Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents

by

Joanna Joy Bryson

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2001

Author . . . .
Department of Electrical Engineering and Computer Science
June 29, 2001

Certified by . . . .
Lynn Andrea Stein
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents

by
Joanna Joy Bryson

Submitted to the Department of Electrical Engineering and Computer Science
on June 29, 2001, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

All intelligence relies on search — for example, the search for an intelligent agent's next action. Search is only likely to succeed in resource-bounded agents if they have already been biased towards finding the right answer. In artificial agents, the primary source of bias is engineering.

This dissertation describes an approach, Behavior-Oriented Design (BOD) for engineering complex agents. A complex agent is one that must arbitrate between potentially conflicting goals or behaviors. Behavior-oriented design builds on work in behavior-based and hybrid architectures for agents, and the object oriented approach to software engineering.

The primary contributions of this dissertation are:

1. The BOD architecture: a modular architecture with each module providing specialized representations to facilitate learning. This includes one pre-specified module and representation for action selection or behavior arbitration. The specialized representation underlying BOD action selection is Parallel-rooted, Ordered, Slip-stack Hierarchical (POSH) reactive plans.

2. The BOD development process: an iterative process that alternately scales the agent's capabilities then optimizes the agent for simplicity, exploiting tradeoffs between the component representations. This ongoing process for controlling complexity not only provides bias for the behaving agent, but also facilitates its maintenance and extendibility.

The secondary contributions of this dissertation include two implementations of POSH action selection, a procedure for identifying useful idioms in agent architectures and using them to distribute knowledge across agent paradigms, several examples of applying BOD idioms to established architectures, an analysis and comparison of the attributes and design trends of a large number of agent architectures, a comparison of biological (particularly mammalian) intelligence to artificial agent architectures, a novel model of primate transitive inference, and many other examples of BOD agents and BOD development.

Thesis Supervisor: Lynn Andrea Stein
Title: Associate Professor of Computer Science

Acknowledgments

There are two kinds of acknowledgement pages — short, formal, safe ones, and long ones. In long ones, you leave off half the people you had always intended to acknowledge and you stick some people in randomly because they talked to you the day before. But I feel obliged to do my best.

Yesterday my new officemate, John Stedl, talked to me. Thanks also to Brian Anthony, Jesse Hong, Robert Irie, Mark Core, Lucas Ruecker, Will Neveitt, Deniz Yuret, Yoky Matsuoka, Greg Huang, and everyone else who ever shared my office. Thanks also to Pearl Tsai Renaker, Elizabeth Wilmer and Eva Patalas, my housemates when I was first at MIT. These are the people who keep you sane. Special thanks to Krzysztof Gajos for being my officemate even though he really worked next door.

Much of what this dissertation is about is how to make things work. Writing about this involves finding out how other things really work. In terms of AI and agent architectures, I've had enormous numbers of conversations over the years I couldn't possibly remember or properly acknowledge. But here are a few people that have been particularly insightful and supportive: Kris Thórisson, Paolo Petta, Push Singh, Mark Humphrys, Ian Horswill, Michael Mateas and Leslie Pack Kaelbling. Rob Ringrose, Chris Wright, Jered Floyd, Olin Shivers, Lynn Stein, Henry Hexmoor, Brendan McGonigle, John Hallam, Greg Sullivan, Johanna Moore, Mark Steedman, Lisa Inserra, Dan Dennett, Rene Schaad, Lola Cañamero, Simon Perkins, Pattie Maes, Luis Correia, Luc Steels, John Hallam, Michael Thomsen, Gillian Hayes, Nuno Chalmique Chagas, Andrew Fitzgibbon, Chris Malcolm, Chris Mellish, Tony Prescott, Libby Levison, Dave Glasspool and Yannis Demiris. Thanks especially to Rob Ringrose and Torbjrn Semb Dahl. Thanks also to Simon Rushton, Terran Lane, Sandy Pentland, Jimmy Lin, and everyone else who helped me at the defense stage.

Understanding how things work at MIT was almost as important as understanding AI. Besides my advisor, committee and officemates I got help from Lynne Parker, Maja Matarić, Anita Flynn, Oded Maron, Latanya Sweeney, David Baggett, Greg Klanderman, Greg Galperin, Holly Yanco, Tina Kapur, Carlo Maley, Raquel Romano, Lily Lee, Mike Wessler, Brian Scassellati, Jerry Pratt, Bill Smart, Jeremy Brown, Marilyn Pierce, Jill Fekete, Annika Pfluger, Lisa Kozsdiy, Dan Paluska, Henry Minsky, Marvin Minsky, Michael Coen, Phil Greenspun, Gerry Sussman, Hal Abelson, Patrick Winston, Nick Matsakis, Pete Dilworth, Robyn Kozierok, Nick Papadakis, Matt Williamson, Lisa Saksida, Erich Prem, Ashley Walker, Blay Whitby, Phil Kime, Geraint Wiggins, Alan Smaill, and Heba Lakany. Thanks also to Howie Schrobe, Trevor Darrell, Peter Szolovitz and Mildred Dressellhaus for being on exam boards for me. Thanks to Bruce Walton, Dan Hagerty, Ron Wiken, Jack Costanza, Leigh Heyman, Petr Swedock, Jonathan Proulx, Chris Johnston and Toby Blake for making stuff work. Thanks to everyone on the Cold Booters, especially Ron Weiss, David Evans, Kah-Kay Sung, Daniel Coore, Sajit Rao, Mike Oltmans and Charlie Kemp. For that matter, thanks to Louis-Philippe Morency and all the Halting Problem people. Special thanks to Gill Pratt and Lynn Andrea Stein who have always been supportive.

Kris Thórisson once commented to me about how often he'd seen me making constructive criticisms of his architecture, Ymir. For the record, I only bother to think about and write about things that are very interesting, and I'm embarrassed that people sometimes fail to realize that the compliments preceeding my criticisms are sincere. I would like to acknowledge Kris in particular: I've worked closer with him than anyone else in AI, particularly during our eight months together at LEGO.

Similarly, Rod Brooks and the subsumption architecture changed my life. It was his paper *Intelligence Without Reason* that introduced me to modular, distributed intelligence, convinced me to do a PhD, and brought me to MIT. It was on his and Lynn Stein's Cog project that I started thinking about specialized learning. I reread that paper a month ago, and I'm amazed the extent to which this dissertation is just an expansion of it, despite the conspicuous differences. I've never met him, but Toby Tyrrell did an amazing piece of work for his PhD. I have a deep respect for John Laird, his group, and the Soar project in general, particularly for the way they document their failures as well as their successes. They also have also been very responsive via email. If I spend more time critisizing these people's work, it is only because I spend a great deal of time thinking about it.

Thanks to my family, the blood relatives: Tom Bryson, Donna Hathaway, Barb, Kevin and Rebecca Bird, and Thelma Swezey; and the new members: Martha Bryson, Brian and Scott Mullany, Patricia and David Lowe, Carolyn Hollins, and Kate Davies. Hilary Till, Amy Sterly, Gill Kendon, Terran Doehrer and Jutta Distler, and Chris and Angel Johnston have been practically family. I would like to remember especially my aunt, Dorothy Bryson, who programmed satellites for GE in the 1950's (and got paid a third of what 'the boys' did for her work), and my grandfather, Roy Swezey, who always wanted me to be the best, and pretty much thought I was regardless.

The team who created the VR project in Chapter 12 were Dent-de-Lion du Midi, Kristinn R. Thórisson, Christian Greuel, Svend Tang-Petersen, Victor Bonilla, Dan Mapes, Claude Aebersold, Inge E. Henriksen, Preben Kæseler, and Julian E. Gómez. Their work as artists, visionaries and engineers is gratefully acknowledged. The AI and dialogue work was directed by Kris Thórisson, and included, in addition to those mentioned above, Mikkel Arentoft, Súsanna Thorvaldsdóttir and Christian George. Claude Aebersold was responsible for the animations mentioned in Section 12.4.3, and Victor Bonilla for the guard character. Lieselotte van Leeuwen assisted with child psychology; Celia Pearce assisted with concept development; Jacob Buck and Michael Nielsen contributed to technical implementation. The castle project was developed using Silicon Graphics equipment and expertise. The team on the dialog project in the same chapter were Johanna Moore, Claus Zinn, Peter Wiemar-Hastings and Mark Core — Mark also helped on my write up. Thanks to Massimo Poesio too. Thanks to Alf-Christian Achilles and his Computer Science bibliography web site, `http://liinwww.ira.uka.de/bibliography/index.html`.

Thanks to my readers: Olin Shivers, Gill Pratt and Bruce Blumberg. They gave me their time, comments and support. But the biggest contributer to this dissertation (well, besides me) was Lynn Andrea Stein. Without her ideas, support (emotional and financial), enthusiasim, patience and determination, this would never have happened.

Will Lowe belongs in every category on this page and a few more. Thank you.

Contents

List of Figures

Chapter 1

Introduction

Intelligence relies on search — particularly, the search an agent makes when it chooses its next act. Search is only likely to succeed in resource-bounded agents if they have already been biased towards finding the right answer. In artificial agents, the primary source of bias is engineering. Thus engineering is the key to artificial intelligence.

This dissertation describes an approach, Behavior-Oriented Design (BOD) for engineering complex agents. A complex agent is one that must arbitrate between potentially conflicting goals or behaviors. Common examples include autonomous robots and virtual reality characters, but the problems are shared by many AI systems, such as intelligent tutors, monitors or environments. Behavior-oriented design builds on work in behavior-based and hybrid architectures for agents, and the object oriented approach to software engineering.

This chapter describes the contributions of this dissertation, first at a high level, then in more detail. There is a preliminary introduction to Behavior-Oriented Design, an argument about the importance of design in artificial intelligence (AI), and an explanation of the forms of evidence provided in this dissertation. Finally, there is a chapter-level description of the rest of the dissertation, including road maps for readers with various interests, and an description of its core motivations.

1.1 Contributions

The primary contributions of this dissertation are:

1. the BOD architecture, and

2. the BOD development process.

The BOD architecture consists of adaptive modules with specialized representations to facilitate learning. This includes one pre-specified module and representation for *action selection*, the arbitration between the expressed behavior of the other modules. The specialized representation underlying BOD action selection is Parallel-rooted, Ordered, Slip-stack Hierarchical (POSH) reactive plans.

The BOD development process is iterative: it alternately scales the agent's capabilities then optimizes the agent for simplicity, exploiting tradeoffs between the component representations. This ongoing process for controlling complexity not only provides bias for the behaving agent, but also facilitates its maintenance and extendibility. BOD provides rules for:

- The initial decomposition of the agent into modules and plans. This decomposition is based on anticipated adaptive state requirements.
- The iterative improvement of the agents design. These rules take the form of heuristics for simplifying the agent, and recognizing when the current decomposition is faulty. Due to BOD's modularity, new decompositions (even switching intelligence between modules and plans) can be achieved with minimal disruption.

Secondary contributions of this dissertation include: an analysis and comparison of the attributes and design trends of a large number of agent architectures, two implementations of POSH action selection, a procedure for identifying useful idioms in agent architectures and using them distribute knowledge across paradigms, several examples of applying BOD idioms to established architectures, a comparison of biological (particularly mammalian) intelligence to artificial agent architectures, a novel model of primate transitive inference, and many other examples of BOD agents and BOD development.

In the analysis of design trends (Chapter 3), I conclude that intelligence for complex agents requires the following three features:

- *modularity*, a decomposition of intelligence to simplify the agent's design,
- *structured control*, a way to focus attention and arbitrate between modules to bring coherence to the agent's behavior, and
- *environment monitoring*, a low-computation means to change the focus of the agent's attention.

In the biological comparison (Chapter 11) I show that mammalian intelligence also shares these features. These features provide the basis of the BOD architecture.

1.2 Behavior-Oriented Design (BOD)

Behavior-Oriented Design is a methodology for constructing complex agents. It is designed to be applicable under any number of languages and most popular agent architectures. As can be gathered from its name, BOD is a derivative of Behavior-Based Artificial Intelligence (BBAI) [Brooks, 1991a, Maes, 1991a, Matarić, 1997], informed by Object-Oriented Design (OOD) [e.g. Coad et al., 1997]. Behavior-Based AI is an approach that specifies that intelligence should be decomposed along the lines of perception and action. Behaviors are described in terms of sets of actions and the sensory capabilities necessary to inform them. This sensing must inform both **when** the actions should be expressed, and **how**. In other words, there are really two forms of sensing: sensing for detecting context, and sensing for parameters and feedback of motor actions.

The central observation of behavior-oriented design is that mere sensing is seldom sufficient for either detecting context or controlling action. Rather, both of these abilities require full perception, which in turn requires memory. Memory adds bias by recording experience and creating expectation. Perception exploits experience and expectation to perform discriminations more reliably than would otherwise be possible.

This observation has two consequences in the BOD methodology. First, memory becomes an essential part of a behavior. In fact, memory requirements serve as the primary cue for *behavior decomposition* — the process of determining how to divide intelligence into a set of modules. This strategy is analogous to the central tenet of object-oriented design, the modular decomposition of a system is best determined by its adaptive state requirements.

The second consequence is that determining context is both sufficiently important and sufficiently difficult that it requires its own representation. Control context decisions, though generally driven by the environment, must often be retained after the original trigger is no longer apparent to sensing. BOD uses hierarchical, reactive, plan structures to both ensure environment monitoring and keep track of control decision context.

The analogy between BOD and OOD begins with the metaphor between the behavior and the object. The primitive elements of BOD reactive plans are encoded as methods on the behavior objects. Equally critical is BOD's emphasis on the design process itself. As in OOD, BOD emphasizes cyclic design with rapid prototyping. The process of developing an agent alternates between developing libraries of behaviors, and developing reactive plans to control the expression of those behaviors. BOD provides guidelines not only for making the initial behavior decomposition, but also for recognizing when a decomposition has turned out to be inadequate, and heuristic rules for correcting these problems. This iterative development system results in the ongoing optimization of a BOD agent for simplicity, clarity, scalability and correctness.

1.3 Design in Artificial Intelligence

Hand design or programming of systems has always been the dominant means of creating AI systems. The intrinsic difficulty of hand coding has lead to a good deal of research into alternate strategies such as machine learning and automated planning. Each of these techniques is very successful in limited domains. However, the problem of complexity has been proven intractable to machine-implemented (resource bounded) algorithms in the limit case. Chapman [1987] has proved planning to be impossible for time- or resource-bounded agents. Wolpert [1996b,a] makes similar demonstrations for machine learning. There can be "No Free Lunch" — learning requires structure and bias to succeed. Wooldridge and Dunne [2001] demonstrate that even determining whether an agent has *some chance* of bringing about a goal state is an NP-complete problem.

More importantly, there is strong evidence that in the average case, the utility of hand design still outstrips the utility of machine learning and planning. This evidence can be found in the research trends of the planning field. AI users working under a many different paradigms have turned repeatedly to designing their plans by hand (see Chapter 3).

Machine learning and automated planning techniques can be very successful in well-specified domains. The point of a design approach such as BOD is not to deprecate these achievements, but to facilitate creating the systems in which these strategies can reliably succeed.

1.3.1 AI is Software Engineering

To reiterate, the thesis of this dissertation is *not* that machine learning or constructive planning are useless. My thesis is that neither strategy in itself will ever be a complete solution for developing complex agents. AI is a form of software engineering, and as such the primary considerations of the two fields are the same. Frederick Brooks lists these as the concerns of software engineering:

> - How to design and build a set of programs into a *system*
> - How to design and build a program or a system into a robust, tested, documented, supported *product*
> - How to maintain intellectual control over *complexity* in large doses.
>
> [Brooks, 1995, p. 288 (emphasis is Brooks')]

This dissertation addresses all of these questions. Behavior-oriented design is about building and incorporating useful modules into an agent capable of coherent behavior. It specifies procedures for developing the modules, and the coordination. These specifications include recommendations for program structure and documentation that is highly maintainable.

BOD is an AI methodology that takes into account the fact that design, development and maintenance are effectively inseparable parts of the same process in modern software engineering. Consequently, the developed system must document its own design and provide for its own maintenance. BOD provides for this by making clear, rational decompositions of program code. These decompositions are not only functionally useful and simplifying, but are easily reflected in file structure.

1.3.2 Learning and Planning are Useful

Referring back to Frederick Brooks' agenda (above), learning and planning are some of the 'programs' that need to be incorporated into an intelligent system. Nearly all of the examples of this dissertation incorporate learning, because its capacity to generalize the applicability of a program helps control the overall complexity of the system (see Chapter 6). One example that is discussed but not demonstrated (the dialog system in Section 12.2) incorporates a constructive planner while at the same time reducing and simplifying its task. Whenever learning and planning can be supported by provided structure, their probability of success increases and their computational cost decreases.

1.4 Evidence in an Engineering Dissertation

Engineering and the design process are critical to artificial intelligence, but they are not easy topics for a research dissertation. Assertions about ease of use usually cannot be proven mathematically. Further, statistically valid scientific evidence demonstrating a significant ease-of-use improvement is difficult and expensive to come by: it requires a large sample of programmers tutored in a variety of methodologies, and comparisons in terms of time, effort and quality of the final product of the programming. Because this is intractable in the average PhD program, most architecture theses resort to combining plausible arguments with a demonstration of some impressive system or systems constructed under the methodology.

In the case of this dissertation, although I have employed both of these latter strategies, I have also attempted to add a version of the scientific approach. Rather than hiring a large number of programmers myself, I examine the history of agent design as made available in the literature. When practitioners from various paradigms of AI research have converged on a particular methodology, I take this as evidence of the viability of that method. This is particularly true when the paradigm began in stated opposition to a particular methodological aspect it later adopts, or when a methodology has been subjected to substantial application with significant results. In these cases the selection of the methodological aspect can reasonably be attributed to forces other than those of personal belief or some other social bias.

This analysis appears in Chapter 3. Chapter 11 also includes a similar look at the structure of naturally evolved complex agents. The combination of such uncertain evidence cannot lead to perfect certainty, but it can lead to an increased probability of correctness. In this sense, my approach is similar to that used in a scientific dissertation, particularly one employing arguments from evolution or history. These dissertations and their theses will always be more controversial than theses conclusively proven, but their approach is better than leaving important areas of research unexamined.

1.5 Dissertation Structure and Motivation

This dissertation represents three different sorts of work, which might potentially be of interest to three different sorts of readers. The work content categories are:

- the development of an AI methodology,
- the review and integration of literature on organizing intelligent control (both artificial and natural), and
- the development of a number of artificial agents.

Although these three activities are deeply interdependent, it is possible that some readers will only be interested only in particular aspects. Therefore this section includes not only a list of chapters, but also a few ‘road maps’ to sections reflecting particular interests. This section concludes with a brief description of my personal interests in this work.

1.5.1 Chapter List

I have already begun this dissertation with a brief description of my contributions and methods, and of the importance of software design issues in artificial intelligence.

Introductory Material

Chapter 2 gives a gentle introduction to BOD, both its architectural components and its design process. This is a useful introduction to 'the big picture' and gives a number of toy examples.

The next chapter provides background material in AI architectures for complex agents. As explained earlier, Chapter 3 is critical to validating both my emphasis on design and the general structure of the BOD architecture.

Behavior-Oriented Design

The next seven chapters present the three primary attributes of BOD in detail.

Chapters 5 and 4 give a detailed description of action selection in BOD. Chapter 5 describes Parallel-rooted, Ordered, Slip-stack Hierarchical (POSH) reactive plans formally. It also discusses how and when to introduce POSH planning structures into *other* architectures. BOD can be used to implement agents under any object oriented language, and under many agent architectures. Chapter 5 introduces the concept of architectural idioms, and how insights derived in one research program can be best distributed throughout the entire agent community. It also includes specific examples of adding a key feature of POSH action selection to other existing architectures.

Chapter 4 goes into more detail on the specifics of my own POSH implementations, with examples, pseudo-code and performance statistics..

Chapters 6 and 7 describe the role of learning and modularity. Unlike most 'behavior-based' architectures that also exploit reactive planning, BOD maintains the concept of behaviors as semi-autonomous programs with their own agendas and specialized representations. Chapter 6 classifies and demonstrates the different types of state used in agent architecture. Chapter 7 continues this discussion in more detail, with multiple examples from two working systems. The first system is a robot in a blocks-world simulation, and the second is a real autonomous mobile robot.

Chapter 8 describes the BOD development process proper. The BOD methodology is critical to maximizing the simplicity and correctness of a BOD agent. Chapter 8 describes the ongoing process of trading-off between the possible BOD representations to keep the agent's code and structure clear and scalable. It also gives very practical instructions for keeping a BOD project organized, including discussing maintenance, debugging, and tool use.

Chapters 9 and 10 demonstrate the BOD methodology. Chapter 9 provides an extended example on a relatively simple modeling task. The agents constructed model primate learning of reactive plans. Besides its pedagogical utility, this chapter also advances current models of primate learning, and illustrates the theoretical interface between plans and behaviors. Chapter 10 provides another, briefer example of BOD. It fills in the gaps from the previous example by using a real-time agents, demonstrating BOD in a Multi-Agent System (MAS) setting, and showing the interaction of traditional emotional/drive theory models of action selection with POSH reactive plans. The model is of social interactions in a primate colony.

The Utility of BOD

The final chapters exhibit and summarize BOD's usefulness and describe some future work.

Chapter 11 relates the hypotheses implicit and structures explicit in BOD agents to those of biological agents, particularly mammals. It also discusses possible future work in creating more adaptive or biologically-correct agent architectures.

Chapter 12 describes the utility of BOD beyond artificial life and psychology research, in the problems of industrial applications of artificial intelligence. Such applications include intelligent environments and monitoring systems. There are extended examples of possible future applications from two real industrial applications: natural language tutoring of undergraduates and virtual reality entertainment of young children.

Finally, Chapter 13 concludes with a summary.

1.5.2 Road Maps

If you are only going to read one chapter of this thesis (besides the introduction), read Chapter 2.

If you are trying to learn about (or choose between!) different agent architectures, start with Chapter 3. Then read Chapter 2 so you are familiar with the terminology of the rest of the dissertation. Next read Chapter 5, which discusses the varying levels of reactiveness in different architectures, and how to implement features of one architecture in another. You might then want to read Chapter 6 which gives my arguments about why specialized representations are important. Finally, you should probably read 12, which gives detailed perspectives on bringing my methods to two large-scale AI projects.

If you are actually interested in natural intelligence, again start with Chapter 2 (it's quick!) then skip to Chapter 11. You may also want to read my primate modeling chapters, 9 and 10. It's possible that any of these chapters will then lead you to want to skim Chapter 3, so you can see alternative ways to represent intelligence in AI.

If you are already familiar with BOD (perhaps from a paper or a talk) and just want to finally get the implementation details, you want to read Chapter 4 and the appendices. You may also want to look at Chapters 2 and 9 for examples of how agents get developed, and Chapter 5 for alternative implementations of POSH action selection. Finally, you really ought to read Chapter 8 on the methodology.

1.5.3 Motivation

The primary personal motivation for this research has been the creation of a methodology for rapidly and reliably constructing psychologically plausible agents for the purpose of creating platforms for the scientific testing of psychological models. However, I am also motivated socially by improving human productivity, and æsthetically by clean design. The complete BOD methodology and its underlying architecture supports my personal goal, as is demonstrated by the experimental work shown in Chapters 9 and 10. In addition, various of its attributes can help in other considerations. Chapter 11 offers a bridge between BOD-like architectures and neurological models of intelligence, both natural and artificial. Pursuing more general productivity and utility particularly motivates Chapters 5 and 12.

However, *any* AI project (in fact, any software project at all) benefits from good, practical methodology facilitating both design and long-term maintenance. This theme is strong throughout the dissertation, but particularly in the three nuts-and-bolts chapters, 5, 6 and 8.

Chapter 2

BOD Basics (or How to Make a Monkey Do Something Smart)

This chapter is a gentle introduction to behavior-oriented design (BOD). It is designed more or less as a tutorial. The rest of the dissertation contains more technical descriptions of each of the concepts introduced here.

2.1 Building Your Own Monkey

This is a tutorial on designing and constructing the behavior for an artificial intelligent agent[1]. *Agent* is a term borrowed from philosophy, meaning an actor — an entity with goals and intentions that brings about changes in the world. The term 'agent' could be applied to a person, an animal, a nation or a robot. It might even be applied to a program.

The tutorial example will provide opportunities to introduce the major problems of agent design and illustrate the BOD approach to solving them. This example is building a robot monkey — one that would live with us, that we might take along to a party.

Designing the intelligence for such an agent requires three things: determining **what** to do **when**, and **how** to do it.

2.1.1 How

In a BOD agent, **how** is controlled by a set of modular programs, called *behaviors*.

If we are building a monkey from scratch, then at some level we have to take care of what every individual component is doing at every instant in time. To make this problem easier, we break it up into pieces and write a different program for each piece. There might be different behaviors for sitting, jumping, playing, eating or screeching.

[1]There are other ways to make monkeys do intelligent things, but this is more interesting and doesn't involve issues of animal welfare.

