introspection*, MIT AI Lab Memo number 701, February 1983.

Braitenberg, Valentino, **Vehicles: Experiments in Synthetic Psychology**, MIT Press, 1984.

Camhi, Jeffrey M., *The Escape System of the Cockroach*, **Scientfic American**, December 1982, pp 158 – 172.

Davis, James R., *Pesce*, computer program modelling fish behavior, September, 1983.

Davis, Randall, *Meta–Rules: Reasoning about Control*, MIT AI Lab Memo number 576, March 1980.

Groelich, Horst, **Vehicles**, software package for Apple Macintosh, MIT Press, 1986.

Hofstadter, Douglas R., *The Copycat Project: An Experiment in Nondeterminism and Creative Analogies*, MIT AI Lab Memo number 755, January 1984.

Jacobs, Walter, *How a Bug's Mind Works* {paper in unidentified book; author has listed affiliation with American University, Washington DC}.

Kay, Alan C., *Trial Vivarium Curriculum, Trial User Interface, Trial Vivarium Graphics and Animation, Trial Vivarium Moist Models*, 1985 {four papers on aspects of the Vivarium project, available by contacting author}.

Kay, Alan C., *Computer Software*, chapter of **Computer Software**, Scientific American, 1984.

Kehler, Thomas P., and Clemenson, Gregory D., *KEE, The Knowledge Engineering Environment for Industry*, Intelligenetics, Inc., 1983 {paper available from Intelligenetics, 124 University Ave, Palo Alto, CA}.

Lenat, Douglas B., *Beings: Knowledge as interacting experts*, **Procedings**

of the Fourth IJCAI, pp 126 - 133, 1975.

Lenat, Douglas B., and Harris, Gregory, *Designing a Rule System that Searches for Scientific Discoveries*, CMU Department of Computer Science.

Lenat, Douglas B., and Brown, John Seeley, *Why AM and EURISKO Appear to Work*, **Artificial Intelligence**, 1984, pp 269 – 294.

Lorenz, Konrad, **King Solomon's Ring**, Thomas Y. Crowell Company, 1952.

MacLaren, Lee S., *A production system architecture based on biological examples*, PhD thesis, U. Washtington Seattle, 1978 {available as University Microfilms order number 79–17604}.

Minsky, Marvin, **The Society of Mind**, Simon and Schuster, 1986 or 1987 {this author greatly thanks Professor Minsky for a pre–publication copy of his forthcoming book}.

Robot Odyssey I, The Learning Company {computer game widely available}.

Stefik, Mark, et al., *Knowledge Programming in Loops: Report on an Experimental Course*, **AI Magazine**, Fall 1983.

Various Authors, **The Brain**, Scientific American, 1979.