2.1.2 When

When can mean "at what time", but it is more powerful if you can be more general and say "under what circumstances." **When** is the problem of *action selection*. At any time, we need to be able to say what the monkey should do *right now*. In BOD, we solve this problem by providing a structured list of circumstances and actions: a *reactive plan*. More formal definitions of these terms can be found in Chapter 5, but for now it's enough that we share a vocabulary.

2.1.3 What

What is a problem of terminology and abstraction — at what level of granularity do we have to determine **when** the monkey will act? How much detail do we have to give? Assume that right now we want the monkey to stay out of trouble while we decide what to do next. Should we tell the monkey to 'wait'? To 'sit'? To 'put your legs under your body, put your hands on your knees, look around the room and at approximately 2 minute intervals (using a normal distribution with a standard deviation of 30 seconds around the interval) randomly select one of three situation-appropriate screeches and deliver it with gesticulations'?

The problem of choosing a **what** comes down to this: which **what** you chose determines how hard a **how** is to write. But making a **how** program easier to write might make describing **when** to execute it harder, and *vice versa*. For example, if we decide that one **what** should be 'create world peace,' or even 'go buy a banana,' programming the **how** becomes complicated. On the other hand, if we make a **what** into something easy to program like 'move your little finger up an inch', we will have to do a lot of work on **when**.

There are two parts to this problem. One is *behavior decomposition*. This is the problem of deciding what should be in each behavior module. If you have ever worked on artificial perception (e.g. vision or speech recognition) you might recognize that behavior decomposition is somewhat analogous to the problem of segmentation.

The other problem is determining an *interface* between **how** and **when**. Unlike many architectures, BOD does not treat the problem of behavior decomposition and interface as the same. For example, there may be a single sitting behavior that has several different interfaces for 'sit down', 'wait quietly', 'wriggle impatiently' and 'get up'. Thus **when** plans aren't made of just of **how**s, but of **what**s. Plans are made of *actions* that behaviors know how to do.

2.1.4 BOD Methodology

The trick to designing an agent is to choose a set of **what**s that make the **how**s and **when**s as easy to build as possible.

We do this by first making an educated guess about what we think the **what**s should be. Then we develop **how** and **when** iteratively. If it turns out we were wrong about our first guess about the **what**s that's OK; we can change them or replace them.

Development is an iterative, ongoing process. We try to build something simple, and then if it doesn't work, we try to fix it. If it does work, we try to build it bigger, better or more interesting. One of the mistakes people sometimes make is to make a project more and more complicated without being careful to be sure they can maintain it. The BOD methodology reminds developers that development is an ongoing process. Its critical to continually look at how the agent can be simplified, and to make it as clear as possible how the agent works.

If a **how** becomes too complicated, we decompose it into simpler pieces (new **what**s). For example, if 'wait' turns out to be too complicated a thing to build, we might split it into 'sit and scratch', 'snooze', 'look around', and 'play banjo.' We then need to recombine the **what**s using some **when**s. For example we might want to say 'snooze if you're tired enough', 'look around every 2 minutes', 'play the banjo when no one is listening' and 'scratch if you aren't doing anything else.'

If a **when** becomes too complicated, we develop new **how**s to support and simplify the decision process. For example, we may want to build a new **how** for the monkey so she can tell whether she's tired, or whether anybody's listening.

BOD exploits the traditional software engineering tools such as hierarchy and modularity to make things as simple as possible. It heavily exploits the advances of object-oriented design and corkscrew development methodologies. It also uses new representations and understandings of intelligent processes from artificial intelligence (AI).

2.2 Behaviors: Saying How

The way we say **how** under BOD is using object-oriented programming methodologies. The particular language isn't that important, except that development in general and our corkscrew methodology in particular, goes much faster in untyped languages like lisp, perl or smalltalk than in typed ones like C++ or Java. Of course, typed languages can *run* relatively quickly, but in general, the time spent *developing* an agent is significantly more important than the speed at which it can, in the best case, execute commands. *Finding* that best case is harder than making the agent run fast.

2.2.1 The Simplest Behavior

In BOD we decompose **how** into modules called *behaviors*, which we code as objects. Behaviors are responsible for perception and action. Perception is the interpretation of sensory input into information useful for controlling effectors. Effectors are anything that affects the external world. They might include motors on a robot, nodes in a model for a virtual-reality character, the speaker or screen of a personal computer, or a teletype for an agent trying to pass the Turing test. Since behaviors are responsible for governing effectors, they must also perform any learning necessary for perception and control. Thus, like *objects* in software engineering, they consist of program code built around the variable state that informs it.

The simplest possible behavior is one that requires no perception at all, and no state. So, let's assume we've given our monkey a sound card attached to a speaker for one of its effectors. One ultimately simple behavior would be a screeching behavior that sends the sound card the instruction to go “EEEEEEEEEE...” all of the time. We’ll call this the screeching behavior. We’ll draw it like this, with the name underlined:

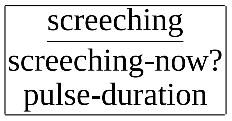
screeching

2.2.2 Behaviors with State

Unfortunately, constant screeching does not have much æsthetic appeal, nor much communicative power. So we might want to make our screeching behavior a little more sophisticated. We can give it a bit of state, and only have the action communicated to the soundboard when that bit is high.

To make our screeching behavior even more interesting, we might want it to be pulsed, like “EEee EEee EEee”. This requires some more state to keep track of where in the pulse we are. If we make our screeching sound a function of the pulse’s current duration, we only need an accumulator to keep track of how long our monkey has been screeching.

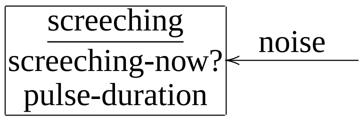
Now the screeching behavior looks like this:



We draw the state inside the behavior’s box, under its name.

2.2.3 Behaviors with Perception

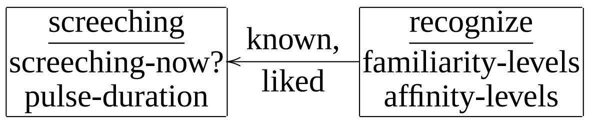
Relatively little behavior operates without regard to other events in the environment, or is controlled *open loop*, without feedback. For example, we might want our monkey to be able to modulate the volume of her screeching to be just loud enough to be heard over the other ambient noise in the room. The monkey should screech louder at a party than while she’s sitting in a quiet house. This requires the monkey to have access to sound input (a microphone of some kind) and to be able to process that information to determine her own volume. We might include this all as part of our screeching behavior.



On the other hand, some perception might be useful for screeching, but require many states or processes that are generally unrelated to screeching. In that situation, it makes more sense for the additional perception to be handled by another behavior, or set of behaviors.

For example, real monkeys start screeching when they see someone enter the room. They also make *different* screeches depending on whether that person is a friend, an enemy, or a stranger. Visual recognition is a fairly complicated task, and is useful for more things than just determining screeching, so we might want to make it a part of another behavior. Replicating state is generally a bad idea in software engineering (it makes it possible the copies will become out of synch), so it is better if the screeching behavior uses the state of the visual recognition behavior to help it select the formants for a particular screech.

Here is a drawing of two behaviors:

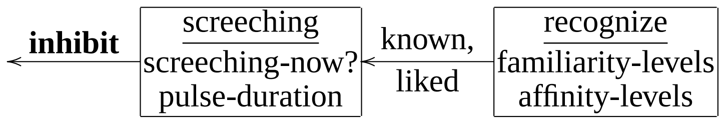


The method calls used to relate the two are put on an arrow between them. The direction of the arrow indicates the flow of information, *not* responsibility for making that information flow. In fact, normally it is the 'receiving' behavior that actively observes information in other behaviors.

2.2.4 Behaviors with Triggers or Processes

Mentioning multiple behaviors brings up the possibility of conflicts between behaviors. For example, what if our monkey is at a surprise party and sees the main guest walk into the room? The monkey should inhibit her screeching until someone gives the signal to start shouting. Similarly, if this is to be a very polite monkey, she shouldn't start screeching exactly when someone new comes up to her if she is eating a bite of cake! First she should swallow.

Under BOD, conflict resolution is handled by allowing an action selection mechanism to determine **when** things should be expressed. The interface between **when** and **how** is called **what**. A **what** is coded as a method on the object underlying a particular behavior. So for our screeching behavior, we might want to add a **what**, 'inhibit', which lets plans specify the exceptional situations where the monkey should stay quiet. Deciding to do a **what** can be viewed either as deciding to *release* or to *trigger* an action of a particular behavior.

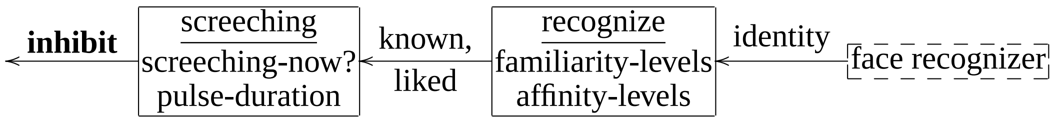


On the other hand, some actions of behaviors (such as learning or perceptual processing) may run continuously or spontaneously without interference from the **when** part of the intelligence. So long as they cannot interfere with other behaviors, there is no reason to coordinate them. For example, there's no reason (unless we build a duplicitous monkey) to control the selection of formants from the **when** system. The screeching behavior could be continuously choosing the appropriate screech to make, regardless of whether it is currently screeching or not, by having a process constantly resetting its state on the basis of the identity (or lack of known identity) of any person the monkey is observing.

2.2.5 Behaviors that Aren't Objects

Some **how**s may be easier to build using other means than coding them from scratch. For example, they may be available in external packages, or they may be easier to learn than to program. That's OK too: in that case, both **what**s and inter-behavior methods are just an interface to those other programs or packages.

Often even in these cases it is useful to have another more conventional behavior that maintains state determined by the external behavior. For example, for our monkey's face recognition, we might use a commercial package that returns the identity of the individual as a vector. We might also have a coded object behavior that learns from experience to categorize these vectors into friend, enemy, familiar neutral or unfamiliar.



2.3 Plans: Saying When

More consideration about programming **how** is given in Chapter 6, and real examples on working systems are shown in Chapter 7. But for now, we will turn to problem of deciding **when**.

In BOD, **when** is controlled using structures that are read by a special behavior for action selection. In AI, structures that control action selection are generally called *plans*. BOD uses hand-coded, flexible plan structures. Such plans are often called *reactive plans*, because with them the agent can react immediately (without thinking) to any given situation.

2.3.1 The Simplest Plan

The simplest plan is just a list of instructions, for example:

$$\langle \text{get a banana} \rightarrow \text{peel a banana} \rightarrow \text{eat a banana} \rangle \qquad (2.1)$$

Such a list is called a simple sequence, or sometimes an *action pattern*.

2.3.2 Conditionality

Of course, specifying the complete behavior for the entire lifetime of your monkey in a sequence would be tedious. (Besides, it's provably impossible.) A more common way to specify **when** is to associate a particular *context* which the agent can perceive with a **what.** Such a pairing is often called a *production rule*. The context is called the rule's *precondition* and the **what** is called its *action*.

For example, the plan 2.1 could be changed into a set of rules:

$$\begin{aligned}\text{(have hunger)} &\Rightarrow \text{get a banana}\\ \text{(have a banana)} &\Rightarrow \text{peel a banana}\\ \text{(have a peeled banana)} &\Rightarrow \text{eat a banana}\end{aligned} \tag{2.2}$$

I have put the contents of the precondition in parentheses to indicate that they are really a question. If the question is answered ‘yes’, then the rule should fire — the action should be executed.

It might look like our new plan is as good as or better than our old plan. For one thing, we’ve specified something new and critical — **when** to execute the plan itself. For another, if somebody hands our monkey a peeled banana, she will be able to execute the rule ‘eat a banana’ without executing the whole sequence in plan 2.1.

Unfortunately, it’s not that easy. What if we had another sequence we wanted our monkey to know how to do. Let’s say that we intend to have our monkey to a dinner party, and we want her to be able to pass bananas to other guests[2]. Here’s the original sequence:

$$\langle \text{get a banana from left} \rightarrow \text{pass a banana to right} \rangle \tag{2.3}$$

But if we translate that into rules:

$$\begin{aligned}\text{(left neighbor offers banana)} &\Rightarrow \text{get a banana from left}\\ \text{(have a banana)} &\Rightarrow \text{pass a banana to right}\end{aligned} \tag{2.4}$$

Now we have two rules that operate in the same context, ‘have a banana’. What should our monkey do?

We could try to help the monkey by adding another piece of context, or *precondition*, to each of the rules. For example, *all* of the rules in plan 2.2 could include the precondition ‘have hunger’, and all the rules in the plan 2.4 could have the condition ‘at a party’. But what if our monkey is at a party *and* she’s hungry? Poor monkey!

The problem for the programmer is worse than for the monkey. If we want to determine what the monkey will do, we might have to add an exception to rules we’ve already written. Assuming we think being polite is more important than eating, when we begin writing our party rules, we’ll have to go back and fix plan 2.2 to include ‘not at a party’. Or, we might have to fix the behavior that runs the monkey’s rules to know that party rules have higher priority than eating rules. But what if we want our monkey to eventually eat at the party?

Thus, although the production rule structure is powerful and useful, it doesn’t have some of the critical things we have in a sequence. A sequence maintains removes ambiguity by storing control context. Thus if our monkey keeps the steps of plan 2.3 together in a sequence, then when she takes a banana from the left, she knows to try to pass it to the right. In any other circumstance, if she has a banana, she never needs to think of passing it.

[2]Don’t try this with real monkeys.

2.3.3 Basic Reactive Plans

To summarize the previous section:

- Production rules are useful because they facilitate flexible behavior and the tying of action to context. However, they rapidly become difficult to manage because of the amount of context needed to differentiate rule firing.

- Sequences work well because they carry that context with them. A **what** embedded in a sequence carries disambiguating information about things that occurred just before or just after. The sequence itself represents an implicit *decision* made by the monkey which disambiguates the monkey's policy for some time.

We can combine many of the attributes of these two features using another structure, the *Basic Reactive Plan* or BRP. Let's try to rewrite plan 2.1/2.2 again:

$$\text{(have hunger)} \Rightarrow \left\uparrow \left\langle \begin{array}{rl} \text{(full)} \Rightarrow & \textit{goal} \\ \text{(have a peeled banana)} \Rightarrow & \text{eat a banana} \\ \text{(have a banana)} \Rightarrow & \text{peel a banana} \\ \Rightarrow & \text{get a banana} \end{array} \right\rangle \right. \tag{2.5}$$

What this notation indicates is that rules relevant to a particular activity have been clustered into a BRP. The BRP, like a sequence, limits attention to a small, fixed set of behaviors. It also encodes an ordering, but this time not a strict temporal one. Instead, it records a *prioritization*. Priority increases in the direction of the vertical arrow on the left. If the monkey already has a peeled banana, she'll eat it. If she has a whole banana, she'll peel it. Otherwise, she'll try to get a banana.

The BRP is a much more powerful structure than a simple sequence. If the monkey eats her banana, and she still isn't full, she'll get another one!

Often (as in this case) the highest priority step of a BRP is a special rule called *goal*. The goal detects when the BRP's task is finished. A BRP ends if either none of its rules can fire, or if it has achieved its goal (if it has one.)

Notice that when we make a sequence into a BRP, we reverse its order. This is because the highest priority item goes on the top, so that its precondition gets checked first. Notice also that the last rule doesn't need a precondition: instead, it's guarded by its low priority. It will only fire when the monkey's action selection attention is in the context of this BRP, and none of the other rules can fire.

There is more information about BRPs in Chapters 5 and 4.

2.4 Making a Complete Agent

2.4.1 Drive Collections

Plans that contain elements that are themselves plans are called *hierarchical*. The natural questions about a hierarchy are "where does it start?" and "where does it end?" We already know that the plan hierarchies end in behaviors, in the specification of **how**.

The start could in principle be any plan. Once that plan ends, the agent's intelligence is just over. This makes sense for certain kinds of software agents that might be called into existence just to perform a certain job. But what if we are interested in making something like a monkey? Something that lasts for some time, that decides what to do based on a set of motivations and principles?

We call this sort of agent a *complete agent*. And at the start (or *root*) of its plan hierarchy, we put a BRP specially designed never to end. This special designing just involves blocking the two ways BRPs can end. First, the BRP has no goal, so it never 'succeeds' and completes. Second, the BRP must have at least one element that can always run, so it never fails.

Here's a BRP that might govern the monkey we've been building:

$$\text{‘life’} \left\uparrow \left\langle \left\langle \begin{array}{rl} \text{(at a party)(obligations exist)} \Rightarrow & \text{be polite} \\ \text{(hungry)} \Rightarrow & \text{eat a banana} \\ \text{(friends around)} \Rightarrow & \text{make friends comfortable} \\ \Rightarrow & \text{wait} \end{array} \right\rangle \right\rangle \right. \tag{2.6}$$

Notice I added a sense for perceiving obligations. That way, our monkey can eat even when she's at a party, so long as she's not aware of any social obligations. I didn't specify a goal, and I included a low-priority behavior that can always run, so that 'life' should never end.

Drive collection is a special name for this top / root BRP. In a BOD agent, the drive collection also works as the environment monitor, something that every agent architecture needs (see Chapter 3). Drive collections have some special features to help make the agent particularly reactive; these are explained in Chapter 4.

2.4.2 The BOD Methodology: Choosing What

The previous parts of this chapter have talked about the elements of a BOD agent's *architecture*. But BOD also has a *methodology* for constructing agents. It has two parts: creating an initial specification, then iteratively building the agent.

Describing and ordering the motivations of a complete agent like I did in the last section is actually part of the specification process. Here's what you need for the entire specification:

1. A high level description of what your agent does.

2. Collections of actions (in sequences and BRPs) that perform the functions the agent needs to be able to do. *The Reactive Plans*

3. The list of **what**s (including questions / senses) that occur in reactive plans. *The Primitive Interface* (the **what**s).

4. The objects collecting state needed by the primitives, and the program code for acquiring and using that state. *The Behavior Library*.

5. A prioritized list of high-level goals the agent will need to attend to. *The Drive Collection*.

And here's what you need to do to build the agent:

1. Choose a piece of the specification to work on.
2. Code, test and debug plans and behaviors to build that piece.
3. Revise the specification.
4. Go back to 1.

Keep doing this part over and over until your agent does everything you want it to.

2.4.3 The Principle of Iterative Design

One of the most important things about BOD and iterative design in general is realizing that specifications are made to be changed. You never really understand a problem before you try to solve it.

Its tempting once you've started working on an agent to try to make your first specification work. But this tends to get messy as you tack fixes onto your program — some parts will get bigger and more complicated, and other parts will stay small and never really get used or finished. This kind of program gets harder and harder to add to, and also can get very hard to debug.

The BOD methodology emphasizes that change will keep happening. That's why you take the time in the iterative cycle to revise the specification. You want to make sure you are keeping the agent as simple as possible, and the specification as clear as possible. That's part of the job of building an agent.

2.4.4 Revising the Specification

There are some tricks to revising a specification that are specific to the BOD architecture. In fact, the BOD architecture is designed to help make this process easy. This section introduces some of the basics; again there will be more detail later (in Chapter 8).

The main design principle of BOD is *when in doubt, favor simplicity.* All other things being equal:

- It's better to use a sequence than to use a BRP.
- It's better to use a single primitive than to use a sequence.
- It's better to use control state than variable state in a behavior.

Now, if these are the rules of thumb, the question is, when do you violate them? Here are the heuristics for knowing when to violate a rule of thumb:

- Use a BRP when some elements of your sequence either often have to be repeated, or often can be skipped.

- Use a sequence instead of one primitive if you want to reuse part, but not all, of a primitive in another plan.
- Add variables to a behavior if control state is unnecessarily redundant, or has too complicated of triggers.

We've already talked about the first rule and heuristic in sections 2.3.2 and 2.3.3. The second heuristic is a basic principle of software engineering: *Never code the same thing twice — Make a generic function instead.* There's a simple reason for this. It's hard enough to get code written correctly and fully debugged once. Doing it again is asking for trouble.

The third rule is the principle of reactive intelligence, and the third heuristic helps explain why you don't want a fully reactive agent. Sometimes having memory around makes control so much easier that it's worth it.

Let's do another example. Consider the primitive 'get a banana', which we used in plan 2.5. **How** does the monkey get a banana? Lets suppose we've coded the monkey to go to the kitchen, climb on the counter, and look in the fruit bowl. If there's a bunch, she should break one off; if there's a loose one, she should take it; if there's none, she should throw a fit.

Clearly, a good deal of this could be coded either as a plan, or as a behavior. The principle of BOD is that it should be a behavior, *until or unless* you (as the programmer) could be using some of those same pieces again. So, for example, if you next decide you want your monkey to be able to get you a glass of water, you now have a motivation to write two plans:

$$\text{`get a banana'} \Rightarrow \langle \text{go to the kitchen} \rightarrow \text{take a banana} \rangle \tag{2.7}$$

$$\text{`get glass of water'} \Rightarrow \langle \text{go to the kitchen} \rightarrow \text{pour glass of water} \rangle \tag{2.8}$$

Notice that these plans are now guarded with not a question, but a plan element, a **what**. We have changed a particular **what** (get a banana) from being a simple method on a behavior to being a sequence. But we *don't* need to change our old plan (2.5). We just update part of the specification.

2.5 Behavior-Oriented Design as an Agent Architecture

The fields of autonomous robots and virtual reality have come to be dominated by 'hybrid', three-layer architectures. (The process of this dominance is documented in Chapter 3.)

Hybrid architectures cross the following:

1. *behavior-based* AI (BBAI), the decomposition of intelligence into simple, robust, reliable modules,
2. *reactive planning*, the ordering of expressed actions via carefully specified program structures, and
3. (optionally) *deliberative planning*, which may inform or create reactive plans, or, in principle, even learn new behaviors.

BBAI makes engineering easier by exploiting modularity. Reactive planning makes BBAI easier to engineer by simplifying the arbitration between behaviors. Deliberative planning is generally included to reorganize existing plan elements in the case of 'unanticipated' changes in the world. For example a planner might choose an alternative route through an office complex if a door is found shut.

The best description of three-layered hybrid systems I know is this one:

> The three-layer architecture arises from the empirical observation that effective algorithms for controlling mobile robots tend to fall into three distinct categories:
>
> 1. reactive control algorithms which map sensors directly onto actuators with little or no internal state;
> 2. algorithms for governing routine sequences of activity which rely extensively on internal state but perform no search; and
> 3. time-consuming (relative to the rate of change of the environment) search-based algorithms such as planners.
>
> [Gat, 1998, p. 209]

Gat's view of three-layer architectures is particularly close to my own view of agent intelligence, because it puts control firmly in the middle, reactive-plan layer. The deliberative 'layer' operates when prompted by requests. We differ, however, in that I do not believe most primitive actions can be defined simply by mapping sensors directly to actuators with little internal state or consideration for the past.

As I said in Chapter 1, nearly all perception is ambiguous, and requires expectations rooted in experience to discriminate. This experience may be extremely recent — for example, a phoneme in speech is much easier to recognize if you remember the phoneme that immediately preceded it, because speech production is affected by the starting position of the mouth. Useful experience may also be only fairly recent, for example remembering where you set down a banana before you answered the phone. Or it may be the result of life-long learning, such as learning to recognize a face, or learning your way around a house or a town.

The primitive actions governed by reactive plans may well be dependent on any of this information. If action is dependent on completely stateless primitive modules, then such information can only be utilized either by having some 'higher' level with state micro-manage the primitive level (which defeats its purpose) or by using some generic parameter stream to communicate between layers (which removes specialization). Neither solution is good. Rather, in BOD I recommend fully embracing modularity. Each 'primitive act' is actually an interface to a semi-autonomous behavior module, which maintains its own state and possibly performs its own 'time-consuming' processes such as memory consolidation or search in parallel to the main activity of the complete agent. BOD is still reactive, because at the time of action, the primitive can do a look-up onto its own current state with minimal computation.

Thus my view of agent control is very similar to Gat's, except that:

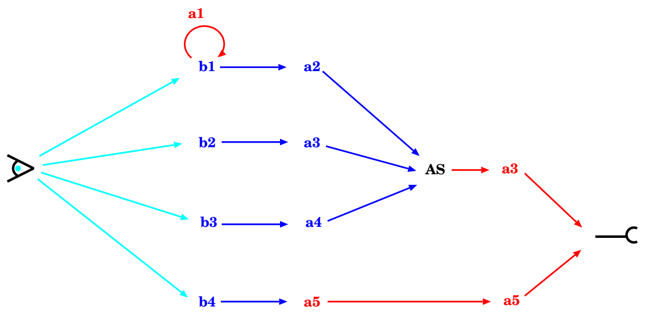


Figure 2-1: Behavior-oriented systems have multiple, semi-autonomous skill modules or *behaviors* ($b_1 \ldots$) which generate *actions* ($a_1 \ldots$) based on their own perception (derived from *sensing* indicated by the eye icon on the left). Actions which affect state outside their generating behavior, whether internal to the agent or external (indicated by the hand icon on the right), are generally subject to arbitration by an *action selection* (AS) system.

1. I increase the number, importance, specificity and potential simplicity of the modules composing his top layer. I call this the *behavior library*.

2. I replace the notion of a bottom layer with that of an interface between the action-selection module of an agent and its (other) behavior modules.

In the words of the tutorial, Gat's high level translates into **how**, his middle layer translates into **when** and his reactive layer is reduced simply to their interface, **what**. In BOD, dealing with shut doors is the domain of one particular behavior that knows about maps, not of a general-purpose reasoning system.

A simple diagram of the BOD architecture can be seen in Figure 2-1. The important points of this drawing are:

- The behaviors are not controlled by action selection. They are semi-autonomous. They may act independently to update their own state, or sometimes even to change the physical world, provided that they are not likely to interfere with other behaviors.

- Action selection itself may be considered just another specialized behavior. The reactive plans are its specialized representation.

2.6 Conclusions

In this chapter I have introduced the basic elements of behavior-oriented design (BOD). These are the architectural elements: semi-autonomous behaviors and hierarchical reactive plans; and the methodological elements: an initial task decomposition, and a procedure of incremental development.

In the next chapter, I will motivate BOD by looking at evidence from the AI literature of the utility of certain architectural features. In the chapters that follow, I will go into considerably more detail on every aspect of BOD. This is in three parts: planning and reactive plans, behaviors and specialized learning, and the design process itself. There will also (eventually) be more monkeys.

Chapter 3

Background: Critical Components of Agent Architectures

3.1 Introduction

All of the working examples in this dissertation have been implemented using control software written by me. However, my thesis claims that the principles of behavior-oriented design are general, and the contributions broadly applicable. This chapter supports this claim in two different ways. First, it documents many architectures and several different paradigms of AI research for agent development. This documentation indicates the general utility of the features of BOD, which were introduced in the previous chapters. Second, it uses these principles to make predictions and suggestions for the future directions of these other architectures and paradigms. Some of these suggestions lead to the work shown in Chapters 5 and 12, which demonstrate extending existing architectures and projects with the features of BOD.

This chapter does not attempt a full review of the related architecture literature. Instead, I concentrate on architectures or architectural traditions that are widely known or used. This increases the amount of selective pressure on the architectures. Also, the changes that are made to an architecture over time are particularly telling, so architectures that have a long and well-documented period of research are particularly interesting.

3.2 Features and Trends in Complete Agent Architectures

3.2.1 Approach

Agent architectures are design methodologies. The assortment of architectures used by the autonomous agents community reflects our collective knowledge about what methodological devices are useful when trying to build an intelligence. I consider this perspective, derived from Maes [1991a] and Wooldridge and Jennings [1995], to be significantly more useful than thinking of an architecture as a uniform skeletal structure specified by a particular program. The definition of an agent architecture as a collection of knowledge and methods provides a better understanding of how a single architecture can evolve [e.g. Laird and Rosenbloom, 1996, Myers, 1996] or two architectures can be combined (cf. Chapter 5).

The design knowledge expressed in agent architectures is of two types: knowledge derived by reasoning, and knowledge derived by experience. Knowledge derived by reasoning is often explicit in the early papers on an architecture: these ideas can be viewed as hypotheses, and the intelligences implemented under the architecture as their evidence. Knowledge derived by experience may be more subtle: though sometimes recognized and reported explicitly, it may be hidden in the skill sets of a group of developers. Worse yet, it may be buried in an unpublished record of failed projects or missed deadlines. Nevertheless, the premise of this chapter is that facts about building intelligence are likely to be found in the history and progress of agent architectures. In other words, architectures tend to include the attributes which have proven useful over time and experience.

Unfortunately, as with most selective processes, it is not always a simple matter to determine for any particular expressed attribute whether it has itself proven useful. A useless feature may be closely associated with other, very useful attributes, and consequently be propagated through the community as part of a well-known, or well-established architecture. Similarly, dominating architectures may lack particular useful elements, but still survive due to a combination of sufficient useful resources and sufficient communal support. For these reasons alone one cannot expect any particular architecture to serve as an ultimate authority on design methodology, even if one ignores the arguments of niche specificity for various architectures. But I do assume that architectural trends can be used as evidence for the utility of a particular design approach.

Identifying the design advantage behind such trends can be useful, because it allows the research community to further develop and exploit the new methodology. This is truer not only within the particular architecture or architectural paradigm in which the trend emerged, but can also benefit the autonomous control community in general. To the extent that all architectures face the same problems of supporting the design of intelligence, any development effort may benefit from emphasizing strategies that have proven useful. Many architectures have a larger number of features than their communities typically utilize. In other words, many architectures are under-specified as design methodologies. Consequently, even established design efforts may be able to exploit new knowledge of design strategy without changing their architectural software tools. They may be able to make simple reorganizations or additions to their established design processes.

In this chapter, I demonstrate this approach for evaluating and enhancing agent architectures. I survey the dominant paradigms of agent architecture technology: behavior-based design; two- and three-layer architectures; PRS and the belief, desire and intention architectures; and Soar and ACT-R. I begin by looking at some of the historic concerns about architectural approach that have shaped and differentiated these communities. I then review each paradigm and the systematic changes which have taken place within it over the last 15 years. I conclude with a discussion of these architectures in terms of the lessons derived from that review, making recommendations for the next stages of development for each paradigm.

3.2.2 Thesis

To this chapter clearer, I will begin by reiterating the results, which were introduced in Chapter 1. My analysis indicates that there are several architectural attributes necessary for producing an agent that is both reactive and capable of complex tasks. One is an explicit means for ordering action selection, in particular a mechanism exploiting hierarchical and sequential structuring. Such a system allows an agent with a large skill set to focus attention and select appropriate actions quickly. This has been a contentious issue in agent architectures, and this controversy is reviewed below. The utility of hierarchical control has been obscured by the fact it is not itself sufficient. The other necessary components include a parallel environment monitoring system for agents in dynamic environments, and modularity, which seems to benefit all architectures.

Modularity substantially simplifies the design process by substantially simplifying the individual components to be built. In this dissertation, I define modularity to be the decomposition of an agent's intelligence, or some part of its intelligence, into a number of smaller, relatively autonomous units. I do not mean to imply the fully encapsulated modules of Fodor [1983], where the state and functionality of one module are strictly unavailable to others. The most useful form of modularity seems to be decomposed along the lines of ability, with the module formed of the perception and action routines necessary for that ability, along with their required or associated state.

Fully modular architectures create new design challenges. If sequential and hierarchical control are avoided, then action selection between the interacting modules becomes difficult. However, an architecture that does allow a specialized action-selection system to focus attention appropriately may fail to notice dangers or opportunities that present themselves unexpectedly. Agents existing in dynamic environments must have architectural support for monitoring the environment for significant changes in order for the complete agent to remain responsive. This environment monitoring may be either a part of the main action-selection system, or a separate system with priority over ordinary action selection.

3.3 The Traditional Approach

I will now begin my review with a brief review of traditional AI approaches to agent organization. A traditional architecture for both psychology and artificial intelligence is shown in Figure 3-1. This architecture indicates that the problems of intelligence are to transform perception into a useful mental representation R; apply a cognitive process f to R to create R', a representation of desired actions; and transform R' into the necessary motor or neural effects. This model has lead many intelligence researchers to feel free to concentrate on only a single aspect of this theory of intelligence, the process between the two transformations, as this has been considered the key element of intelligence.

This model (in Figure 3-1) may seem sufficiently general as to be both necessarily correct and uninformative, but in fact it makes a number of assumptions known to be wrong. First, it assumes that both perception and action can be separated successfully from cognitive process. However, perception is known to be guided by expectations and context — many perceptual experiences cannot be otherwise explained [e.g. Neely, 1991, McGurk and

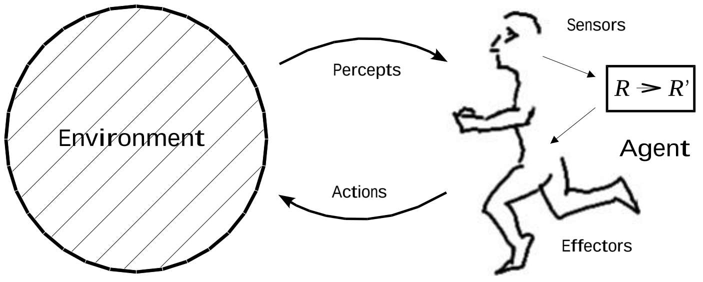


Figure 3-1: A traditional AI architecture [after Russell and Norvig, 1995].

MacDonald, 1976]. Further, brain lesion studies on limb control have shown that many actions require constant perceptual feedback for control, but do not seem to require cognitive contribution, even for their initiation [e.g. Matheson, 1997, Bizzi et al., 1995].

A second problem with this architecture as a hypothesis of intelligence is that the separation of representation from cognitive process is not necessarily coherent. Many neural theories postulate that an assembly of neurons processes information from perception, from themselves and from each other [e.g. McClelland and Rumelhart, 1988, Port and van Gelder, 1995]. This processing continues until a recognized configuration is settled. If that configuration involves reaching the critical activation to fire motor neurons, then there might be only one process running between the perception and the activity. If the levels of activation of the various neurons are taken as a representation, then the process is itself a continuous chain of re-representation. Notice that the concept of a "stopping point" in cognition is artificial — the provision of perceptual information and the processing activity itself is actually continuous for any dynamic agent. The activations of the motor system are incidental, not consummatory.

3.4 Behavior-Based Architectures

3.4.1 The Society of Mind

Though traceable in philosophy at least as far back as Hume [1748], and in psychology as far back as Freud [1900], the notion of decomposing intelligence into semi-autonomous independent agencies was first popularized in AI by Minsky [1985]. Minsky's model promotes the idea of multiple **agencies** specialized for particular tasks and containing specialized knowledge. Minsky proposes that the control of such units would be easier to evolve as a species or learn as an individual than a single monolithic system. He also argues that such a model better describes the diversity and inconsistency of human behavior.

Minsky's "agents of mind" are hierarchical and only semi-autonomous. For example, he postulates, a child might have separate agencies for directing behavior involving sleeping, eating and playing. These compete for control. When a victorious agent emerges, its subsidiary agencies in turn compete. Once playing is chosen, blocks compete with dolls and books; if blocks are chosen, building and knocking down compete during the block-playing episode. Meanwhile, the agency in charge of eating may overwhelm the agency in charge of playing, and coherent behavior may be interrupted in mid-stride as different agencies swap to take control.

The cost of theories that successfully explain the incoherence of human thought and activity is that they often fail to explain its coherence. Minsky addresses this by postulating a modular rather than a completely distributed system of thought. He explains coherent behavior as being the output of a single agency or suite of agents, and incoherence as a consequence of competing agencies. He also recognizes that there can be coherent transitions between apparently modular behaviors. To address this, he postulates another type of structure, the *k-line*. K-lines connect modules associated in time, space, or as parts of the same entity. He also posits fairly traditional elements of knowledge representation, frames and knowledge hierarchies, for maintaining databases of knowledge used by the various agents.

3.4.2 Subsumption Architecture

Brooks [1986] took modularity to a greater extreme when he established the behavior-based movement in AI. In Brooks' model, *subsumption architecture*, each module must be computationally simple and independent. These modules, now referred to as "behaviors," were originally to consist only of finite state machines. That is, there are an explicit number of states the behavior can be in, each with a characteristic, predefined output. A finite state machine also completely specifies which new states can be reached from any given state, with transitions dependent on the input to the machine.

Brooks' intent in constraining all intelligence to finite state machines was not only to simplify the engineering of the behaviors, but also to force the intelligence to be *reactive*. A fully reactive agent has several advantages. Because its behavior is linked directly to sensing, it is able to respond quickly to new circumstances or changes in the environment. This in turn allows it to be *opportunistic*. Where a conventional planner might continue to execute a plan oblivious to the fact that the plan's goal (presumably the agent's intention) had either been fulfilled or rendered impossible by other events, an opportunistic agent notices when it has an opportunity to fulfill any of its goals and exploits that opportunity.

Two traits make the robots built under subsumption architecture highly reactive. First, each individual behavior can exploit opportunities or avoid dangers as they arise. This is a consequence of each behavior having its own sensing and running continuously (in parallel) with every other behavior. Second, no behavior executes as a result of out-of-date information. This is because no information is stored — all information is a reflection of the current environment. Although useful for the reasons expressed, these traits also create problems for designing agents capable of complex behavior. To begin with, if there are two behaviors pursuing different goals, then it might be impossible for both to be opportunistic simultaneously. Consequently, any agent sophisticated enough to have potentially conflicting goals (such as “eat” and “escape danger”) must also have some form of behavior arbitration.

Subsumption architecture provides behavior arbitration through several mechanisms. First, behaviors are organized into *layers*, each of which pursues a single goal, e.g. walking. Behaviors within the same goal are assumed not to contradict each other. Higher layers are added to lower layers with the capability observe their input and suppress and even replace individual behaviors’ output if necessary. These actions occur on communications channels between the behaviors (*wires*, originally in the literal sense), not in the behaviors themselves. All such interference is designed as part of the layer; it does not affect the inner workings of a behavior, only the expressed consequences of those workings.

After experimentation, a third mechanism of behavior selection was introduced into subsumption architecture. The description of a behavior was changed from “a finite state machine” to “a finite state machine augmented by a timer.” This timer *can* be set by external behaviors, with the result being that the behavior is deactivated until the timer runs out. The timer mechanism was added to subsumption architecture because of a problem found during the development of Herbert, the can-retrieving robot [Connell, 1990]. When Herbert had found a can and began to pick it up, its arm blocked its camera, making it impossible for the robot to see the can. This would allow the robot’s “search” behavior to dominate its “pick up can” behavior, and the can could never be successfully retrieved. With the timer, the “pick up can” behavior was able to effectively *pause* all the other behaviors while it monopolized action selection for a moment.

3.4.3 Diversification in Behavior-Based AI

The use of the reactive and/or behavior-based approach is still widespread, particularly in academic robotics and character-based virtual reality. However, no single architecture is used by even ten percent of these researchers. Subsumption architecture, described above, is by far the best known of the architectures, but relatively few agents have been built that adhere to it strictly. For example, Matarić [1990], Bryson [1992] and Pebody [1995] all include adaptive extensions; Appleby and Steward [1994] make the behaviors nearly completely independent — they would now be called *agents*. Most roboticists, even within Brooks’ own laboratory, seem to have been more inspired to develop their own architecture, or to develop code without a completely specified architecture, than to attend to the details of subsumption [e.g. Horswill, 1993, Steels, 1994a, Marjanovic et al., 1996, Parker, 1998, Tu, 1999, Stone and Veloso, 1999]. Steels [1994b] goes so far as to claim that behaviors should be built so as to require neither action selection nor subsumption, but simply to run continuously in parallel with each other[1].

Of the many behavior-based architectures inspired by subsumption, the one that in turn

[1]I have been told that this strategy was abandoned for engineering reasons, although it was feasible and still considered in the lab to be a valid hypothesis for biological intelligence. It tends to require each behavior to model all of the others to a sufficient extent that they do not interfere with each other. Such modeling was too much overhead for the programmers and was abandoned in favor of inter-behavior communication.

attracted the most attention has been Maes' spreading activation network [Maes, 1991a]. Maes' architecture consists of a number of nodes, including action nodes, perception nodes, and goal nodes. The nodes are connected to one another by a two-way system of links. One link specifies the extent to which the second node requires the first node to have executed, the other specifies the extent to which the first node enables the second node to fire. These conduits are used to allow activation to spread both bottom-up, starting from the perception nodes, and top-down, starting from the goal nodes. When a single node gets sufficient activation (over a threshold) that node is executed.

Maes' greatest explicit hypothetical difference from subsumption architecture is her belief that agents must have multiple, manipulable goals [see Maes, 1990b]. Maes' claim in that paper that subsumption architecture only allows the encoding of a single goal per agent is mistaken; however the strictly stacked goal structure of subsumption is sufficiently rigid that her arguments are still valid. A more implicit hypothesis is the need for a way to specify sequential behaviors, which her weighting of connections allows. On the other hand, Maes is very explicitly opposed to the notion of hierarchical behavior control [Maes, 1991b]. Maes states that using hierarchical methods for behavior arbitration creates a bottleneck that necessarily makes such a system incapable of being sufficiently reactive to control agents in a dynamic environment.

This hypothesis was disputed by Tyrrell [1993], who showed several flaws in Maes approach, most notably that it is insufficiently directed, or in other words, does not adequately focus attention. There appears to be no means to set the weights between behaviors in such a way that nodes composing a particular "plan of action" or behavior sequence are very likely to chain in order. Unrelated behaviors may alternate firing, creating a situation known as "dithering". There is actually a bias against a consummatory or goal behavior being performed rather than one of its preceding nodes, even if it has been enabled, because the goal, being in a terminating position, is typically connected to fewer sources of activation.

Tyrrell's competing hypothesis is that hierarchy can be exploited in action selection, providing that all behaviors are allowed to be fully active in parallel, and that the final decision is made by combining their computation. Tyrrell refers to this strategy as a *free-flow hierarchy* and attributes it to Rosenblatt and Payton [1989]. Tyrrell [1993] gives evidence for his hypothesis by comparing Maes' architecture directly against several hierarchical ones, of both free-flow and and traditional hierarchies, in a purpose-built artificial life environment. In Tyrrell's test world, a small animal needs to balance a large number of often conflicting goals of very different types. For example, it must eat, maintain body temperature, sleep in its home at night, avoid two different types of predators, and mate as frequently as possible. Simulations cover up to 10 days of life and involve thousands of decision cycles per day. Using extensive experimentation, Tyrrell demonstrates substantial advantage for all of the hierarchical architectures he modeled over Maes' approach.

Tyrrell also shows statistically significant superiority of the free-flow hierarchy over its nearest strictly-hierarchical competitor, which was in fact the most simple one, a drive-based model of control. He claims that a free-flow hierarchy must be an optimal action selection mechanism, because it is able to take into account the needs of all behaviors. These sorts of cooperative rules have been further refined. For example, Humphrys [1997] suggests choosing a course that minimizes the maximum unhappiness or disapproval of the elements tends to lead to the optimal solutions. Such thoroughly distributed approaches have been challenged by my work. Bryson [2000a] suggests that simplicity in finding an optimal design, whether by a programmer or by a learning process such as evolution, outweighs the advantage of cooperative negotiation. My action-selection system uses a hierarchical controller where only a small subset of nodes, corresponding in number to the elements in the top layer in the hierarchy, actively vie for control of the agent. Further, these nodes do not compete on the basis of relative activation levels, but are activated by threshold and strictly prioritized. Thus on any particular cycle, the highest priority node that has threshold activation takes control. Within the winner's branch of the hierarchy, further competitions then take place. This is very similar to a traditional hierarchy, excepting parallel roots and some other details of execution, yet Bryson [2000a] shows a statistically significant improvement over the Tyrrell [1993] results using the same system for evaluation.

Blumberg [1996] presents another architecture which takes considerable inspiration from both Maes and Tyrrell, but also extends the control trend further towards conventional hierarchy. Blumberg's system, like Tyrrell's, organizes behaviors into a hierarchy while allowing them to be activated in parallel. However, in Blumberg's system the highest activated module wins and locks any critical resources it requires, such as legs if the module regulates walking. Nodes that are also active but do not require locked resources are allowed to express themselves. Thus a dog can both walk and wag its tail at the same time for two different reasons. The hierarchy is also exploited to focus attention in the voting system. Not every behavior participates in the vote, a fact that was initially minimalized [Blumberg, 1996], but more recently has become a stated feature of the system [Kline and Blumberg, 1999]. Blumberg's architecture is being used by his own and other research groups (including Brooks' [Breazeal and Scassellati, 1999b]) as well as a major commercial animation corporation, so its future development should be of significant interest.

Summary

All behavior-based systems are modular; the modular design strategy to a large part defines the paradigm. Most behavior-based systems rely on their modularity as their source of reactiveness — any particular behavior may express itself opportunistically or when needed. This has, however, lead to difficulties in action selection that seem to have limited the complexity of the tasks addressed by these systems. Action selection mechanisms vary widely between individual architectures, indicating that the field has not settled on a stable solution. However, several architectures are now incorporating hierarchical and or sequential elements.

3.5 Multi-Layered Architectures

The achievements of behavior-based and reactive AI researchers have been very influential outside of their own communities. In fact, there is an almost universal acceptance that at least some amount of intelligence is best modeled in these terms, though relatively few would agree that all cognition can be described this way. Many researchers have attempted to establish a hybrid strategy, where a behavior-based system is designed to work with a traditional AI planner, which deduces the next action by searching a knowledge base for an act that will bring it closer to a goal. Traditionally, planners have micro-managed, scripting every individual motion. By making their elements semi-autonomous behaviors which will react or adapt to limited uncertainty, the planners themselves can be simplified. The following is a recent account of a project from the late 1980s:

> The behavior-based plan execution was implemented bottom up to have as much useful capability as possible, where a useful capability is one which looked like it would simplify the design of the planner. Similarly, the planner was designed top down towards this interface, clarifying the nature of useful capabilities at which the behavior-based system should aim. This design method greatly reduced the complexity of the planner, increasing the complexity of the agent much less than this reduction, and thus reduced the overall system complexity. It also produced a robust system, capable of executing novel plans reliably despite... uncertainty.
>
> [Malcolm, 1997, Section 3.1]

Malcolm's system can be seen as a two-layer system: a behavior-based foundation controlled by a planning system. More popular of late have been three-layer systems, as introduced in Section 2.5 above. Two- and three-layer systems are similar, except that there is a middle layer that consists of precoded plan fragments, sometimes referred to as "implicit knowledge", in contrast to the "explicit" reasoning by the top-level planner. Another distinction is that the middle layer is often considered reactive, in that it does not create plans, but selects them based on the situation; while the top layer is a traditional constructive planner. In most systems, the top-layer planner manipulates or generates this intermediate representation level rather than acting directly on the behavior primitives.

One currently successful layered robot architecture is 3T [Bonasso et al., 1997], which features Reactive Action Packages (RAPs) [Firby, 1987], for its middle layer. RAPs is a system for creating reactive, flexible, situation-driven plans, and itself uses a lower layer of behavior primitives. 3T integrates this system with a constructive planner. 3T has been used on numerous robots, from academic mobile robots, to robotic arms used for manipulating hazardous substances, previously controlled by teleoperation, to maintenance robots for NASA's planned space station. Leon et al. [1997] uses 3T in simulation to run an entire space station, including farming and environmental maintenance. Hexmoor et al. [1997] and Kortenkamp et al. [1998] provide fairly recent reviews of many two- and three-layer architectures.

3T may seem a more likely tool for modeling of human-like intelligence than the behavior-based models discussed earlier, in that it has something approximating logical competence. However, planning has been mathematically proven an unrealistic model of intelligence because it relies on search [Chapman, 1987]. Search is combinatorially explosive: more behaviors or a more complex task leads to an exponentially more difficult search. Though there is no doubt that animals do search in certain contexts (e.g. seeking food, or for a human, choosing a gift), the search space must be tightly confined for the strategy to be successful. A better model of this sort of process is ATLANTIS [Gat, 1991], which is controlled by its middle layer, and only operates its top, planning layer on demand. This model is in fact quite similar to the Norman and Shallice [1986] model of human action selection, where conscious control is essentially interrupt driven, triggered by particularly difficult or dangerous situations. Although the alternative model, with the top level being the main controller, is more typical [Bonasso et al., 1997, Albus, 1997, Hexmoor et al., 1997, Malcolm, 1997], Gat's model would also seem a more natural extension of the behavior-based approach. It is also notable that Bonasso et al. [1997] report a number of 3T projects completed using only the lower two layers.

Another incompatibility between at least early behavior-based work and the layered system approach is the behavior-based systems' emphasis on emergence. For a hybrid system, emergent behavior is useless [Malcolm, 1997]. This is because an emergent behavior definitionally has no name or "handle" within the system; consequently the planning layer cannot use it. In humans at least, acquired skills can be recognized and deliberately redeployed [Karmiloff-Smith, 1992]. Hexmoor [1995] attempts to model both the development of a skill (an element of the middle layer) from actions performed deliberately (planned by the top layer) and the acquisition of deliberate control of skills. His hypothesis of requiring both these forms of learning are probably valid, but his actual representations and mechanisms are still relatively unproven. Another group researching the issue of learning behaviors and assigning their levels is that of Stone and Veloso [1999]. Veloso's group has had a series of highly successful entrants into various leagues of robot soccer; their architecture is thus also under strenuous selective pressure. It also seems to be converging to modularity in the areas which are most specialized, such as communication and learning, while having a directed, acyclic graph (DAG) for general action selection over preset plans.

Summary

Two- and three-layer architectures succeed at complex tasks in real environments. They generally have simplified behavior modules as their first (lowest) layer, and reactive plans in their second layer. The plan layers are carefully organized in order to maintain reactivity, although some architectures rely on the bottom-level behaviors for this function, and others do not operate in dynamic environments. Modularity has generally been limited to the lower level, though in some architectures the top-level planner can also be seen as a specialized module. Current research indicates there are still open questions concerning the optimal kind of planning for the top layer, and how to manipulate and shift information between representations, particularly learned skills.

3.6 PRS — Beliefs, Desires and Intentions

Although robotics has been dominated by three-layer architectures of late, the field of autonomous agents is dominated, if by any single architecture, by the Procedural Reasoning System, or PRS [Georgeff and Lansky, 1987, d'Inverno et al., 1997]. PRS also began as a robot architecture, but has proven sufficiently reliable to be used extensively for tasks such as defense simulations. It was originally developed at roughly the same time as subsumption architecture, as a part of a follow-up program to the longest running robot experiment ever, Shakey [Nilsson, 1984]. PRS is designed to fix problems with traditional planning architectures exposed by the Shakey project. Such problems include:

- Constructing a complete plan before beginning action. This is a necessary part of the search process underlying constructive planning — a planner cannot determine whether a plan is viable before it is complete. Many plans are in fact formed backwards: first selecting the last action needed to reach the goal, then the second last and so on. However, besides the issues of opportunism already discussed, many details of a real problem cannot be known until the plan is executed. For example, when crossing a room full of people, the locations of the people are not determined until the time of the actual crossing.

- Taking too long to create a plan, thereby ignoring the demands of the moment. The standard example is trying to cross a road — a robot will not have time to replan if it suddenly spots a car; it needs to reactively move out of the way.

- Being unable to create plans that contain elements other than primitive acts — to take advantage of skills or learned procedures.

- Being unable to manipulate plans and goals. Plans may need to be abandoned, or multiple goals pursued simultaneously.

Obviously, this list is very similar to the problems the behavior-based programmers attempted to solve. There are, however, two main differences in approach. First, PRS, like the layered architectures, maintains as a priority the ability to construct plans of action. The architecture allows for incorporating specialized planners or problem solvers. The second difference is that PRS development is couched very much in psychological terms, the opposite of Brooks' deprecation of conscious impact on intelligent processes. PRS is referred to as a BDI architecture, because it is built around the concepts of beliefs, desires and intentions.

Many researchers appreciate the belief, desires and intentions approach in concept, without embracing PRS itself. For example, Sloman and Logan [1998] consider the notions of belief, desire, intention and emotion as central to an agent, but propose expressing them in a three-layer architecture. Sloman's top layer is reflective, the middle deliberative, and the bottom layer reactive. This is similar to Malcolm [1997] or the first and third layers of 3T [Bonasso et al., 1997], but with an additional layer dedicated to manipulating the goals of Malcolm or Bonasso's top layers, and considering its own current effectiveness. This particular role assignment for the layers of a three-layer architecture is also proposed in Figure 3-2, below.

The PRS architecture consists of four main components connected by an interpreter (sometimes called the "reasoner") that drives the processes of sensing, acting, and rationality. The first component is a database of *beliefs*. This is knowledge of the outside world from sensors, of the agent's own internal states, and possibly knowledge introduced by outside operators. It also includes memories built from previous knowledge. The second component is a set of *desires*, or goals. These take the form of behaviors the system might execute, rather than descriptions of external world state as are often found in traditional planners. The third PRS component is a set of *plans*, also known as knowledge areas. Each plan is not necessarily completely specified, but is more likely to be a list of subgoals useful towards achieving a particular end, somewhat like BOD's POSH action selection. These may include means by which to manipulate the database (beliefs) to construct a next action or some new knowledge. The final main component is a stack of *intentions*. Intentions are simply the set of plans currently operating. A stack indicates that only one plan is actually driving the command system at a time, but multiple plans may be on the stack. Typically, ordering of the stack is only changed if one plan is interrupted, but new information may trigger a reorganization.

Like multi-layer architectures, PRS works from the hypothesis that a system needs both the ability to plan in some situations, such as navigation, and the ability to execute skilled acts for situations where search is not reasonable, such as avoiding trucks. In some sense, each plan is like a behavior in behavior-based AI. Behavior-based AI is essentially a retreat to allowing programmers to solve in advance the hard and important problems an agent is going to face. A procedure to solve an individual problem is usually relatively easy to design. Thus some modularity can be found in the design of the knowledge areas that make up the plan library. On the other hand, PRS does not see specialized state and representations dedicated to particular processes as worth the tradeoff from having access to general information. It has moved the procedural element of traditional planners closer to a behavior-based ideal, but not the state. It only allows for specialized or modularized data by tagging. The interpreter, goal list and intention stack are the action-selection device of PRS.

PRS and its variants exist both as a planning engine and as a set of development tools. They are used by industry and the US government as well as for research. PRS has gone through a number of revisions; in fact the original project seems to be dying. One large change in the basic structure of the original PRS was the adoption of the ACT formalism for its plan libraries, which can also be used by a conventional constructive planner [Wilkins et al., 1995]. This move can be seen as a part of a general trend in current PRS research to attempt to make the system easier to use — the idea of a planner is to allow plan libraries to be generated automatically. There is also a "PRS-lite" [Myers, 1996] which uses easily combinable "fuzzy behaviors". A number of labs have worked on formalizing PRS plans in order to make its planning provably correct [e.g d'Inverno et al., 1997]. However, these efforts had difficulty with the reactive element of the architecture, the meta-reasoning. The original development lab for PRS, SRI, is now focusing effort on a much more modularized AI architecture, built under a multi-agent paradigm [Wilkins and Myers, 1998]. Some PRS systems that are still in active use are derived from UM-PRS [e.g Huber, 1999]. One modification these systems have made is providing for the prioritization of the reactive plans in order to simplify meta-reasoning.

The pre-history of PRS, the Shakey project, also has relevant evolutionary trends [Nilsson, 1984]. Although Shakey had a traditional planner (called STRIPS), over the term of the project the concept of *triangle tables* was developed. A triangle table decomposes a plan into its steps and assumptions, then creates a contingency table allowing the plan to be restarted from any point. Perception is then used to determine which element of the plan should be executed next. This allows action selection to be reactive within the confines of the plan, rather than relying on memory of what steps should have already been executed.

This approach leads naturally into teleo-reactive plans [Nilsson, 1994], another recently developed form of storage for skilled behaviors developed by planners. Benson [1996] describes using this as the basis of a system that learns to fly airplanes in flight simulators, and the architecture is being used at a number of research laboratories.

The Shakey project also moved from having multiple world models in its first implementation to having a single storage place for predicates of observed data. Any predicate used to form a new plan was rechecked by observation. This development under the selective pressure of experimentation lends credence to the mandate of reactive AI to simplify stored models.

Summary

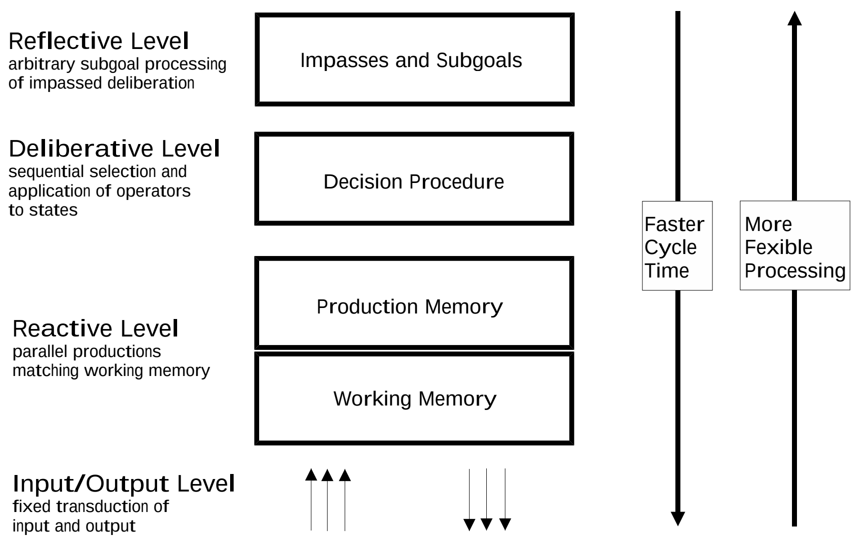
PRS and its related BDI architectures have been much more popular than behavior-based systems in some academic settings. This may be because they are easier to program. They provide significant support for developing the action-selection mechanism, a hierarchical library of plans, and a separate, specialized mechanism for reprioritizing the agent's attention in response to the environment. Particularly when taken over their long-term history, however, these architectures have converged on some of the same important principles such as simplified representations (though not specialized ones) and modularization (at least in the plan libraries.) Current research trends indicate that designing the agent is still a critical problem (see further Chapter 5).

3.7 Soar and ACT-R

Soar [Newell, 1990] and ACT-R [Anderson, 1993] are the AI architectures currently used by the largest number of researchers, not only in AI, but also in psychology and particularly cognitive science. Soar is the most 'cognitive' architecture typically used in U.S. Department of Defense simulations, though even so it is not used extensively due to its high computational overhead. These architectures are fundamentally different from the previously reviewed architectures. Both are also older, dating to the late 1970s and early 1980s for their original versions, but both are still in active development [Laird and Rosenbloom, 1996, Anderson and Matessa, 1998]. The Soar community in particular has responded to the behavior-based revolution, both by participating directly in competitions with the approach [Kitano et al., 1997] and even by portraying their architecture in three layers (see Figure 3-2).

Soar and ACT-R both characterize all knowledge as coming in two types: data or procedures. Both characterize data in traditional computer science ways as labeled fields and procedures in the form of production rules.

Soar is a system that learns to solve problems. The normal procedure is to match its production rules against the current state of the world, find one that is applicable, and apply it. This is automatic, roughly equivalent to the middle or bottom layer of a three-layer architecture. If more than one production might work, or no production will fire, or nothing has changed since the previous application of a production, then Soar considers itself to be at an *impasse*. When Soar encounters an impasse, it enters a new problem space of trying to



Reflective Level
arbitrary subgoal processing of impassed deliberation
Impasses and Subgoals
Deliberative Level
sequential selection and application of operators to states
Decision Procedure
Production Memory
Reactive Level
parallel productions matching working memory
Working Memory
Input/Output Level
fixed transduction of input and output
Faster Cycle Time
More Fexible Processing

Figure 3-2: Soar as a three-layer architecture) [after Laird and Rosenbloom, 1996].

solve the impasse rather than the current goal. The new problem space may use any means available to it to solve the problem, including planning-like searches. Soar has several built-in general purpose problem solving approaches, and uses the most powerful approach possible given the current amount of information. This process is thus something like the way ATLANTIS [Gat, 1991] invokes its top level. Soar, however, allows the process to recurse, so the meta-reasoner can itself hit an impasse and another new reasoning process is begun.

Soar includes built-in learning, but only of one type of information. When an impasse is resolved, the original situation is taken as a precondition and the solution as a procedure, and a new rule is created that takes priority over any other possible solution if the situation is met again. This is something like creating automatic skills out of declarative procedures, except that it happens quickly, on only one exemplar. This learning system can be cumbersome, as it can add new rules at a very high rate, and the speed of the system is inversely related to the number of rules.

Soar addresses the combinatorics of many productions in two ways. First, Soar has the concept of a *problem space*, a discrete set of productions involved in solving a particular goal or working in a particular context. This makes the system roughly hierarchical even in its non-impasse-solving mode. Soar also has carefully crafted optimizations, such as the RETE algorithm [Forgy, 1982] for optimizing production firing. Nevertheless, many industrial users of the system choose not to exploit the learning built into Soar.

ACT-R is essentially simpler than Soar: it does not have the impasse mechanism nor does it learn new skills in the same way. Nevertheless, ACT-R is used extensively for cognitive modeling, and has been used to replicate many psychological studies in decision making and categorization [Anderson, 1993]. ACT-R also faces the difficulty of combinatorics, but it takes a significantly different approach: it attempts to mimic human memory by modeling the probability that a particular rule or data is recalled. Besides the two sets of "symbolic" knowledge it shares with Soar, ACT-R keeps Bayesian statistical records of the contexts in which information is found, its frequency, recency and utility [Anderson and Matessa, 1998]. It uses this information to weight which productions are likely to fire. It also has a noise factor included in this statistical, "sub-symbolic" system, which can result in less-likely alternatives being chosen occasionally, giving a better replication of the unpredictability of human behavior. Using alternatives is useful for exploring and learning new strategies, though it will often result in suboptimal performance as most experiments prove to be less useful than the best currently-known strategy.

Soar, like PRS, is used on an industrial level. However, the fact that it is losing popularity within the cognitive science research community to ACT-R is attributed by researchers largely to the the fact that ACT-R is significantly easier to work with. This is largely because Soar was designed primarily to learn — researchers compared programming Soar to teaching by brain surgery. One simplification made in ACT-R proved to be too extreme. Originally it did not have problem spaces, but over the course of research it was found that hierarchical focusing of attention was necessary to doing anything nearly as complex as modeling human mathematical competences, the primary goal of ACT-R's development team [Anderson, 1993]. ACT-R does not seem to be used in industrial or real-time situations.

Soar has also evolved significantly [Laird and Rosenbloom, 1996]. In particular, when moving to solve problems in a dynamic, real-world domain, it was found to be critical to allow programmers to specify chains or sequences of events explicitly, rather than in terms of simple productions (see further Section 2.3.2). The encoding of time and duration was another major challenge that had to be overcome when Soar moved into robotics — a problem that also needed to be addressed in early versions of PRS and RAP, the middle layer of 3T [Myers, 1996]. ACT-R has not yet been adapted to the problems of operating in a dynamic world: representing noisy and contradictory data, and reasoning about events over time.

Summary

Despite coming from significantly different paradigms and research communities, the long and well-documented histories of Soar and ACT-R exhibit many of the same trends as the other paradigms previously examined. Since both systems at least simulate extreme distribution, (their control is based almost entirely on production rules) they are necessarily very reactive. In fact, Soar had to compromise this feature to be able to provide real-time control. Modularity of control if not data is provided in problem spaces, which can be hierarchical, and Soar now provides for explicit sequential action selection. Soar's generic representations were also found to be not entirely satisfactory. There has been forced specialization of procedure types due to the new benchmark tasks of the 1990's, particularly mobile robotics. Soar still suffers from an extreme overhead in programming difficulty, but is also still in widespread use. ACT-R exploits a niche in the research community as a simpler though similar form of learning system, and has been further specialized to improve its ability to model human cognition.

3.8 Discussion and Recommendations

There have been complaints within the autonomous control community about the over-generation of architectures: what is wanted by users are improvements on systems with which they are already familiar, rather than a continuous diversification. This argument contains some truth. However, it overlooks the perspective stated in the introduction: an agent architecture is a design methodology, and a design methodology is not simply a piece of software. Although some architectural features will conflict, in many cases there is no reason architectures cannot be combined, or one architecture implemented within another. I discuss and demonstrate this in Chapter 5.

Behavior-based architectures began with many of the advantages of modularity and reactive systems, but development of complex control software in them has been hampered by the lack of specific control architectures for supporting hierarchical and sequential ordering of action selection. This is largely due to theoretical opposition: Is a system truly autonomous if it is forced to carry out a plan? Is centralized control biologically plausible? The answer to both of these questions is almost certainly "yes"; see for example Barber and Martin [1999] and Bryson [2000b] respectively for some discussion. Regardless, it can be observed empirically that all autonomous agents do still have and require action selection mechanisms. In behavior-based systems, these systems are often distributed across the behaviors. This may lead to some improvement of robustness, but at a considerable cost in programmability and ease of debugging.

The shift to layered architectures may therefore seem a natural progression for behavior-based AI, but I have some reservations about this model. Many of the systems have the deliberate or constructive planner in ultimate control, which may be intuitive but has not yet been demonstrated to be desirable. The frequent lack of such a layer within this research tradition, and the success of PRS and Soar with something more like a middle layer in primary control of action selection, are good indications that primary action selection should probably emphasize reactive planning rather than deliberation.

A further concern is that layered systems, and indeed some of the more recent behavior-based systems such as HAP [Bates et al., 1992, Reilly, 1996] or the free-flow hierarchy architectures reviewed above, have denigrated the concept of a "behavior" to a mere programming language primitive, thus losing much of the advantage of modularity. Blumberg [1996] addressed this by creating "clusters of behaviors". I believe that these clusters are at the more appropriate level for a behavior.

Behaviors were originally designed as essentially autonomous entities that closely couple perception and action to achieve a particular competence. Unfortunately, they were also conceived as finite state machines, with no internal variable state. In nature, perception is universally accompanied by memory and learning: much of development in mammals is dedicated to learning to categorize and discriminate. This is why I believe that behaviors should also contain state appropriate to their competence, and further that this state and learning should be at the center of behavior decomposition, much as it is at the center of modern object decomposition in object-oriented design.

My primary suggestion for behavior-based AI is further attention to easing the design of action selection. I also suggest experimenting with limited functional modules for abilities such as operating sequential plans and smoothing motor output. This development would be parallel to the nearly universal, though still reductionist, use of state in this paradigm. My recommendation for three-layered architectures is that they look for ways to increase support of modularity in their systems, and that they follow the lead of ATLANTIS for focusing action-selection control in the middle layer. It is still not clear whether it is a better idea for a system to separate action selection from goal manipulation, as Soar and PRS do, rather than using one system for both, as do 3T and most behavior-based architectures. BOD is an example of the latter approach.

PRS is in some ways similar to a three-layer architecture with the emphasis on the middle layer — the building of the plan library. In particular, the software version of PRS distributed by SRI has a fairly impressive GUI for supporting the editing and debugging of this level of intelligence. As might be gathered from the discussion of three-layer architectures above, I consider this type of support very useful.

Unfortunately, PRS still leaves two important levels of abstraction largely unsupported and difficult to manage. The construction of primitives is left to the user, to be done in the language of the PRS implementation; in the case of SRI's implementation, this is a reduced set of common lisp. My suggestions for behavior-based and three-layer architectures applies equally here: primitives should be ordered modularly. They can in fact be built from methods on objects with proprietary state, not shared by the PRS database system. I recognize that this might offend PRS purists, particularly because it might have consequences for the theoretical work on proving program correctness that relies on the database. Nevertheless, I stand by my claim that state is a part of perception. Having some state proprietary to a module should be no more difficult than having an external sensor proprietary to a primitive function; in fact it is exactly equivalent.

The other design level that is surprisingly neglected is the hierarchical organization and prioritization of the various elements of the plan library. Although it is possible to organize plans in the file space (a collection of plans may be saved in a single file) and in lisp by placing them in packages, there is no GUI tool that allows for viewing more than one plan at a time. There is no tool for ordering plans within clusters or agents. Consequently, there is no visual idiom for prioritizing plans that might otherwise be simultaneously able to fire. Prioritization must be handled in poorly documented lisp code that is triggered during the meta-rule section of the main processing cycle. Providing a tool to address this would make it far simpler to program a reactive plan structure like the BRP (see Section 2.3.3 and Chapter 5).

PRS-lite actually addresses both of these complaints, though not in the manner recommended above. It supports "fuzzy" behaviors as primitives, which have their own design methodology, and it attempts to eliminate the need for meta-reasoning or prioritization by a combination of simplifying the task and increasing the power of the goal descriptions [Myers, 1996]. Whether these solutions prove adequate, the fact that these areas are a focus of change indicates agreement on the areas of difficulty in using PRS.

Of the paradigms reviewed, I have the least personal experience with Soar and ACT-R, having only experienced them through tutorials and the anecdotes of programmers. Given their very different background and structure, they appear to have remarkably similar design issues to those experienced under the early behavior-based architectures. This is perhaps unsurprising since both systems are thoroughly distributed. The parallel between the story of the augmenting of subsumption architecture recounted above and the story of the augmentation of Soar with time and sequencing in order to facilitate robot control recounted in Laird and Rosenbloom [1996] is also striking. My suggestions for improving Soar are consequently essentially my recommendations for agent architectures in general: to focus on making agents easier to design via enhancing the ease of use of modular decomposition and pre-programmed action selection, while still maintaining Soar's provision for reactivity and opportunism.

3.9 Conclusions

Every autonomous agent architecture seems to need:

- A modular structure and approach for developing the agent's basic behaviors, including perception, action and learning.
- A means to easily engineer individual competences for complex tasks. This evidently requires a means to order action selection in both sequential and hierarchical terms, using both situation-based triggers and agent-based priorities derived from the task structure.
- A mechanism for reacting quickly to changes in the environment. This generally takes the form of a system operating in parallel to the action selection, which monitors the environment for salient features or events.

In addition to the technical requirements just listed, the central theme of this chapter is that agent architectures are first and foremost design methodologies. The advantages of one strategy over another are largely a consequence of how effectively programmers working within the approach can specify and develop the behavior of the agent they are attempting to build. This stance is not necessarily antithetical to concerns such as biological plausibility or machine learning: natural evolution and automatic learning mechanisms both face the same problems of managing complexity as human designers. The sorts of bias that help a designer may also help these other processes. Similarly, where it is understood, natural intelligence serves as a knowledge source just as well as any other successful agent. This will be discussed further in Chapter 11. The next several chapters will explain how BOD provides for these characteristics, beginning with structured action selection.

Chapter 4

Parallel-rooted, Ordered, Slip-stack Hierarchical (POSH) Reactive Plans

4.1 Introduction

Behavior-oriented design consists of three equally important elements:

- an iterative design process,
- parallel, modular *behaviors*, which determine **how** an agent behaves, and
- *action selection,* which determines **when** a behavior is expressed.

This chapter describes in detail the Parallel-rooted, Ordered, Slip-stack Hierarchical (POSH) reactive plans that underlie the action selection for BOD agents. This chapter describes how to implement POSH action selection directly in a standard programming language. The next chapter discusses implementing key elements of POSH control in other agent architectures. Behaviors and the BOD methodology itself will be covered in the succeeding chapters.

I begin with an aside for the theorists and purists who may still doubt that planning is necessary in a behavior-based architecture.

4.2 Basic Issues

I have already motivated the use of reactive planning both by argument (in Chapter 1) and by induction from the history of agent architectures (Chapter 3). In this section I tie up a few loose ends for researchers who still object either to the term or to planning in principle.

4.2.1 What does 'Reactive Planning' Really Mean?

The terms 'reactive intelligence', 'reactive planning' and 'reactive plan' appear to be closely related, but actually signify the development of several different ideas. *Reactive intelligence* controls a reactive agent — one that can respond very quickly to changes in its situation. Reactive intelligence has sometimes been equated with statelessness, but that association is exaggerated. Reactive intelligence is however associated with minimal representations and the lack of deliberation [Brooks, 1991b, Agre and Chapman, 1990, Wooldridge and Jennings, 1995]. As I said in Chapter 1, reactive intelligence is essentially action selection by look-up.

Reactive planning is something of an oxymoron. The reason the term exists is that early AI systems used (conventional, constructive) planning for action selection, so much so that 'planning' became synonymous with 'action selection'. Many researchers who are generally considered to do reactive AI hate the term 'reactive planning' and refuse to apply it to their own work. But it really just means 'reactive action selection'. When reactive planning is supported by architecturally distinct structures, these structures are called *reactive plans.* As documented in Chapter 3, not all reactive intelligence uses reactive plans.

I embrace the term 'reactive planning' for several reasons. First, it has wide-spread acceptance in the general AI community. Second, some the problems of action selection are sufficiently universal that 'planning' workshops often *are* interesting for reactive planners like myself. Similarly, there are common representational issues for constructed and reactive plans. Finally, the move to actually using explicit reactive plans makes using the term 'reactive planning' seem somewhat more natural, though it is still misleading.

4.2.2 Isn't Having Any Kind of Plans Bad?

I have addressed elsewhere at length [Bryson, 2000b] the concerns of some researchers that any sort of hierarchically structured plan must be insufficiently reactive or not biologically plausible. This belief has been prevalent particularly amongst practitioners of behavior-based or 'new' AI [e.g. Maes, 1991b, Hendriks-Jansen, 1996] and of the 'dynamical hypothesis' of cognitive science [e.g. Kelso, 1995, van Gelder, 1998]. Hierarchical plans and centralized behavior arbitration *are* biologically plausible [Dawkins, 1976, Tanji and Shima, 1994, Hallam et al., 1995, Byrne and Russon, 1998, Prescott et al., to appear]. They are also sufficiently reactive to control robots in complex dynamic domains [e.g. Hexmoor et al., 1997, Bryson and McGonigle, 1998, Kortenkamp et al., 1998] and have been shown experimentally to be as reactive as non-hierarchical, de-centralized systems [Tyrrell, 1993, Bryson, 2000a]. Although they do provide a single failure point, this can either be addressed by standard Multi-Agent System (MAS) techniques [e.g. Bansal et al., 1998], or be accepted as a characteristic of critical systems, like a power supply or a brain. Finally, as demonstrated by coordinated MAS as well as by BOD (e.g. Chapter 6 below), they do not necessarily preclude the existence of semi-autonomous behaviors operating in parallel.

This last point is the most significant with respect to the contributions of this dissertation. Modularity is critical to simplicity of design, and parallelism is critical to a reactive agent. BOD supports all of these attributes.

4.3 Basic Elements of Reactive Plans

Reactive plans provide action selection. At any given time step, most agents have a number of actions which could potentially be expressed, at least some of which cannot be expressed simultaneously, for example sitting and walking. In architectures without centralized action selection such as the Subsumption Architecture [Brooks, 1986] or the Agent Network Architecture (ANA) [Maes, 1990b], the designer must fully characterize *for each action* how to determine when it might be expressed. For engineers, it is generally easier to describe the desired behavior in terms of sequences of events, as this is characteristic of our own conscious planning and temporally-oriented memories.

POSH plans contain an element to describe simple sequences of actions, called an *action pattern*. Action patterns supply quick, simple control in situations where actions reliably follow one from another.

Of course, control is often complicated by the non-determinism of both the environment and an agents' own capabilities. Several types of events may interrupt the completion of an intended action sequence. These events fall into two categories:

1. Some combination of opportunities or difficulties may require the current 'sequence' to be reordered: some elements may need to be repeated, while others could be skipped.

2. Some event, whether a hazard, an opportunity or simply a request, may make make it more practical to pursue a different sequence of actions rather than finishing the current one.

POSH action selection addresses these forms of non-determinism with a fundamental reactive-planning idiom, the Basic Reactive Plan (BRP). The BRP will be formally described in this section; its relevance to reactive planning in general will be examined in Chapter 5.

In POSH, the first situation described above is handled by a BRP derivative called a *competence*. A competence allows attention to be focussed on a subset of plan steps that are applicable in a particular situation. The competence and the action pattern address the second requirement for agent architectures (after modularity) described in Chapter 3 (see page 54), structures to facilitate the appropriate focus of action-selection attention.

The second situation above is addressed by another variant of the BRP, the *drive collection*. A drive collection constantly monitors the environment for indications that the agent should switch between plans. This addresses the third requirement from Chapter 3, the need for an environment monitor or alarm system. In POSH, drive collections are continuous with the rest of action selection; one forms the root of an agent's plan hierarchy.

The remainder of this section provides formal descriptions of sequences and BRPs. The following section will detail the POSH elements refining these basic idioms.

4.3.1 Simple Sequences

One structure fundamental to reactive control is the simple sequence of primitive actions: $\iota_1, \iota_2, \ldots \iota_n$. Including the sequence as an element type is useful for two reasons. First, it allows an agent designer to keep the system as simple as possible, which both makes it more likely to succeed, and communicates more clearly to a subsequent designer the expected behavior of that plan segment. Second, it allows for speed optimization of elements that are reliably run in order, which can be particularly useful in sequences of preconditions or in fine motor control.

Executing a sequential plan involves priming or activating the sequence, then releasing for execution the first primitive act ι_1. The completion of any ι_i releases the following ι_{i+1} until no active elements remain. Notice that this is *not* equivalent to the process of *chaining*, where each element is essentially an independent production, with a precondition set to the firing of the prior element. A sequence is an additional piece of control state; its elements may also occur in different orders in other sequences [see further Section 2.3.2 (rules about bananas), and Lashley, 1951, Houghton and Hartley, 1995].

Depending on implementation, the fact that sequence elements are released by the *termination* of prior elements can be significant in real time environments, and the fact that they are actively repressed by the existence of their prior element can increase plan robustness. This definition of sequence is derived from biological models of serial ordering (e.g. [Henson and Burgess, 1997]).

4.3.2 Basic Reactive Plans

The next element type supports the case when changes in circumstance can affect the order in which a plan is executed. Because this idiom is so characteristic of reactive planning, I refer to the generic idiom as a *Basic Reactive Plan* or BRP[1].

A *BRP step* is a tuple $\langle\pi, \rho, \alpha\rangle$, where π is a priority, ρ is a releaser, and α is an action. A *BRP* is a small set (typically 3–7) of plan steps $\{\langle\pi_i, \rho_i, \alpha_i\rangle *\}$ associated with achieving a particular goal condition. The releaser ρ_i is a conjunction of boolean perceptual primitives which determine whether the step can execute. Each priority π_i is drawn from a total order, but is not necessarily unique. Each action α_i may be a primitive action, another BRP or a sequence as described above.

The order of expression of plan steps is determined by two means: the releaser and the priority. If more than one step is operable, then the priority determines which step's α is executed. If more than one step is released with the same priority, then the winner is determined arbitrarily. Normally, however, the releasers ρ_i on steps with the same priority are mutually exclusive. If no step can fire, then the BRP terminates. The top priority step of a BRP is often, though not necessarily a goal condition. In that case, its releaser, ρ_1, recognizes that the BRP has succeeded, and its action, α_1 terminates the BRP.

The details of the operation of a BRP are best explained through an example. BRPs have been used to control such complex systems as mobile robots and flight simulators [Nilsson, 1984, Correia and Steiger-Garção, 1995, Benson, 1996]. However, for clarity we draw this example from blocks world. Assume that the world consists of stacks of colored blocks, and that an agent wants to hold a blue block[2]. A possible plan would be:

[1]BRPs occur in other architectures besides BOD, see Section 5.2.

[2]This example is due to Whitehead [1992]. The perceptual operations in this plan are based on the visual routine theory of Ullman [1984], as implemented by Horswill [1995]. The example is discussed further in Chapter 6.

$$\Bigg\uparrow \left\langle \begin{array}{crl} \underline{\text{Priority}} & \underline{\text{Releaser}} \Rightarrow & \underline{\text{Action}} \\ 4 & \text{(holding block) (block blue)} \Rightarrow & \textit{goal} \\ 3 & \text{(holding block)} \Rightarrow & \text{drop-held, lose-fixation} \\ 2 & \text{(fixated-on blue)} \Rightarrow & \text{grasp-top-of-stack} \\ 1 & \text{(blue-in-scene)} \Rightarrow & \text{fixate-blue} \end{array} \right\rangle \tag{4.1}$$

In this case priority is strictly ordered and represented by position, with the highest priority step at the top. We refer to steps by priority.

In the case where the world consists of a stack with a red block sitting on the blue block. If the agent has not already fixated on the blue block before this plan is activated (and it is not holding anything), then the first operation to be performed would be element **1** because it is the only one whose releaser is satisfied. If, as a part of some previous plan, the agent has already fixated on blue, **1** would be skipped because the higher priority step **2** has its releaser satisfied. Once a fixation is established, element **2** will trigger. If the grasp is successful, this will be followed by element **3**, otherwise **2** will be repeated. Assuming that the red block is eventually grasped and discarded, the next successful operation of element **2** will result in the blue block being held, at which point element **4** should recognize that the goal has been achieved, and terminate the plan.

This single reactive plan can generate a large number of expressed sequential plans. In the context of a red block on a blue block, we might expect the plan 1–2–3–1–2–4 to execute. But if the agent is already fixated on blue and fails to grasp the red block successfully on first attempt, the expressed plan would look like 2–1–2–3–1–2–4. If the unsuccessful grasp knocked the red block off the blue, the expressed plan might be 2–1–2–4. The reactive plan is identically robust and opportunistic to changes caused by another agent.

The most significant feature of a BRP is that it is relatively easy to engineer. To build a BRP, the developer imagines a worst-case scenario for solving a particular goal, ignoring any redundant steps. The priorities on each step are then set in the inverse order that the steps might have to be executed. Next, preconditions are set, starting from the highest priority step, to determine whether it can fire.

The process of setting preconditions is simplified by the facts that

- the programmer can assume that the agent is already in the context of the current BRP, and
- no higher priority step has been able to fire.

For example, step **3** does not need the precondition (not (block blue)), and no step needs to say “If trying to find a blue block and nothing more important has happened then...”

If an action fails repeatedly, (e.g. grasp-top-of-stack above) then a BRP like the above might lead to an infinite loop. This can be prevented through several means. A competence (described below) allows a retry limit to be set at the step level. Other related systems (e.g. Soar [Newell, 1990]) sometimes use generic rules to check for absence of progress or change. Soar determines lack of progress by monitoring its database.

On a more complete agent level, such ‘rules’ might be modeled as motivations pertaining to boredom or impatience. Section 7.6.3 demonstrates a BOD agent using specialized episodic memory to keep track of its progress; Chapter 10 demonstrates the modeling of an agent’s motivations for this sort of bookkeeping.

4.4 POSH Plan Elements

The previous section explained the basic elements of reactive planning, the sequence and the BRP. In Chapter 5 I will discuss how such fundamental architectural concepts are identified, and will more generally discuss extending existing architectures with them. That chapter concentrates on the BRP, and details both what existing architectures do and don’t support it. I also describe implementing the BRP in several architectures capable of supporting it. This chapter, however, is devoted to explaining POSH action selection in the context of BOD.

4.4.1 Action Patterns

In POSH, I call the simple sequence an *action pattern* (AP). An AP doesn’t differ significantly from the sequence described above. The current implementation of POSH action selection allows action patterns to contain parallel or unordered elements. This change was introduced because such structures seem ubiquitous in the literature, and again serves as documentation to future developers of the fact that there is no particular reason for some ordering. However, I have never yet had reason to use this feature.

4.4.2 Competences

A *competences* is a form of BRP. Like an AP, a competence focuses attention on a particular set of elements suited to performing a particular task. A competence is useful when these elements cannot be ordered in advance.

Competences are archetypical BRPs. The only difference between the POSH competence and the formal definition of BRP described above is that a competence allows for the specification of a limit to the number of retries. This limit can be set individually for each competence step. Thus a *competence step* is really a quadruple $\langle\pi, \rho, \alpha, \eta\rangle$, where η is the optional maximum number of retries. A negative η indicates unlimited retries. I initially experimented with optional ‘habituation’ and ‘recovery’ values which operated on the priority level of the competence step. This was inspired by neural and spreading activation [e.g. Maes, 1991a] models of action selection, which are considered desirable because of the biological plausibility. However, the difficulty of managing the design of such strategies convinced me to be biologically plausible at a different level of abstraction.

Competences also return a value: $\top$ if they terminate due to their goal trigger firing, and $\perp$ if they terminate because none of their steps can fire. These values are irrelevant when POSH action selection is implemented using the version of drive collections described next. However, it becomes occasionally relevant in Section 4.6.3 below.

4.4.3 Drive Collections and the Slip-Stack

A reactive agent must be able to change the current focus of its action-selection attention — to deal with context changes (whether environmental or internal) which require switching *between* plans, rather than reordering steps within them. Some hybrid architectures control this from their 'highest' level, considering the problem the domain of deliberation or introspection. However, BOD treats this problem as continuous with the general problem of action selection, both in terms of constraints, such as the need for reactiveness, and of solution.

The third element type in BOD, the *drive collection*, is also an elaboration of the BRP. A 'step', or in this case, *drive element*, now has five elements $\langle \pi, \rho, \alpha, A, \nu \rangle$. For a drive, the priority and releaser π and ρ are as in a BRP, but the actions are different. A is the *root* of a BRP hierarchy, while α is the *currently active element* of the drive. When a drive collection element is triggered, the α is fired, just as in a standard BRP. However, if the α is a competence and triggers a child, β which is also a POSH element (a competence or action pattern), then α for that drive collection is assigned the value of β. On the other hand, if the α is a competence or action pattern and it terminates, or if this is the first time the drive element has fired, then α is replaced with A, the root of the hierarchy.

This policy of having only one active POSH element assigned to each step of the drive collection is one of the key features of POSH to plans — the *slip-stack hierarchy*. The slip stack defeats the overhead of 'hierarchy bottleneck' that Maes [1991b] warns of. For any cycle of the action selection, only the drive collection itself and at most one other compound POSH element will have their priorities examined[3].

The slip-stack hierarchy improves reaction time by eliminating the stack that might be produced when traversing a plan hierarchy. The slip-stack also allows the agent to occasionally re-traverse its decision tree and notice any context change. I have found this to be a good balance between being persistent and being reactive, particularly since urgent matters are checked per cycle by the drive collection. The slip-stack also allows the hierarchy of BRPs to contain cycles or oscillations. Since there is no stack, there is no *obligation* for a chain of competences to ever terminate.

The fifth member of a drive element, ν, is an optional maximum *frequency* at which this element is visited. This is a convenience for clarity, like the retry limit η on the competence steps — either could also be controlled through preconditions. The frequency in a real-time system sets a temporal limit on how frequently a drive element may be executed. For example, a mobile robot might have its highest priority drive-element check the robot's battery level, but only execute every two minutes. The next highest priority might be checking the robot's sensors, but this should only happen several times the second. Other, lower-priority processes can then use the remaining interspersed cycles (see Section 7.6.2 below).

In a non-real-time system, the frequency is specified in actual number of cycles (see Section 4.5.1 below).

One further characteristic discriminates drive collections from competences or BRPs. Only one element of a competence is expected to be operating at any one time, but for a

[3]This is the most basic form of a slip stack. The current version, which holds a bit more context state, is described below in Section 4.6.3.

drive collection, multiple drives may be effectively active simultaneously. If a high-priority drive takes the attention of the action-selection mechanism, the program state of any active lower drive is preserved. In the case of our robot, if the navigation drive is in the process of selecting a destination when the battery needs to be checked, attention returns to the selection process exactly where it left off once the battery drive is finished. Further, remember that action primitives in our system are not stand-alone, consumatory acts, but are interfaces to semi-autonomous behaviors which may be operating in parallel (see Chapter 6). Thus the action ‘move’ in our robot’s plan might merely confirm or transmit current target velocities to already active controllers. A moving robot does not need to stop rolling while its executive attends to its batteries or its sensors.

4.5 An Example of a Complete POSH Hierarchy

This section illustrates the workings of a POSH system with an example. The next section will show the implementation of this system, and also explains the current, enhanced version of the drive collection used in some of the extended examples of this dissertation.

4.5.1 Managing Multiple Conflicting Goals

Tyrrell [1993] created an extensive artificial life (Alife) test bed for action selection, which he called simply the SE for simulated environment. Tyrrell’s SE postulates a small rodent trying to live on a savannah, plagued by many dangers, both passive and active, and driven to find nourishment, shelter, and reproductive opportunities. The rodent also has very limited sensing abilities, seldom being certain of anything but its immediate environment. It can see further during the day by standing on its hind legs, but this increases its visibility to predators.

Here is the list of goals Tyrrell specifies for agent in the SE:

1. Finding sustenance. In addition to water, there are three forms of nutrition, satisfied in varying degrees by three different types of food.

2. Escaping predators. There are feline and avian predators, which have different perceptual and motion capabilities.

3. Avoiding hazards. Passive dangers in the environment include wandering herds of ungulates, cliffs, poisonous food and water, temperature extremes and darkness. The environment also provides various forms of shelter including trees, grass, and a den.

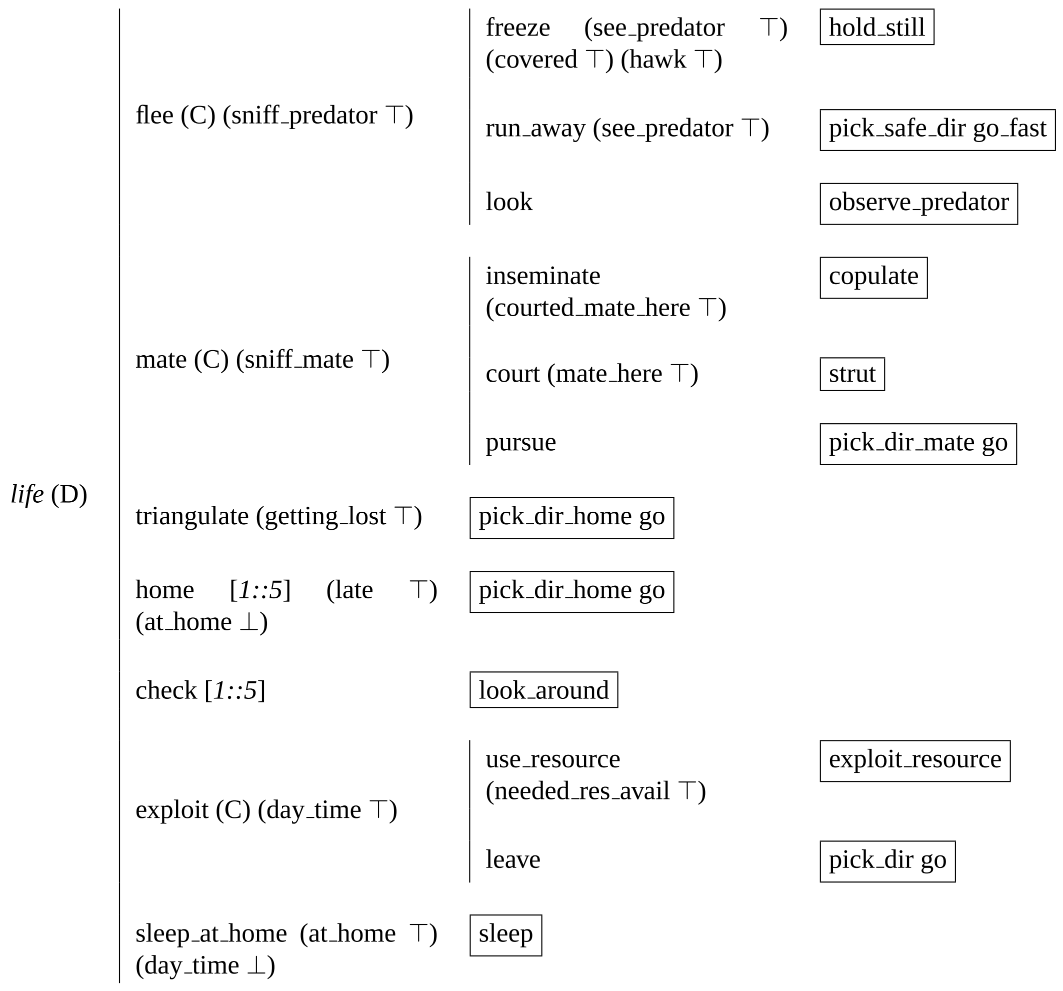
4. Grooming. Grooming is necessary for homeostatic temperature control and general health.

5. Sleeping at home. The animal is blind at night; its den provides shelter from predators and other hazards, and helps the animal maintain body temperature while conserving energy.

6. Reproduction. The animal is male, thus its reproductive task is reduced to finding, courting and inseminating mates. Attempting to inseminate unreceptive mates is hazardous.

These problems vary along several axes: homeostatic vs. non-homeostatic, dependency on external vs. internal stimuli, periodicity, continual vs. occasional expression, degree of urgency and finally, whether prescriptive or proscriptive with regard to particular actions. In addition to these problems, the environment is highly dynamic. Food and water quantities, temperature and light vary, and animals move. Sensing and action are uncertain. Perception in particular is extremely limited and severely corrupted with noise; the animal usually misperceives anything not immediately next to it, unless it chooses to spend time and expose itself by rearing up and “looking around” in an uncovered area.

The success of the rodent is considered to be the number of times it mates in a lifetime. This is highly correlated with life length, but long life does not guarantee reproductive opportunities.


life (D)
flee (C) (sniff_predator ⊤)
freeze (see_predator ⊤) (covered ⊤) (hawk ⊤)
hold_still
run_away (see_predator ⊤)
pick_safe_dir go_fast
look
observe_predator
mate (C) (sniff_mate ⊤)
inseminate (courted_mate_here ⊤)
copulate
court (mate_here ⊤)
strut
pursue
pick_dir_mate go
triangulate (getting_lost ⊤)
pick_dir_home go
home [*1::5*] (late ⊤) (at_home ⊥)
pick_dir_home go
check [*1::5*]
look_around
exploit (C) (day_time ⊤)
use_resource (needed_res_avail ⊤)
exploit_resource
leave
pick_dir go
sleep_at_home (at_home ⊤) (day_time ⊥)
sleep

(4.2)

Figure 4-1: Priorities for a beast that juggles contradictory goals in Tyrrell’s SE.

Plan 4.2 in Figure 4-1 has been demonstrated not only adequate in that environment, but significantly better than any of the action-selection mechanisms Tyrrell himself tested [Bryson, 2000b,a]. Here I am using a different notation for the plan in order to make the full hierarchy apparent. The vertical lines are BRPs with priority directly related to height on the page, as usual. *D* indicates a drive collection, *C* a competence. Each branch is labeled with its name, as in the priority lists following plan 7.4 (green-on-green) above. This is followed by any preconditions for that branch. Two of the drive-collection elements also have a scheduling factor: since this was a discrete time-step Alife system rather than a real time system, scheduling is per *N* cycles, so [1 :: 5] means that if this element is the highest priority, and has not fired within the past 5 cycles, it will fire. Boxes indicate action patterns, which for the C++ implementation of my control system were the only structure that could hold primitives, so they are sometimes only one element long.

Because Tyrrell [1993] focussed on action selection, his environment provides primitives not only for action and perception, but also for roughly keeping track of its own location relative to its den. Like the other senses, this is noisy and the animal can get lost if it spends too long away from its home without taking time to learn landmarks. The only additional behaviors I added were one for choosing the direction of motion, which would avoid as many hazards as possible in a particular context, and one for exploiting the benefits of a particular location, which ate, drank, basked or groomed as was opportune and useful. Some of these had initially been part of the control hierarchy, but the savannah proved sufficiently lush that specific goals relating to hunger, thirst or cleanliness proved unnecessary complications.

The most difficult part of the decision process for the rodent is determining whether to attend to predators when one was sensed. If every vague sensing is attended to, the animal gets nothing else done, particularly not mating which is a multi-step process dependent on staying near another roaming agent. But if the wrong sightings are ignored, the animal gets eaten. The solution above has the animal attempt to escape if it is fairly certain there is a predator around, and try to increase its certainty if it is more marginally certain. Since the relationship between these variables was nonlinear (and there were a separate pair for the two kinds of predators) I used a sort of two-generation genetic algorithm to set the variables: I set the variables randomly on a large number of animals who got one life span, then took the top 12 performers and ran them through 600 lifespans to see which performed consistently the best. Here again is an example of having the developer doing the learning rather than the agent.

For a more complete account of the development process for this agent see Bryson [2000b].

4.5.2 Exploiting the Slip-Stack Hierarchy

The slip-stack hierarchy is exploited by any plan with nested competences (e.g. Plan 7.14 below) in that response time is saved because only the currently-active competences' priorities are examined. The slip stack is also designed to enable cyclic graphs in POSH hierarchies.

I must admit that I have not found it strictly necessary to use this capacity yet. I have used it only to alternate between competences in a pilot study of emulating rat navigation run in a robot simulator. But this method of creating oscillation was gratuitous; it could also have been done within a drive hierarchy (see Chapter 10 for an example.) I believe the reason I have not used competence cycles yet is a combination of two considerations — the sorts of domains I have been working in, and the deliberately shallow level of control complexity. For example, natural language is a domain where complex control occurs in redundant structures, but I have so far done no serious natural language applications. Also, ALife simulations such as Tyrrell's above allow complexity to be masked as a single primitive. In a more realistic simulation, courting and inseminating would probably be chained competences, each with multiple sorts of indications for transitions between behavior patterns from both the agent itself and from its partner in copulation.

The main reason to use chained competences rather than a single master competence or separate drive elements is to force an ordering of competences. Two competences that are both under the same drive element can never be executed at the same time. Using chains of competences are in this way analogous to using sequences rather than production rules (see Section 2.3.2), only with more complex elements. APs should generally not be used to sequence actions that take longer than a couple of hundred milliseconds, because they monopolize the action selection and make the agent less reactive. Actions of a long or uncertain duration should be represented using competences (see Section 6.2). Also, APs of course cannot have sub-components with flexible, BRP-like ordering. If a BRP needs to be sequenced, then they should be sequenced using competence chaining.

4.6 The Inner Workings of POSH Action Selection

For thoroughness, this section begins with a pre-history of POSH control. I then describe the simple implementation of POSH described above and used in the experiments described in Chapter 7. Next I describe a more recent elaboration of POSH structure that was used for the examples in Chapters 9 and 10. This section concludes with a summary.

4.6.1 Early History and Caveats

In Chapter 3 I argue that the record of changes to an architecture is an important source of information about the fundamental nature of the problem of designing intelligent agents. For this reason, this section includes some documentation of the history of my architectures implementing POSH design. This subsection is not critical to understanding current POSH systems.

There have been four implementations leading to the current version of POSH action selection. The first was in Common Lisp Object System (CLOS), the second in object-oriented perl (ver. 5.003), the third in C++, and the fourth is in CLOS again. The first was called Braniff, and the third Edmund. The fourth is a simplified version of SoL, which is described in Section 12.3.2. The language shifts were partly motivated by circumstances, and partly by the fact that I like changing languages on rewrites, because it forces a complete rewrite and rethinking of assumptions as you shift between programming paradigms. I also strongly dislike strong typing, because it slows down development significantly and inhibits creativity. Despite the fact that the version of perl I was using was very buggy, and the initial research platform (a mobile robot) for the perl and the C++ versions of the architecture were the same, and that I had programmed professionally in C and C++ for years, development of behavior libraries slowed by a factor of approximately 5 when I switched to C++. Fortunately, most of the difficult libraries were already constructed before I made the change, and translation was relatively simple.

In Braniff, each action pattern was divided into four sub-sections, any of which could be nil. These subsections were: triggering perceptual checks, perceptual actions (those that changed attention, but not the environment), pre-motor perceptual checks, and motor actions. I simplified this to a single homogeneous sequence that terminated if any element failed (whether a perception check or an action). This proved a poor idea because of scheduling. The sequences are now broken into two parts — trigger sequences, where all elements are checked atomically within the scheduler, and action patterns, which allow a control cycle between the execution of each element. An obvious extension to my current system would be the creation of another primitive class, trigger-actions, which would be the only kind of actions that could occur in triggers, and would be constrained to operate very quickly. However, as I stated earlier, I dislike unnecessary typing, so have left timing issues as a matter of heuristics.

Braniff did not initially have drive collections, only competences. The competence elements each had a fixed priority between 0 and 100. They could also have a value by which their priority was reduced so that an element that was tried repeatedly would 'habituate' and allow other strategies to be tried, or eventually allow the entire competence to fail. When I introduced drive collections, they had a similar system, but also with a recovery factor, so that over time a drive element might be operated again. Competence elements never recover, but each time a competence is first invoked by a drive its elements start over from the original set of priorities. During the development of Edmund, all of these systems were simplified. Habituation of a competence element is now discrete — it simply will not fire after a fixed number of attempts. Scheduling in the drive collections is now in terms of frequency. The reason for these simplifications is that they are much simpler to design and manage. Shifting priority order lead to unnecessary complications in most situations. Where variable motivation levels are useful (as in the drive system of Tu [1999] or Grand et al. [1997]) they can be easily modeled in the behavior system (see Chapter 10).

4.6.2 A Simple Implementation of POSH Control

This section documents in pseudo-code the implementation of the drive collection as described in Section 4.4.3. This is the final version of the third, C++ implementation of POSH mentioned in the previous section, sometimes referred to as Edmund [e.g. Bryson and McGonigle, 1998].

For every cycle of the action scheduler, the following code is executed. 'This_de' means 'this drive element', α and A are as defined in Section 4.4.3.

```
@drive-elements = priority_sort (elements (drive_root));

do (forever) { // unless 'return' is called
```

```
    result = nil;
    for this_de in @drive-elements {
        if (trigger (this_de) and not-too-recent (frequency (this_de))) {
        if goal (this_de)
            { return (succeed); }
        // if α is a primitive, will get fail or succeed
         // if it is another POSH element, then execute it next time.
        result = execute ( α(this_de));
        if (typeof (result) == (‘competence’ or ‘action pattern’))
            { α(this_de) =   // slip stack -- replace own alpha...
              // with a local copy of control state (see below)
              executable_instance_of (result); }
        if (result == (fail or succeed))
            { α(this_de) = A(this_de); }  // restart at root
        // otherwise, will execute same thing next time it triggers
        }
    } // end of for
    if (result == nil)  // nothing triggered
        { return (fail); }
}  // end of do
```

An *executable_instance_of* a POSH composite type is just an instance with disposable state for keeping track of things like which element was last executed in an action pattern, or how many times an element has been executed in a competence (if that element habituates.) Executing an action pattern then is just:

```
result = execute (element (instance_counter));
instance_counter += 1;
if ((result == fail) or (instance_counter == sequence_length))
    { return (result); }
  else
    { return (continue); }
```

And a competence looks very much like the drive collection, except simpler (assume my_elements are pre-sorted):

```
for this_ce in @my_elements {
    if (trigger (this_ce) and not-too-many (number_tries (this_ce))) {
        if goal (this_ce)
            { return (succeed); }
        if (typeof(α(this_ce)) == (‘competence’ or ‘action pattern’))
            { return ( α(this_ce)); }
          else
            { return (execute( α(this_ce)));}
        }
} // end of for

return (fail);  // nothing triggered
```

The maximum cycle rate for this architecture in Gnu C++ on a 486 running Linux with primitive actions doing disk writes was over 3,000Hz. On the other hand, a primitive that takes time can obviously slow this down arbitrarily. In the robot example in Section 7.6 below, with sonar sensing done at 7Hz, the cycle rate was about 340Hz. This was still more than enough to operate the robot plans shown in Chapter 7 (Plans 7.13 and 7.14), but it indicates that if the robot had to do some heavy thinking, the sonar update task should have been moved to another process.

4.6.3 POSH Control with an Action Scheduler

My experiences with programming robots in this architecture (documented in Section 7.6.3) eventually motivated me to change the way drive collections operate. In particular, Plan 7.13 assumes that a competence element can fail if the competence it calls fails the first time through. In the architecture above, this clearly isn't true.

Another motivation was the behavior of action patterns. If a robot suspends a drive-element to do something very rapid, such as a sonar check or a speech act, then the action pattern should continue from where it left off, so that the POSH action selection can simulate pseudo-parallelism. However, if the robot instead diverts its attention to a long activity, such as disengaging from an impact, restarting an action pattern from the middle may be irrelevant. This isn't disastrous — if the action pattern fails and if it is still needed it can be restarted. However, it is evidence that the solution to the first problem might not be based simply on increasing the fixed size of a slip stack, but rather on allowing a POSH element to persist for only *a fixed amount of time* without action selection attention. This solution no longer strictly guarantees a maximum number of compound elements will be checked. However, it is more psychologically plausable, in that it models that some sort of priming or activation remains for a period after a decision is made.

I found a device that allowed me to implement this form of action selection in Ymir [Thórisson, 1999]. (See Sections 5.5.1 and 12.3.2.) The current version of POSH action selection does not retain all of Ymir, but only two elements: a *schedule* and a *bulletin board*. The bulletin board is a form of short-term recent episodic memory for action selection. It is also useful for debugging, this is discussed further in Section 8.4.3.

The Ymir control cycle was essentially two-phased. On every cycle:

- add anything needed to the schedule, then

- pass through the schedule

– executing anything pending that you can execute, and
– deleting anything pending that has timed out.

Here is the modified version of POSH that takes advantage of the scheduler:

```
@drive-elements = priority_sort (elements (drive_root));

do (forever) { // unless ‘‘return’’ is called
    result = nil;
    for this_de in @drive-elements {
        if (trigger (this_de) and not-too-recent (frequency (this_de))) {
            if goal (this_de)
                { return (succeed); }
            result = execute_schedule (this_de);
            if (result == nil)
                { add_to_schedule (executable_instance_of ($A$(this_de)));
                  result = $\top$;  // indicate something happened
                }
    } // end of for
    if (result == nil)  // only happens if nothing triggered
        { return (fail); }
}  // end of do
```

Notice that the drive no longer needs to keep track of α — the current action is now maintained on the action scheduler. On the other hand, competences and action patterns now need a new bit of state, a *timeout*, to indicate how long they should be kept on the schedule.
If you put an action pattern on the schedule, you do something like this:

```
put_on_schedule (executable_instance_of (element[0]));
for (iii = 1; iii < action_pattern_length; iii++) {
    temp = executable_instance_of (element[iii]);
    add_precondition (temp, succeeded (element[iii - 1]));
    put_on_schedule (temp);
} // end of for
```

Here we assume that either executable_instance_of or put_on_schedule somehow reads the timeout for the action-pattern, and computes the time limit from the current time. If an element fails, the other elements will sit for a while, but will be cleaned up at some time. Meanwhile, since nothing could fire, their drive_element may have already restarted their root.

A competence is significantly different. It simply puts itself (not any of its elements) on the schedule when first invoked. When executed, it does this:

```
for this_ce in @my_elements {
    if (trigger (this_ce) and (not-too-many (number_tries (this_ce)))) {
        if goal (this_ce)
            { record (succeed); }
          else
            { temp_el = executable_instance_of (α(this_ce)));
              add_to_schedule (temp_el);
                temp_comp = executable_instance_of (me);
              add_precondition (temp_comp, terminate (element[iii - 1]));
              add_to_schedule (temp_comp); }
    } // end of if
} // end of for

record (fail);  // nothing triggered
```

Notice outcomes are now recorded (on the bulletin board) rather than returned. Also, notice that the parent waits for its child to terminate, then automatically will go through this same process again. If the child is another competence, it is unlikely to terminate before the parent is cleaned up. However, it can, and in that case it will continue without the drive-element needing to restart from the beginning.

One disadvantage of this system is that now execution of nearly all actions is dependent on searching this short-term memory, the bulletin board. As such, performance of this system becomes highly dependent on the duration of this short-term-memory. For debugging, I often have the short term memory set to 10 minutes, but for production I have it set to only 2 seconds. The maximum cycle rate for this architecture in Xanalys (formerly Harlequin) LispWorks on a 400MHx MMX PII with actions some of which wrote to disk is about 50 Hz if the bulletin board only trims entries after 2 minutes, but it runs over 260 Hz with trimming at half a second. Even at half-second trimming, checking the bulletin board still takes nearly 70% of the lisp system time.

These rates could doubtless be further optimized. For example, I could more closely follow the example of Ymir [Thórisson, 1999] and use more than one bulletin board for different sorts of events. This might drop the bulletin board search time by an order of magnitude. But speed hasn't been a high priority issue yet, even though I am now running some real-time, multi-agent simulations. Currently, there is more value in building debugging tools than reducing cycle time.

4.6.4 Summary: Critical Features of POSH Action Selection

I have presented two different implementations of POSH action selection in this section. They are both valid, and have different strengths. The former is faster and cleaner, the latter is slightly more biologically plausible, and has good handles for debugging. Either will support the BOD process. The critical aspects of POSH are in both: it supports the

BRP, it limits stack growth and allows cycles in its hierarchy, it supports pseudo-parallelism and the changing of attention to higher priorities, and it restarts a plan hierarchy from its root if it terminates.

4.7 Conclusions

The most expedient solution to the design problem of reactive planning is to categorize action selection into three categories: things that need to be checked regularly, things that only need to be checked in a particular context, and things that one can get by not checking at all. These categories correspond to the three types of POSH plan elements: drive collections, competences, and action patterns.

This chapter presented a detailed description of POSH action selection, including formal descriptions of the fundamental elements, an extended example of a control hierarchy, implementation details and a design history. In the next chapter I discuss the relationship between POSH control and other architectures with particular focus on the BRP. More generally, that chapter demonstrates how to transfer technological advances between agent architectures. The following chapters will address the other aspects of BOD modular behaviors and specialized learning, and the agent development process.

Chapter 5

Architectural Idioms: POSH Action Selection in Other Architectures

5.1 Introduction

In order for a field as a whole to advance, key discoveries must be communicated throughout the field's research community. In Chapter 3 I demonstrated some cross-paradigm analysis, but the procedure was time consuming, and even so omitted many architectures.

In this chapter I propose a meta-methodological strategy for the problem of incorporating the advances of new architectures into established development efforts. My proposal is simple: a researcher, after having developed a new architecture, should express its major contributions in terms of one or more of the current 'standard' architectures. The result of this process is a set of differences that can be rapidly understood by and absorbed into established user communities.

In Chapter 3 I discussed the general applicability of the principles behind the BOD methodology. Since BOD modularity rests on object-oriented design, if there is an impediment to fully applying BOD in a particular architecture, it is usually the lack of POSH action selection. In this chapter, as a demonstration of the general meta-methodology of transmitting information across architectures, I make an example of implementing one of the key fundamentals of POSH control. Specifically, I describe my experiences implementing BRPs in three different architectures — Ymir Thórisson [1999], PRS-CL Myers [1997,1999] and JAM Huber [1999], a Java-based extension of UM-PRS; I also discuss a hypothetical implementation in Soar[1].

This chapter begins with a discussion of to what extent BRPs already exist in other architectures. It concludes with a discussion of the roles of architecture, methodology, and toolkit in the problem of intelligent agent design.

[1]This chapter makes frequent reference to architectures that were introduced and described in Chapter 3.

5.2 The BRP in Other Reactive Architectures

The operation of the BRP seems so central to reactive planning, that one would expect it to be expressible in most reactive architectures. And indeed, the BRP has been developed several times, with varying levels of independence [Fikes et al., 1972, Nilsson, 1994, Correia and Steiger-Garção, 1995, Bryson and McGonigle, 1998]. Presumably Nilsson [1994] was inspired at least in part by his own previous work with Fikes et al. [1972], however there is a considerable lag between these developments. I personally was not aware of Fikes' work before I read about Shakey [Nilsson, 1984] in 1995[2]. I found Nilsson's teleo-reactive plans [Nilsson, 1994] and Correia's action selection system shortly thereafter.

Yet despite the fact that some of these implementations have had considerable influence in AI, it is not yet a common attribute of planning systems calling themselves 'reactive.' As this chapter will demonstrate, I have found this feature surprisingly lacking in several architectures, and totally inexpressible in others. In effect, architectures using plan-scripting languages like PRS [Georgeff and Lansky, 1987] or RAP [Firby, 1995] seem to expect that most of behavior can be sequenced in advance, and that being reactive is only necessary for dealing with external interruptions by switching plans. On the other hand, architectures such as subsumption [Brooks, 1991b] or ANA [Maes, 1990b] expect that there is so *little* regularity in the arbitration of behavior that all actions must be considered for execution at all times. The only architecture that I have found with a well-established research community and that works at a similar level of reactiveness to POSH is the teleo-reactive architecture Nilsson [1994].

5.3 Architectures, Idioms and Progress in Agent Design

Because, as I argued in Chapter 1, the development of production-quality agents always requires the employment of human designers, there is a high cost associated with switching architectures during a project. In fact, there is a high cost even for making changes to an architecture. The engineers responsible for building systems in an upgraded architecture require time to learn new structures and paradigms, and their libraries of existing solutions must be ported to or rewritten under the new version. These problems alone deter the adoption of new architectures. They are further exacerbated by the cost, for the architect, of creating documentation and maintaining a production-level architecture, and for the project manager, of evaluating new architectures. Nevertheless, new architectures often hold important insights into the problems of designing intelligence. In this section, I discuss somewhat formally the circumstances under which insights from one architecture can be transfered to another.

Consider the problem of expressing a feature of one architecture in another. There are two possible outcomes. A feature f_1, of architecture A_1 may be completely expressible in

[2]Leslie Pack Kaelbling first saw the similarity between my work and Fikes' triangle tables, and recommended I read the Nilsson [1984]. Triangle tables are essentially BRPs automatically expanded out of sequential plans. Effectively, a new, shorter plan is generated that starts at each step of the initial sequence and continues to the goal. This allows the plan to be restarted at any juncture if an action fails, and accounts for the tables' triangular shape.

A_2. Assuming that this expression is not trivial (e.g. one line of code) then A_1 *constrains* A_2 in some way. On the other hand, if f_1 cannot be expressed in A_2 without altering the latter architecture, then A_1 *extends* A_2. These conditions are not mutually exclusive — two architectures generally both constrain and extend each other, often in multiple ways. Identifying these points of difference allows one architecture to be described in terms of another.

When I speak of the relative expressive power of two architectures, I am not really comparing their linguistic expressibility in the classical sense. Almost all agent architectures are Turing-complete; that is, a universal computing machine can be constructed within almost any agent architecture. This universal computing machine can then be used as an implementation substrate for another agent architecture. So, in the formal sense, all agent architectures are inter-reducible. I am concerned instead with the kinds of computational idioms that are *efficaciously expressible*[3] in a particular architecture. In this sense, an architecture A_1 may be considered to extend A_2 when there is no way to express reasonably succinctly the attributes of A_1 in A_2.

If, on the other hand, a feature f_1 of A_1 can be translated into a coding i^{f_1} of A_2 with reasonable efficiency, then that coding i^{f_1} is an *idiom*. As I explained above, the existence of such an idiom means A_1 constrains A_2. This notion of constraint may seem counterintuitive, because new features of an architecture are usually thought of as extensions. However, as I argued in the introduction, extending the capabilities of the developer often means reducing the expressibility of the architecture in order to biases the search for the correct solution to the problem of designing an agent.

Although in my example A_1 is constrained relative to A_2 due to feature f_1 of A_1, adding the idiom i^{f_1} is unlikely to constrain A_2. A_2 retains its full expressive power so long as the use of i^{f_1} is not mandatory. For an example, consider object-oriented programming. In a strictly object-based language such as smalltalk, OOP is a considerable constraint, which can consequently lead to effective and elegant program design. In contrast, C++ has added the features of objects, but still allows the full expression of C. Thus, for the C++ programmer, the elegance of OOP is an option, not a requirement.

An idiom is a compact, regularized way of expressing a frequently useful set of ideas or functionality. I borrow the notion of idiom both from natural language and computer science, though in computer science, the term ‘idiom’ (or ‘design pattern’) is sometimes used for a less rigorous mapping than I mean to imply. An architecture can be expressed as a set of idioms, either on programming languages or sometimes on other architectures. Researchers seeking to demonstrate that their architecture makes a contribution to agent design might do well to express their architecture in terms of idioms in familiar architectures. In this way, the architecture can be both readily understood and examined. I demonstrate this approach in the following two sections.

It is important to observe that this meta-methodology is different from though related

[3]In computational complexity theory, the notion of reducibility is augmented with the asymptotic worst case complexity of the reduction. So, for example, in the theory of NP-completeness, polynomial-time reducibility plays a crucial role. The notion of efficacious expressibility does not rely on any criterion so sharply defined as the computational complexity of the reduction computation, but is intended to evoke a similar spectrum of reduction complexity.

to the practice of publishing extensions of architectures. First, I do not discourage the practice of building entirely new architectures. If an architecture has been built as an entity, it is more likely to have significant variation from standard architectures, potentially including vastly different emphases and specializations for particular tasks. These specializations may turn out to be generally useful contributions, or to be critical to a particular set of problems. Second, the idiomatic approach emphasizes the search for generally applicable strategies. Generality here does not necessarily mean across all possible problems, but it should mean an idiomatic solution relevant across a number of different underlying architectures. Thus even if an idiom is developed in the context of a well known architecture, it would be useful if, on publication, the researcher describes it in terms of general applicability.

5.4 Identifying a Valuable Idiom

How does one find a useful architectural feature, and how does one distinguish whether it is worth expressing as an idiom? Features are distinguished by the methodology described in Section 5.3, by comparison and reduction of an architecture to one or more others. Features can be idioms if they can be expressed in other architectures. Idioms are valuable within an architecture if they perform useful functions; they are valuable to the community if they are not yet regular features of existing architectures or methodologies. In this section, I illustrate the process of identification by going through the history of the identification of the BRP in my own research. I also give two counter-examples from the same source.

As introduced in Chapter 2 and described in Chapter 4, the three structural elements of BOD's POSH reactive plans are action patterns, competences and drive collections. To illustrate the search for valuable idioms, I consider a reduction of each of these three features in the context of Subsumption Architecture (SA) [Brooks, 1986], the Agent Network Architecture (ANA) [Maes, 1991a], the Procedural Reasoning System (PRS) [Georgeff and Lansky, 1987] and Soar [Newell, 1990].

Although deceptively simple, action patterns actually required extensions to the original versions of each of the above architectures except PRS. Within SA, a sequence can only be expressed within a single behavior, as part of its FSM. As described in Chapter 3, when the need for behavior sequencing was discovered, a mechanism for suppressing all but one behavior was developed [Connell, 1990]. ANA explicitly represents the links of plans through chains of pre- and post-conditions, but with no privileged activation of a particular plan's elements. This sequencing strategy is inadequate [Tyrrell, 1993], and has been improved in more recent derivative architectures [Rhodes, 1996, Blumberg, 1996]. Soar initially represented sequences only as production chains. This mechanism is insufficient in real-time applications. The problem has now been addressed with a dedicated sequencing mechanism that monitors durations [Laird and Rosenbloom, 1996]. PRS, on the other hand, has a reactive plan structure, the Act, which allows for the coding not only of sequences, but of partial plans. Although an action pattern could therefore be seen as an idiom on an Act, I have no strong reason to argue that this particular reduction in power is useful. In conclusion, there is evidence from the history of multiple architectures that an action pattern is an important feature. However, it is not one that can easily be implemented as an idiom, because it generally extends rather than constrains architectures that lack a trivial way to express it.

As for a parallel mechanism for allowing attention shifts, some implementation of this feature is ubiquitous in reactive architectures (again, see Chapter 3). SA assumes that all behaviors operate in continuous parallel, and can always grasp attention. ANA is similar — each behavior is always evaluated as a possible next act. PRS addresses both control and reactivity on each cycle: it first persists on the currently active plan, then engages meta-reasoning to check whether a different plan deserves top priority. Soar also seems to have struck a balance between persistence and reactivity. Being production based, it is naturally distributed and reactive, similarly to SA and ANA. Persistence is encouraged not only by the new seriating mechanism mentioned above, but primarily by clustering productions into *problem spaces*. A problem space is actually somewhat like a BRP in that it focuses attention on a subset of possible productions. Because all of these architectures have means for monitoring the environment and switching attention, introducing drive collections on top of these mechanisms does not have clear utility.

The BRP is a different matter[4]. First, despite the several examples from the planning literature mentioned earlier, it is not present as a single feature in any of these four architectures. To implement them in SA or ANA would require extensions, for much the same reason as the implementation of sequences requires extensions. There is no intrinsic way to favor or order a set of expressed actions in either architecture except by manipulating the environment. PRS and Soar, on the other hand, contain sufficient ordering mechanisms that implementing a BRP idiom should be tractable.

In summary, the value of an idiom is dependent on two things. It must be expressible but not trivially present in some interesting set of architectures, and it must be useful. Utility may be indicated by one's own experience, but also by the existence of similar features in other architectures. With respect to the BRP, it is present in several architectures in the literature, and I have independently found its programming advantages sufficient to lead me to implement it in several architectures besides BOD. The next section documents these efforts.

5.5 Expressing BRPs in Other Architectures

As I explained in Chapter 4, the BRP along with the sequence are fundamental to POSH action selection. As the previous section showed, the BRP (at least in its basic, competence-like form) is a good candidate for an idiom providing an architecture already has a viable mechanism to support simple sequencing. In this section, I document the implementation of feature as an idiom on a number of architectures. Section 5.6 will discuss how to best exploit such new architectural features.

[4]The fact drive collections are in fact BRPs is irrelevant at this point. They were eliminated in the previous paragraph due to their function, not their mechanism.

5.5.1 Ymir

My first effort to generalize the benefits of POSH action selection was not in a widely-used standard architecture, but was rather in another relatively recent one, Ymir ([Thórisson, 1999] see also [Bryson and Thórisson, 2001]). Ymir is designed to build complex agents capable of engaging in multi-modal dialog. A typical Ymir agent can both hear a human conversant and observe their gestures. The agent both speaks and provides non-verbal feedback via an animated character interface with a large number of degrees of freedom. Ymir is a reactive and behavior-based architecture. Its technical emphasis is on supporting interpretation of and responses to the human conversant on a number of different levels of time and abstraction. These levels are the following:

- a "reactive layer", for process-related back-channel feedback and low-level functional analysis. To be effective, this layer must be able operate within 100 millisecond constraints,

- a "process control layer", which deals with the reconstruction of dialogue structure and monitoring of process-related behaviors by the user, and

- a "content layer", for choosing, recognizing, and determining the success of content level dialogue goals.

Ymir also contains a key feature, the *action scheduler*, that autonomously determines the exact expression of behaviors chosen by the various layers. This serves to reduce the cognitive load, accelerate the response rate, and ensure that expressed behavior is smooth and coherent.

Although Ymir excels at handling the complexity of multimodality and human conversations, it does not have a built in capacity for motivation or long-term planning. Ymir is purely reactive, forming sentences for turn taking when prompted by a human user.

Because of Ymir's action scheduler, the implementation of drives, action patterns and BRPs was significantly different from my then-current implementation of POSH action selection (see Section 4.6.2). The scheduler could be relied on to "clean up" behaviors that had been triggered but were not expressed after a timeout, but it could also be signaled to allow their lifetimes to be renewed.

I have already detailed my ultimate solution to implementing BRPs in this context (see Section 4.6.3). Ymir actually lacked sequences, bit I implemented them simply by posting all of the elements to the schedule, each with a unique tag, and all but the first with a precondition requiring that its predecessor complete before it began operating.

The BRP is implemented as an Ymir behavior object which is posted to the action-scheduler. When executed, the BRP selects a step (as per Section 4.3.2) and adds the step to the scheduler. The BRP then adds itself to the scheduler with the termination of its child as a precondition. The original copy of the BRP then terminates and is cleaned up by the scheduler. If the child or its descendents maintain control for any length of time, the 'new' parent BRP will also be cleaned up (see further Section 4.6.3). Otherwise, the BRP persists in selecting plan elements until it either terminates or is terminated by another decision process.

The implementation of POSH action selection in Ymir was so clean, and the action-scheduler feature so apparently useful, that I wound up adopting these features back into my own POSH implementation after finishing work with Ymir, as documented in Section 4.6.3.

5.5.2 PRS-CL

My next implementation of BRPs came during a project exploring the use of reactive planning in dialogue management. Because this was a relatively large-scale project, a well-established architecture, PRS, was chosen for the reactive planning. Because of other legacy code, the language of the project was Lisp. Consequently, we used the SRI implementation of PRS, PRS-CL [Myers, 1997,1999]. PRS-CL provides not only an implementation of PRS, but also documentation and a set of GUI tools for developing and debugging PRS-CL agent systems. These tools are useful both for creating and debugging the main plan elements, the Act graphs.

Acts are roughly equivalent to action patterns described above, but significantly more powerful, allowing for parallel or alternative routes through the plan space and for cycles. I initially thought that a BRP would be best expressed within a single Act. However, there is no elegant way to express the inhibition of lower priority elements on an Act choice node. Instead, I implemented the BRP as a collection of Acts which are activated in response to the BRP's name being asserted as a goal. This results in the activation of all the Acts (steps) whose preconditions have been met.

PRS-CL has no built-in priority attribute for selecting between Acts. Selection is handled by meta-rules, which operate during the second half of the PRS control cycle (as mentioned in Section 5.4). I created a special function for the meta-rule that selects which of the Acts that have been triggered on a cycle is allowed to persist. This function is shown in Figure 5-1.

The BRP function I built for PRS-CL depends on a list of priority lists, where each priority list is associated with the name of the BRP. This is somewhat unfortunate, because it creates redundant information. The Act graphs contain similar information implicitly. Any such replication often leads to bugs caused by inconsistencies in long-term maintenance. Ideally, the priority lists would be edited and maintained within the same framework as the Acts are edited and maintained, so that consistency could be checked automatically.

The fact that PRS-CL and its associated tool set emphasize the construction of very complex plan elements in the form of Acts, but provide relatively little support for the construction of meta-rules or the manipulation of plans as hierarchies, would seem to reflect an expectation that switching attention during plans is an unusual exception. Normal behavior is based on the execution of the elaborate Act plans. This puts PRS-CL near the opposite end of the reactive planning spectrum from architectures such as Subsumption (SA). As I described in the beginning of this chapter, SA assumes that unpredictability in action scheduling is the norm, and predictably sequenced actions are the exception. The BRP reflects a moderation between these two extremes. The BRP expects and handles the unexpected, but provides for the specification of solutions that require multiple, ordered steps.

```
(defun BRP (list-of-ACTs)
  (let* ((comp-list (consult-db '(prs::speaker-competence prs::x.1)))
         (current-BRP (BRP-name (first comp-list)))
         (current-priorities (priorities-from-name current-BRP)))

    ; loop over priorities in order, terminate on first one available
    ; to fire (as indicated by presence in list-of-ACTs)
    (do ((priorities current-priorities (rest priorities))
         (result))
        ; this is the 'until' condition in a lisp 'do' loop ---
        ; if it is true, the 'do' returns a list containing ''result''
        ((setf result (BRP-find-ACT (first priorities) list-of-ACTs))
         (list result))
      ; if we have no priorities, we return something random
      (unless (and priorities list-of-ACTs)
        (return (set-randomly list-of-ACTs))))
                      ))
    )) ; defun BRP
```

Figure 5-1: BRP prioritization implemented as a function for PRS-CL meta-reasoning. Since relative priority is situation dependent, the BRP function must query the database to determine the current competence context. Priorities are maintained as a list of Act names, each associated with a BRP name.

5.5.3 JAM / UM-PRS

I was not entirely happy with PRS-CL, so I began exploring other architectures for the same dialogue project. JAM is a Java based extension of UM-PRS, which is in turn a C++ version of PRS that is more recently developed than PRS-CL. The control cycle in all three languages is similar. JAM and UM-PRS have somewhat simplified their analog of the Act so that it no longer allows cycles, but it is still more powerful than POSH action patterns. The JAM Act analog is called simply a "plan"; for clarity, I will refer to these as JAM-plans.

JAM-plans do have a notion of priority built in, which is then used by the default meta-reasoner to select between the JAM-plans that have been activated on any particular cycle. My current implementation of BRPs in JAM is consequently a simplified version of the BRP in PRS-CL. A JAM BRP also consists primarily of a set of JAM-plans which respond to an "achieve" goal with the name of the BRP. However, in JAM, the priority of a step within the BRP is specified by hand-coding priority values into the JAM-plans. This is simpler and neater than the PRS-CL solution described above (and works more reliably). On the other hand, losing the list structure results in the loss of a single edit point for all of the priorities of a particular competence. This again creates exposure to potential software bugs if a competence needs to be rescaled and some element's priority is accidently omitted.

Both PRS implementations lack the elegance of the Ymir and BOD solutions in that Acts or JAM-plans contain both local intelligence in their plan contents, and information about their parent's intelligence, in the priority and goal activation. In POSH plans, all local information can be reused in a number of different BRPs, potentially with different relative priorities. The Ymir BRP implementation also allows for this, because the BRP (and sequence) information is present in wrapper objects, rather than in the plans themselves. I have not yet added this extra level of complexity in either PRS-CL or JAM, but such an improvement should be possible in principle. However, I did not find the advantages of working within these architectures sufficient to compensate for the difficulties, so I returned to maintaining my own action-selection system.

5.5.4 Soar

I have not actually implemented a BRP in Soar yet, but for completeness with relation to the previous section, I include a short description of the expected mechanism. Much as in PRS, I would expect each currently operable member element of the BRP to trigger in response to their mutual goal. This could be achieved either by preconditions, or exploiting the problem space mechanism. In Soar, if more than one procedure triggers, this results in an impasse which can be solved via meta-level reasoning. I assume it would be relatively simple to add a meta-level reasoning system that could recognize the highest priority element operable, since Soar is intended to be easily extendible to adapt various reasoning systems. This should operate correctly with or without chunking. This should also avoid the problem I had with PRS of including priority information on the individual steps.

The Soar impasse mechanism is also already set for monitoring lack of progress in plans, a useful feature in BRPs mentioned in Section 4.3.2. In POSH competences, retries are limited by setting habituation limits on the number of times a particular plan step will fire during a single episode (see Section 4.4.2). Ymir also supplies its own monitoring system; I have not yet addressed this problem in PRS-CL or JAM implementations.

5.6 Architecture, Methodology, or Tool?

An agent architecture has been defined as a methodology by which an agent can be constructed [Wooldridge and Jennings, 1995]. However, for the purpose of this discussion, I will narrow this definition to be closer to what seems to be the more common usage of the term. For this discussion, an *architecture* is a piece of software that allows the specification of an agent in an executable format. This actually moves the definition of architecture closer to the original definition of agent language, as a collection of "the right primitives for programming an intelligent agent" [Wooldridge and Jennings, 1994]. A *methodology* is a set of practices which is appropriate for constructing an agent. A *tool* is a GUI or other software device which creates code suitable for an architecture (as defined above), but code which may still be edited. In other words, the output of an architecture is an agent, while the output of a tool is code for an agent. A methodology has no output, but governs the use of architectures and tools.

This chapter emphasizes the use of idioms to communicate new concepts throughout the community regardless of architecture. In natural language, an idiom can be recognized as a phrase whose meaning cannot be deduced from the meanings of the individual words. If an idiom is built directly into an architecture, as a feature, there may be an analogous loss. Some features may be impossible to express in the same architecture, such as the BRP and fully autonomous behavior modules. Features implemented directly as part of an architecture reduce its flexibility. However, if a feature is implemented as an idiom, that can be overridden by direct access to the underlying code, then the problem of conflicting idioms can be dealt with at a project management level, rather than through architectural revision.

Accessibility to different idioms may explain why some architectures, such as SA or ANA, despite wide interest, have not established communities of industrial users, while others, such as Soar and PRS, have. Soar and PRS are sufficiently general to allow for the expression of a number of methodologies. However, as I said earlier, generality is not necessarily the most desirable characteristic of an agent development approach. If it were, the dominant agent "architectures" would be lisp and C. Bias towards development practices that have proven useful accelerates the development process.

I believe GUI toolkits are therefore one of the more useful ways to communicate information. They are essentially encoded methodologies: their output can be generalized to a variety of architectures (see further [DeLoach and Wood, 2001]). A toolkit might actually be an assemblage of tools chosen by a project manager. Each tool might be seen as supporting a particular idiom or related set of idioms. A GUI tool that would support the BRP would need to be able to parse files listing primitive functions, and existing sequential plans and BRPs. A new BRP could then be created by assembling these items into a prioritized list with preconditions. This assemblage can then be named, encoded and stored as a new BRP. Such a tool might also facilitate the editing of new primitive elements and preconditions in the native architecture.

Of course, not all idioms will necessarily support or require GUI interfaces. Ymir's action scheduler, discussed in Section 5.5.1, is a structure that might easily be a useful idiom in any number of reactive architectures if they are employed in handling a large numbers of degrees of freedom. In this case, the "tool" is likely to be a stand-alone module that serves as an API to the agent's body. Its function would be to simplify control by smoothing the output of the system, much as the cerebellum intercedes between the mammalian forebrain and the signals sent to the muscular system.

What then belongs in an architecture? I believe architectures should only contain structures of extremely general utility. Program structures which might be best expressed as architectural attributes are those where professional coding of an attribute assists in the efficiency of the produced agents. This follows the discussion of agent languages given in [Meyer, 1999]. Examples of such general structures are the interpreter cycle in PRS or the production system and RETE algorithm in Soar. Other structures, such as the BRP, should be implemented via idioms, and tools developed to facilitate the correct generation of those idioms.

Again, I do not discourage the development of novel architectures. An architecture may be a useful level of abstraction for developing specialized ideas and applications. However, when distributing these inventions and discoveries to the wider community, tools and idioms may be a more useful device. Note that a specialist in the use of a particular tool could be employed on a number of projects in different languages or architectures with no learning overhead, provided the tool's underlying idioms have already been expressed in those languages or architectures.

5.7 Conclusions and Discussion

In this chapter I have argued that methodology is the main currency of agent design. Novel architectures are useful platforms for developing methodology, but they are not very useful for communicating those advances to the community at large. Instead, the features of the architecture should be distilled through a process of reduction to more standard architectures. This allows for the discovery of both extensions and idioms. Idioms are particularly useful, because they allow for methodological advances to be absorbed into established communities of developers. Given that this is the aim, I consider the development of tools for efficiently composing these idioms to often be a better use of time than attempting to bring an architecture to production quality.

As an ancillary point, the discussion of reactivity in Section 5.5.2 above demonstrates that this process of reduction is a good way to analyze and describe differences in architectures. This process is analogous to the process of "embedding" described in [Hindriks et al., 1999] (see also [Hindriks et al., 2001]), and the comparisons done in Chapter 3. The reductions in that chapter were not particularly rigorous. Doing such work with the precision of [Hindriks et al., 1999] might be very illuminating, particularly if the reductions were fully implemented and tested. A particularly valuable unification might be one between a BDI architecture such as UM-PRS or JAM and Soar, since these are two large communities of agent researchers with little overlapping work.

The agent community's search for agent methodology is analogous to (though hopefully more directed than) evolution's search for the genome. When we find a strategy set which is sufficiently powerful, we can expect an explosion in the complexity and utility of our agents. While we are searching, we need both a large variety of novel innovations, and powerful methods of recombination of the solutions we have already found. This chapter has focussed on the means of recombination. I presented a definition of idiom, explained the process I used to determine that the BRP is an important one, and then described my experiences in implementing the BRP in three architectures. I also discussed the issue of publication — if one's top priority is communicating one's advances in engineering, then I recommend distributing:

- Descriptions of that advance in at least one (preferably several) different architectures, languages or paradigms, and
- Toolkits which can be adapted to a variety of languages and architectures, rather than a particular architecture.

Chapter 6

Modularity and Specialized Learning

6.1 Introduction

Chapter 4 focussed on the *organization* of behavior — what actions should occur in what order, what motivations should be met in what circumstances. I have argued that the best means to address these issues is through hand-crafted reactive plans. However, whether plans are created by human design, by automated planning, by social learning or by evolutionary search, their construction still battles problems of combinatorics and complexity. Regardless of how good a representation is chosen, and how good a method of construction, the plausibility of generating useful reactive plans depends on the size of the space that must be searched for an appropriate plan.

There are two ways to limit this search space. One is to limit the complexity of the full task specification. If we are building a nut-cracker, we could require it to understand natural language so it could be instructed to find, open and deliver a nut, or we could just use a good-sized rock. The power of simple approaches and simple agents has been thoroughly explored in the reactive-planning literature [e.g. Brooks and Flynn, 1989]. It will not be discussed at length in this dissertation, although I believe it is a critically useful insight, and one supported by the BOD process.

The other way to limit the required complexity of the reactive plan is to increase the power of the plan's primitives. This is essentially delegation, and of course comes with a trade off. In order to make it as easy as possible to create a particular agent, we must strike a balance between the complexity of building the reactive plans, and the complexity of building the plan primitives. Behavior-oriented design helps to do this in three ways:

1. by making it as easy as possible to design powerful plans,
2. by making it as easy as possible to design powerful primitives, and
3. by co-developing the plans and primitives simultaneously.

Co-developing the plans and primitives encourages the designer to make the necessary tradeoffs as they become apparent. This is the topic of Chapter 8.

The next two chapters focus on designing powerful primitives as simply as possible. BOD does this by exploiting principles of modular software engineering. This chapter explains the roles of modularity and specialized learning in a BOD agent. I propose a decomposition of the possible roles of adaptation in intelligent control, and illustrate these with a toy example. The following chapter, Chapter 7, goes into more detail on each type of representation, and presents two working systems as examples. The first is a blocks-world assembly simulation, the second a map-building autonomous mobile robot.

6.2 Behaviors

The fundamental capabilities of the BOD agents are constructed as a *behavior library*. *Behaviors* are software modules expressing a particular capacity of an agent in terms of:

- the actions needed to execute the capacity,
- perceptual information needed to inform these actions, and
- the variable state required to support either perception or action.

Behaviors are normally encoded as objects in an object-oriented language such as Java, C++, Python or the Common Lisp Object System (CLOS). However, they can also be independent processes or software packages, so long as a simple interface is built to the reactive planning system.

There are two types of interface between action selection and behaviors:

- *action primitives*, which reference actions the behaviors are able to express, and
- *sense primitives*, which reference perceptual state the behaviors maintain.

Perception, which is built into behaviors, has two major functions in a BOD agent. It is used directly by the behavior to determine **how** an action is expressed, and indirectly through action selection to determine **when** an aspect of behavior should be expressed. Perception can also trigger action directly with a behavior, provided this cannot interfere with the workings of any other behavior (see Figure 2-1, page 35).

POSH action selection will wait until a primitive returns. This allows a primitive's duration to reflect the duration of an act, which is a necessary feature for action patterns (see Section 4.3.1). However, the duration of any pause in the action selection sets the limit for the granularity of the reactiveness of the entire agent. Consequently, for an act of any significant duration, it is better to use an action primitive that simply prompts the possibly already engaged activity of the behavior. This can be complemented with the use of sense primitives to allow the POSH system to determine when the action has finished, or should be finished.

Examples of both approaches can be seen in the mobile robot controller shown in Section 7.6.2. When the robot actually hits an obstacle, it engages in an extended action pattern to disengage itself which may take over a minute. Higher level drives are only checked between items of the action pattern. On the other hand, during normal navigation, the action primitive ‘move’ merely adjusts or confirms target velocities. The forward-motion behavior is expressed continuously until progress is blocked, but action selection operates normally throughout this behavior expression.

6.3 Learning

This chapter talks about the role of specialized learning in simplifying the specification of an agent. It is important to understand that from an engineering perspective, the distinctions between *learning* and other forms of adaptation are somewhat arbitrary. Consequently, I mean to use the term ‘learning’ in a broader sense than is conventional. Learning here denotes any meaningful, persistent change in computational state. By “meaningful” I mean affecting expressed behavior. By “persistent” I mean lasting longer than a transition between control states. Learning in this sense encompasses more than just “lifetime learning” — semantic or category acquisition that persists for the lifetime of the agent. It also includes more transient knowledge acquisition, such as short-term memory for perceptual processes.

Time Scale of Human Action				**Face-to-Face Interaction**
Scale (sec)	Time Units	System	World (theory)	Levels
10^{7}	months			
10^{6}	weeks		**Social Band**	
10^{5}	days			
10^{4}	hours	Task		$\Uparrow$
10^{3}	10 Minutes	Task	**Rational Band**	$\|$
10^{2}	minutes	Task		$\|$
10^{1}	10 sec	Unit Task		$\Uparrow$ $\Downarrow$ Conversation
10^{0}	1 sec	Operations	**Cognitive Band**	$\Downarrow$ Turn
10^{-1}	100 ms	Deliberate Act		$\Updownarrow$ Back Channel
10^{-2}	10 ms	Neural Circuit		
10^{-3}	1 ms	Neuron	**Biological Band**	
10^{-4}	100 μs	Organelle		

Figure 6-1: The different time courses for different behaviors involved in a dialogue. Conversations can take from $\sim$ 10 seconds to hours, turns take only $\sim$ 1–30 seconds, and back-channel feedback (often subliminal) happens in $\sim$ 100–300 msec. After Thórisson [1999], after categories in Newell [1990].

Figure 6-1, for example, shows the large number of different time-scales on which events happen, and for which memory is required in order to respond appropriately. This figure shows the different time scales for the constituent behaviors of human conversation, including gestures. Each processing level requires its own information, but there is no reason for memory of this information to persist significantly longer than the process that attends to it. There *are* reasons of capacity for limiting the duration of these memories. Indeed, the nature of language processes almost certainly reflects evolutionary optimizations in the face of capacity limits [Kirby, 1999].

This section motivates the use in BOD of highly specialized representations and a wide variety of time durations. I intend to make it clear that such learning is a significant part of both natural and artificial intelligence. The following sections will discuss the use of state and learning in BOD. The relationship between BOD and other models of natural and artificial intelligence is discussed at greater length in Chapters 11 and 3 respectively.

6.3.1 Learning in Animals

Animals are our primary working example of what we consider intelligence to be [Brooks, 1991a, McGonigle, 1991]. Autonomous-agent research more than most branches of artificial intelligence has always acknowledged the extent to which it exploits the solutions of natural intelligence [see for example Meyer et al., 2000, Dautenhahn and Nehaniv, 1999].

Earlier this century, behaviorists (psychologists and animal researchers who concentrated on laboratory experiments), proposed that animals learn only through a general process of being able to create associations.

> The [behaviorists' general process assumption] position is that all learning is based on the capacity to form associations; there are general laws of learning that apply equally to all domains of stimuli, responses, and reinforcers; the more frequent the pairings between the elements to be associated, the stronger the associative strength; the more proximate the members of an association pair, the more likely the learning.
>
> [Gallistel et al., 1991]

Learning by association, also called 'conditioning', does appear to be a general learning mechanism with parameters that hold across species, presumably indicating a common underlying mechanism. However, behaviorist research itself eventually demonstrated that animals *cannot* learn to associate any arbitrary stimulus with any arbitrary response. Pigeons can learn to peck for food, but cannot learn to peck to avoid a shock. Conversely, they can learn to flap their wings to avoid a shock, but not for food [Hineline and Rachlin, 1969]. In related experiments, rats presented with "bad" water learned different cues for its badness depending on the consequences of drinking it. If drinking lead to shocks, they learned visual or auditory cues and if drinking lead to poisoning they learned taste or smell cues [Garcia and Koelling, 1966].

These examples demonstrate highly specific, constrained and ecologically relevant learning mechanisms. For example, the content of the associations rats are able to make biases their learning towards information likely to be relevant: poison is often indicated by smell or taste, while acute pain is often the consequence of something that can be seen or heard. Such results were originally interpreted as constraints placed on general learning to avoid dangerous associations, but research has since indicated the inverse. Specialized systems exist to form important associations [Roper, 1983]. For example, poison avoidance in the rats is handled by a specific one-shot-learning mechanism in the olfactory section of their amygdala.

The current ethological hypothesis is that learning by an individual organism serves as a last resort for evolution [Roper, 1983, Gallistel et al., 1991]. It is introduced only when behavior cannot be fully predetermined, because the competence involved requires flexibility on a less than evolutionary time scale. Further, such learning is not necessarily associative. Barn owl chicks learn to calibrate acoustically the precise location of prey (necessary because it is dependent on the shape of the individual bird's head) and bees learn the ephemeris of the sun for navigation (dependent on season and latitude). Rats learn navigation through unfamiliar environments regardless of the presence of explicit reward [Blodgett, 1929, Tolman and Honzik, 1930, Tolman, 1948, Adams, 1984]. These examples suggest that animals may be born with limited units of variable state which are instantiated during development by observing the world.

Variable instantiation can also take the form of perceptual learning or categorization. Vervet monkeys have three distinctive warning cries for predators which require different defensive action. These cries are dedicated to pythons, martial eagles, and leopards. Baby vervets make cries from a very early age, but across more general objects. For example, they may give the 'eagle' cry for anything in the sky, the 'leopard' cry for any animal, the 'python' cry for a stick on the ground. They are born attending to the sorts of stimuli they need to be aware of, but learn fine discrimination experientially [Seyfarth et al., 1980].

It should be noted that animal learning is not quite as clean as this short and admittedly biased review implies. For example, evolutionary-modeling research on the Baldwin Effect suggests that there is little selective pressure to genetically hard-code things that are consistently and universally learned by individuals of a species [Hinton and Nowlan, 1987]. Further, although animal learning is specialized, individual elements are not necessarily constrained to a single purpose or behavior. A single adaptive solution or mechanism may be leveraged by multiple processes once established. Nevertheless, the dominance of specialized learning theory in ethology is sufficient to have elicited the following description from de Waal [1996] for a popular science audience:

> The mind does not start out as a tabula rasa, but rather as a checklist with spaces allotted to particular types of incoming information.
>
> [de Waal, 1996, p.35]

In a BOD system, these "spaces" are the variable state at the heart of the behaviors.

6.3.2 Reactive and Behavior-Based Modeling of Intelligence

The most convincingly animal-like artificial agents have typically been produced under the reactive and behavior-based approaches to artificial intelligence [e.g. Blumberg, 1996, Tu, 1999, Sengers, 1999]. However, none of these have systematically supported the exploitation of multiple, interacting forms of specialized learning as the previous section suggests animals have. This section reviews the history of reactive and behavior-based approaches to in order to explain why.

The reactive approach to AI requires that an agent respond more or less directly to each situation, without the intervention of detailed deliberation or planning between sensation and action. Although this approach runs counter to many intuitive notions of rationality and intelligence, it has proved effective for problems ranging from navigation-based robot tasks [Connell, 1990, Horswill, 1993] to playing video games [Agre and Chapman, 1987] to modeling human perception and problem solving [Ballard et al., 1997].

In behavior-based AI many small, relatively simple elements of intelligence act in parallel, each handling its own area of expertise [Brooks, 1991b, Matarić, 1997]. In theory, these simpler elements are both easier to design and more likely to have evolved. The apparent complexity of intelligent behavior arises from two sources: the interaction between multiple units running in parallel, and the complexity of the environment the units are reacting to.

The central design problem for behavior-based systems is thus behavior arbitration: determining which parallel module controls physical behavior at any one time. This is a problem not only for design but also during development. It can be very difficult to determine if faulty behavior is the result of one or more behaviors operating simultaneously.

The central problem of reactive planning is somewhat different. In what is sometimes called *the external Markov assumption*, fully reactive planning expects that the next action can be entirely determined by external state. Unfortunately, this is false (recall the monkey holding the banana in Section 2.3.2). Most intelligent agents will often find themselves experiencing identical environments as the result of different original initiatives that should require the agent to select different behaviors. For example, the offices in a laboratory where a robot operates may be connected by a single hallway. The hallway is essentially the same environment whichever office the robot needs to enter next. Further, even if situations are different, they may appear the same to the limited perception of the robot, a problem sometimes referred to as *perceptual aliasing*.

Behavior-based systems are not necessarily reactive. Minsky [1985] intended planning and complex representation to be parts of the elements of his “society of agencies”. Reactive systems, on the other hand, are likely to be at least partly behavior-based. Decomposing intelligence into large units decreases the number of actions to be selected between. This makes reacting easier.

In practice, behavior-based systems do tend to be reactive. This is because the first widely known example of behavior-based AI was also strongly reactive and confounded the two paradigms. This is the Subsumption Architecture [Brooks, 1989]. Although Brooks himself championed specialized learning early [Brooks, 1991b, pp. 157–158], in the same paper he states:

> We do claim however, that there need be no explicit representation of either the world or the intentions of the system to generate intelligent behaviors for a Creature... Even at a local level we do not have traditional AI representations. We never use tokens which have any semantics that can be attached to them... There are no variables that need instantiation in reasoning processes. There are no rules which need to be selected through pattern matching. There

are no choices to be made. To a large extent, the world determines the action of the Creature.

[Brooks, 1991b, p. 149]

"The world is its own best model" became a mantra of behavior-based AI. This is a view that I largely agree with, but not to the extreme of this quotation, and even less so the extreme to which it was interpreted.

Another problem with behavior-based AI was that, in their struggle to minimize state, the early architects of reactive systems [e.g. Brooks, 1991b, Maes, 1989] complicated both learning and control unnecessarily by confounding the flow of information with the flow of control. BOD divides the issues of control, or **when** a behavior is expressed, from perception and action, or **how** it is expressed. One of the most significant contributions of this dissertation is the integration of behavior-based control and systematic support for developing specialized learning and representations. This integration is deceptively obvious from the perspective of object-oriented design, but it is a significant advance in the state-of-the-art for behavior-based architectures.

6.4 State and Learning in BOD

An autonomous agent in a dynamic environment should be as reactive as possible. Learning is applied when programming control is otherwise prohibitively complex. For example, if the triggers that discriminate the set of elements of a competence become too convoluted, either because of multiple dependencies or multiple steps to find those dependencies, it will be easier to reduce the triggers to one or two statements based on new perceptual state.

In general, learning should be used as little as possible. In particular, an obvious indication that it has been overused is if the agent cannot learn required perceptual information reliably. The BOD thesis that intelligent agents can best be developed through the use of specialized learning modules is really a restatement of the well-established result that learning is dependent on bias [Kaelbling, 1997]. We attempt to maximize bias by minimizing learning and constraining each necessary adaptive element individually. When learning cannot be performed reliably, a further constraint is needed. Notice that the additional information for that constraint may take the form of more perceptual state rather than more control.

The introduction of new perceptual state also implies that some new behavior will be needed to keep the state continuously updated, or at least make it appear to be so, so that some value is always available on demand. The BOD approach emphasizes the separate flow of control from the flow of data, but the continuous flow of data is not eliminated. Remember again Figure 2-1 (page 35) which shows the behaviors as autonomous units, with action selection as a filter on their expressed acts. The reactive hierarchical control architecture is developed in conjunction with the behaviors it triggers. New perceptual requirements driven by the action selection can in turn motivate new state and learning, which requires new behavior modules.

6.4.1 Types of State

Learning and knowledge are expressed in different types of state which we can divide into four categories:

Control State State in the form of the currently active behavior or behaviors, and the paths that activation can follow. Control state is analogous to a program counter — it records the immediate past in terms of action selection and determines, in combination with sensory information, the agent's next act. Conventional AI systems hold control state in a plan, in which a program pointer determines what the next step should be. Some plan-less behavior-based architectures [e.g. Brooks, 1991b] hold similar state in finite state machines (FSMs), though here the information is distributed amongst behaviors.

Deictic Variables Simple, pointer-like variables which refer to a particular object to which the agent is currently attending. Deictic variables allow a system to generalize over cases where particular plans embodied in control state may operate. An example from ethology is the classic work on imprinting by Lorenz [1973]: a gosling will follow and learn from whatever mother-shaped thing it sees (possibly a baby-buggy) during a critical period after hatching. Upon reaching adulthood, a male goose will try to mate with similar objects. There is considerable evidence that humans use similar strategies [Ballard et al., 1997, Horowitz and Wolfe, 1998]. Rhodes [1995] has shown how deictic representation significantly extends purely reactive architectures.

Perceptual and Motor Memory Specialized systems and representations where information can accumulate. Perceptual memory may last for only a fraction of a second, or for the lifetime of the agent. Perception requires memory because very little information is available in each snapshot of sensory information. Motor control may require memory if fine-tuning cannot be specified in advance.

Meta-State State about other internal state, or learning to learn. Although the distinction between learning and learning to learn may seem obscure, it is supported by biological research. For example, Bannerman et al. [1995] demonstrates that maze-learning ability is intact in mice that have lost hippocampal learning, provided they have prior experience in learning mazes. Many researchers have suggested such ability is critical to developing human-like intelligence [e.g. Elman et al., 1996].

6.4.2 Choosing Representations in BOD

This decomposition of state is roughly analogous to the decomposition of control state presented in Section 2.3. In this case, favoring simplicity is favoring a more reactive representation. Intelligence is more reactive when it relies on simpler representations.

- Control state is the simplest element of this typology. It is not necessarily reactive, but reactive systems all have some element of control state, made reactive by using constant sensory monitoring to determine the branches between state. Control state does not represent the world, except indirectly. It must deal with its environment. Consequently, information about the environment can be deduced from the control structure.

- Deictic variables are set to represent or at least reference some particular object in or aspect of the environment. As such, they restrict attention and make control more manageable. They should be used when control structures would otherwise be replicated. Replication of control state is cumbersome and unreliable in terms of an evolving system, since multiple instantiations may need to have the same updating. It also consumes program space and programmer time.

- Excessive control state may also indicate the need for perceptual memory. Determining the appropriate context for a behavior must not require too many control steps, or the system will become complicated and slow. This situation indicates that the sensory information for determining the context should be consolidated into a representation. Such representations can be updated by processes parallel to the control structure, while the interface to the control system can be reduced to a single "sensory" check. Determining the need for adaptive state for motor control is similar. Perceptual state serves as the basis of behavior decomposition in BOD (see Section 7.6.4).

- State needed for motor control may also be stored in behaviors. Although none of the experiments described in this dissertation have used the strategy, there are many examples in the AI literature of learned control that could be of use to a BOD agent. Examples include the learned passing and kicking behaviors of Stone and Veloso [1999]'s soccer-playing agents, Schaal and Atkeson [1994]'s learned arm control, and the models learned for video agents by Baumberg and Hogg [1996] and Brand et al. [1997]. Other related work includes the fuzzy behaviors used in the robot of Konolige and Myers [1998], and the vector maps used byArkin [1998]. We have already discussed including such often-self-contained algorithms into a BOD agent (see Section 2.2). In Chapter 12 I describe integrating BOD into an architecture, Ymir [Thórisson, 1999], with a special behavior for micro-managing motor control.

- Meta-state is necessary for an agent to learn from or redescribe its own experience or alter its own processes. Karmiloff-Smith [1992] proposes that redescription of knowledge is central to human development. For example, the ability to learn a skill as a unit by rote then decompose its components for reuse in a different strategy. Norman and Shallice [1986] argue that deliberation is a way to bring increased attention to bear on routine processes, thus increasing their precision or reliability.

BOD does not really support the use of meta-state, though its might be modeled essentially as perceptual state. The main purpose of BOD is to replace the long, frequently arcane process of development by an organism with a long, but hopefully orderly process of development by an engineer. Perhaps because of this emphasis, I have not yet found a simple heuristic for determining when meta-learning is necessary or preferable to a new phase of human design. Possible future extensions to BOD to support meta-learning are discussed briefly in Section 7.7 and more extensively in Chapter 11.

6.5 Trading Off Control and Learning

To demonstrate the differences between representational styles, let's think about an insect robot with two 'feelers' (bump sensors), but no other way of sensing its environment.

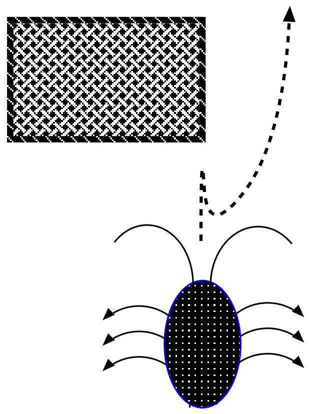


Figure 6-2: An insect-like robot with no long-range sensors (e.g. eyes) needs to use its feelers to find its way around a box.

6.5.1 Control State Only

This plan is in the notation from Chapter 2, except that words that reference other bits of control state rather than primitive actions are in bold face. Assume that the 'walk' primitives take some time (say 5 seconds) and move the insect a couple of centimeters on the diagram. Also, assume turning traces an arc rather than happening in place (this is the way most 6 legged robots work.) Below is probably the simplest program that can be written using entirely control state.

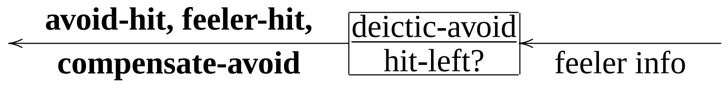
$$\textbf{walk} \Rightarrow \left.\Big\uparrow\right. \left\langle \begin{array}{r} \text{(left-feeler-hit)} \Rightarrow \textbf{avoid-obstacle-left} \\ \text{(right-feeler-hit)} \Rightarrow \textbf{avoid-obstacle-right} \\ \Rightarrow \text{walk-straight} \end{array} \right\rangle \tag{6.1}$$

$$\textbf{avoid-obstacle-left} \Rightarrow \langle \text{walk backwards} \rightarrow \text{walk right} \rightarrow \text{walk left} \rangle \tag{6.2}$$

$$\textbf{avoid-obstacle-right} \Rightarrow \langle \text{walk backwards} \rightarrow \text{walk left} \rightarrow \text{walk right} \rangle \tag{6.3}$$

6.5.2 Deictic State as Well

If we are willing to include a behavior with just one bit of variable state in it, then we can simplify the control state for the program.



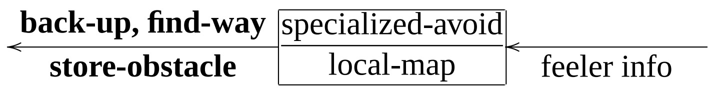
In this behavior, the bit hit-left? serves as the deictic variable the-side-I-just-hit-on. **Avoid-hit** and **compensate-avoid** turn in the appropriate direction by accessing this variable. This allows a reduction in redundancy in the plan, including the elimination of one of the action patterns.

$$\textbf{walk} \Rightarrow \left.\uparrow\right| \left\langle \begin{array}{r} \text{(feeler-hit)} \Rightarrow \textbf{avoid-obstacle} \\ \Rightarrow \text{walk-straight} \end{array} \right\rangle \tag{6.4}$$

$$\textbf{avoid-obstacle} \Rightarrow \langle \text{walk backwards} \rightarrow \text{avoid hit} \rightarrow \text{compensate avoid} \rangle \tag{6.5}$$

6.5.3 Specialized Instead of Deictic State

Instead of using a simple reference, we could also use a more complicated representation, say an allocentric representation of where the obstacle is relative to the bug that is updated automatically as the bug moves, and forgotten as the bug moves away from the location of the impact. Since this strategy requires the state to be updated continuously as the bug moves, walking must be a method (**find-way**) on this behavior.



$$\textbf{walk} \Rightarrow \left.\uparrow\right| \left\langle \begin{array}{r} \text{(feeler-hit)} \Rightarrow \textbf{react-to-bump} \\ \Rightarrow \text{find-way} \end{array} \right\rangle \tag{6.6}$$

$$\textbf{react-to-bump} \Rightarrow \langle \text{store-obstacle} \rightarrow \text{walk backwards} \rangle \tag{6.7}$$

If this is really the only navigation ability our bug has, then the vast increase in complexity of this behavior does not justify the savings in control state. On the other hand, if our bug already has some kind of allocentric representation, it might be sensible to piggyback the feeler information on top of it. For example, if the bug has a vector created by a multi-faceted eye representing approximate distance to visible obstacles, but has bumped into something hard to see (like a window), it might be parsimonious to store the bump information in the vision vector, providing that updating the information with the bug's own motion isn't too much trouble. Again, insects actually seem able to do this [e.g. Hartmann and Wehner, 1995], and some robots also come with primitive odometric 'senses' that make this easy, providing precision is not important. This is in fact the model of bump detection used in the extended real robot model described in Section 7.6.2.

6.5.4 State for Meta-Learning

How would the bug's control look if it used a representation suitable for meta-reasoning? Honestly, not sufficiently different from the control for the specialized robot to make it worth drawing. The primary difference would be that the representation wouldn't be any kind of simple allocentric map, but would rather have to be a more universal representation, such as logic predicates or a homogeneous neural representation. This would allow the same operators to act on *any* knowledge the bug happens to store.

The problem with universal representations is roughly the same as the problem with production rules (see Section 2.3.2). In order for the information to be useful, it has to be tagged with a great deal of information about the context in which it is to be used. Although this strategy has been demonstrated feasible [Newell, 1990, Georgeff and Lansky, 1987], it loses the advantages of modularity. The programmer loses the localization of related state and program code; the machine learning algorithm loses its specialized, helpfully biased representation. If the data is being stored in a neural network, then the same problems apply, only more so. Developing modularity in neural networks is still very much an open question [Wermter et al., 2001]. Essentially, both these universal representations have problems of combinatorics. BOD addresses combinatorics by using modularity and hierarchy.

6.6 Conclusions

This chapter has discussed the fundamental role of behaviors, state and learning in a BOD system. It has also shown that state can appear in a variety of forms, facilitating highly specialized learning, in a variety of degrees of persistence and kinds of representation. The next chapter further illustrates the uses of each of the four types of state identified here, with references from related literature as well as working examples I implemented under BOD.

Chapter 7

Learning by Design

7.1 Introduction

Chapter 6 introduced and explained modularity and specialized learning in behavior-oriented design. It decomposed variable state for agents into four categories: control state, deictic representations, perceptual or motor learning, and meta-reasoning. This chapter explores each of these in more depth, with extended examples for deictic and perceptual learning in particular.

7.2 Control State

One of the myths of artificial intelligence is that the early experimenters in reactive planning advocated stateless systems. In fact, the fundamental units of subsumption architecture [Brooks, 1991b] are augmented finite state machines. At each moment every behavior records a particular state of execution, and this state helps determine what that behavior will do next.

The most reactive behaviors do respond directly and continually to factors in the environment. For example, in the first few Braitenburg Vehicles [Braitenberg, 1984] motor speed directly correlates to light or heat intensity. The Polly vision-based obstacle-avoidance algorithm also determines speed and direction by the apparent distance to obstacles in each individual visual frame. [Horswill, 1993].

However, even simple behaviors may require some kind of stored internal state, and often sequential control. For example, many of the insect-like robots first built under subsumption architecture would back off and turn away from obstacles detected by bumping their antennae[1]. The environmental state to which the robot reacts is only detected while the antenna is bent, but the behavior determined by that state must be perpetuated over the period of backing up after the antenna is disengaged. A variety of solutions to this problem were described in the last section of the previous chapter (Section 6.5), using varying combinations of control, deictic, perceptual and meta state.

[1]Not the best known one though: Genghis just tried to lift its front leg over the obstacle [Angle, 1989, Brooks, 1989]. Nevertheless, the same problem holds.

In a conventional AI system, the initial collision would have resulted in a persistent internal representation of the collision point as an assumed obstacle, and the registration of this knowledge would have resulted in a planner selecting a new behavior (such as exploring the obstacle) or a new trajectory. In a fully reactive agent, the collision is registered only temporarily in control state. One behavior is forced into a different state, which results in different expressed actions for the agent as a whole. Once normal behavior is resumed, the event will be forgotten.

This disposing of information may appear wasteful, but in practice the information is difficult to record accurately and is often transient. The superiority of the reactive approach to local navigation has been shown empirically [e.g. Maes, 1990a, Bonasso et al., 1997]. A more significant criticism, however, is that this approach exchanges the complexity of adequately representing the environment for complexity in designing an agent's control. As Chapter 3 documents, AI developers have largely moved away from systems that rely purely on control state.

7.3 Deictic Representation

One of the problems of traditional representation has been the number of items that a system needs to represent if it represents a complete world model. A solution to this problem that is gaining acceptance is the use of deictic variables [see e.g. Ballard et al., 1997, Horswill, 1997]. A deictic variable is a permanent memory structure with changeable external reference that can be incorporated into plans. For example, representing a blocks world problem might conventionally require individually labeling each block. Deictic variables provide a restricted set of labels such as the-block-I'm-holding or even the-green-block-I-most-recently-saw.

Deictic variables were popularized in the work of Agre and Chapman [Agre and Chapman, 1990, Chapman, 1990] who used them to write a reactive system that successfully plays computer games. The ideas behind deictic variables can also be found in Minsky's pronemes [Minsky, 1985], in visual attention [Ullman, 1984], and apparently in the philosophical work of Heidegger [Dreyfus, 1992]. Whitehead [1992] uses deictic variables to simplify reinforcement learning. Horswill [1995] uses them in combination with a live visual image to produce a system that answers prolog-like queries ("Is there a green block on a red block?") about a scene using no other representation other than the visual signal itself.

The benefits deictic representations bring to reactive architectures are similar to what it brings to more traditional representations, though possibly more substantial. One of the common criticisms of reactive planning is that it transfers the complexity of data representation into control complexity. Thus the reduction of complexity brought about by using deictic variables for a reactive system is not in terms of amount of data, but in the size of the control architecture, the number of behaviors, or the number of reactive plans [Horswill, 1997, Rhodes, 1996].

7.4 Braniff — Examples of Deictic Representation

Deictic representation requires two things. First, it requires some state to be allocated for the variable. This does not necessarily have to be added in code. For a real robot, the reference of the variable may be what is in the hand, where the camera is centered, or what the vision system's tracking is fixated on. Thus in a real robot, the deictic variable state may be situated either in the external world (e.g. the place right in front of the robot), within the robot's mechanism, within its RAM, or it may refer to state in a peripheral system such as a visual routine processor. The second thing deictic representation requires are primitives for setting and sampling that variable.

The following examples are based on the work of Whitehead [1992]. Whitehead worked in a simulated blocks world, where he attempted to have a simulated robot arm learn to pick a green block out of a stack. Whitehead's work was at the the time of these experiments one of the leading pieces of work in one of the leading machine-learning algorithms, reinforcement learning (RL). Also, I had access to a beta version of an advanced vision system, the visual routine processor (VRP) [Horswill, 1995] that actually performed all the sensing primitives assumed by Whitehead's thesis. This system was connected to the VRP observing LEGO Duplo blocks, but unfortunately never to a functioning robot hand. The results in this section are primarily from a simple blocks-world simulator I wrote as an interim solution.

In the code that follows, the **bold face** represents the names of program structures. Everything else was a primitive action in Whitehead [1992] and was mapped to primitive actions in the VRP.

7.4.1 Grab Green

Grab-green produces the behavior produced by Whitehead's dissertation. It looks like this ($\perp$ stands for *false*):

grab-green $\Rightarrow$

$$\uparrow\left|\begin{array}{r}
\text{(action-frame-color 'green)(action-frame-in-hand)} \Rightarrow \textit{goal} \\
\left\langle\begin{array}{rl}
\text{(object-in-hand)} \Rightarrow & \textbf{lose-block} \\
\text{(attn-frame-color 'green) (frames-vertically-aligned } \perp\text{)} \Rightarrow & \text{move-action-frame-to-green} \\
\text{(attn-frame-color 'green))} \Rightarrow & \textbf{grasp-stack-top}
\end{array}\right\rangle \\
\text{(green-in-scene)} \Rightarrow \text{move-attn-frame-to-green}
\end{array}\right. \tag{7.1}$$

grasp-stack-top $\Rightarrow$

$$\langle \text{move-action-frame-to-stack-top} \rightarrow \text{grasp-object-at-action-frame} \rangle \tag{7.2}$$

lose-block $\Rightarrow$

$$\langle \text{move-action-frame-to-table} \rightarrow \text{(action-frame-tabel-below)} \rightarrow \text{place-object-at-action-frame} \rangle \tag{7.3}$$

This is obviously an elaboration of the BRP 4.1 explained in detail in Chapter 4[2].

[2]This is not the original code. My original code for this competence actually contains longer sequences

Whitehead originally used two attention markers — deictic variables for the visual system first hypothesized by Ullman [1984]. Ullman believed that the visual system had a limited number of these markers (countable by determining how many independent objects a person can track, usually about four.)

Whitehead constrained one marker to be purely for visual attention, while the other governs action by the hand. Constraining the types of primitives applied to each marker reduced the size of the search space for his RL algorithm, which was still severely tasked to order the primitives appropriately.

The markers can be moved to attractive 'pop-out' attributes, such as color, compared to each other, moved relative to each other, and so on. The markers also follow any object to which they are attending. So, for example, in the above plan, the attention frame is still on the block after it has been grasped and thus moves to the robot's hand.

Notice that this program also effectively uses the hand as a third deictic variable, though one external to the 'mind' that therefore requires the other variables to be directed at it in order to analyze it.

This program works: it succeeded in grasping a green block for any initial block configuration so long as there was a green block, and terminated if there was no green block present.

7.4.2 Green on Green

This program illustrates a simple BRP hierarchy. Early in his dissertation, Whitehead said his goal was to pile a red block on a green block, but this problem was never demonstrated or revisited. It was the logical next test for the action-selection system. However, since I had only implemented grabbing green blocks, I started with green on green.

The preconditions in this plan are often to long to fit on a single line. When they seem to guard a '+', this actually means they are continued onto the next line.

because I was not yet astute at programming with competences. In fact, I was debugging both the code and concept for a competence while I was developing this plan. There were consequently redundant checks that, for example, that the hand was not grasping anything, and were in fact separate steps for grasping the top block on the stack depending on whether or not it was green.

green-on-green $\Rightarrow$

$$\uparrow\left\langle \begin{array}{rl}
\text{(attn-frame-home)} \Rightarrow + & \\
\text{(attn-frame-shape 'block) (attn-frame-color 'green)} \Rightarrow + & \\
\text{action-trace-up-from-attn (action-frame-shape-block)} \Rightarrow + & \\
\text{(action-frame-color 'green) (frames-synched } \perp\text{)} \Rightarrow & \textit{goal} \\
\text{(object-in-hand) (attn-frame-home) (attn-frame-color 'green)} \Rightarrow & \textbf{place-green-on-green} \\
\text{(object-in-hand) (attn-frame-home) (attn-frame-table-below)} \Rightarrow & \textbf{place-first-green} \\
\text{(object-in-hand } \perp\text{) (attn-frame-home) (attn-frame-shape 'table)} \Rightarrow & \text{look-away } \textbf{grab-green} \\
\text{(object-in-hand } \perp\text{) (attn-frame-home) (attn-frame-shape 'block)} \Rightarrow + & \\
\text{(attn-frame-color 'green)} \Rightarrow & \text{look-away } \textbf{grab-green} \\
\text{(attn-frame-home } \perp\text{)} \Rightarrow & \text{move-attn-frame-home} \\
\text{(object-in-hand } \perp\text{) (attn-frame-home) (attn-frame-shape 'block)} \Rightarrow & \textbf{clear home} \\
\text{(object-in-hand)} \Rightarrow & \textbf{lose-block}
\end{array} \right\rangle \tag{7.4}$$

There is actually more information in the BRP than this notation represents. For one, each element is labeled with a name so that it is easier to see what that element does. For another, some elements actually have equivalent priority. Here is the list of the elements' names, with elements of shared priority on the same line. Notice that the elements are in the same order as they are above, this is simply another representation with different information.

green-on-green-goal
place-green-on-green, place-first-green
check-home, get-another-green, get-first-green
clear-home, lose-block

So in this plan, if a green block is on another green block, the goal has been reached. Otherwise if it is holding a green block, it places it on the first green block if one exists, or on the goal location if it does not. If the agent is not grasping a green block, it will either check the status of its home, or get a green block, if it can. If it can't because there is a wrong block on its goal location, it will pick that block up. If it can't because it's already holding a block (which can't be green at this point) then it drops the block.

This plan requires several sub elements:

grab-green $\Rightarrow$

$$\uparrow\left\langle \begin{array}{rl}
\text{(action-frame-color 'green)(action-frame-in-hand)} \Rightarrow & \textit{goal} \\
\text{(object-in-hand)} \Rightarrow & \textbf{lose-block} \\
\text{(attn-frame-color 'green) (attn-frame-home)} \Rightarrow & \text{move-action-frame-to-green} \\
\text{(attn-frame-color 'green))} \Rightarrow & \textbf{grasp-stack-top} \\
\text{(attn-frame-color 'green) (frames-vertically-aligned } \perp\text{)} \Rightarrow & \text{move-action-frame-to-green} \\
\text{(green-in-scene)} \Rightarrow & \text{move-attn-frame-to-green}
\end{array} \right\rangle \tag{7.5}$$

grab-green-goal
lose-block, veto-green
unbury-green

synch-on-green
find-green

This has one additional element to plan 7.1: if the attention frame has ‘popped’ to a block on the home pile, then it looks for another one. This is necessary because of the stochastic nature of the low-level visual routines — there is no other way to constrain their search without forcing the home tower to be in a different visual field from the source piles for the robot.

place-green-on-green $\Rightarrow$

$$\left\langle \begin{array}{r} \text{move-attn-frame-to-hand} \rightarrow \text{move-action-frame-to-attn} \rightarrow \text{move-attn-frame-home} \rightarrow \\ \text{(action-frame-color 'green)} \rightarrow \text{(action-frame-in-hand)} \rightarrow \text{move-action-frame-to-attn} \rightarrow \\ \text{action-trace-up-from-attn} \rightarrow \text{place-object-at-action-frame} \end{array} \right\rangle \tag{7.6}$$

place-first-green $\Rightarrow$

$$\left\langle \begin{array}{r} \text{move-attn-frame-to-hand} \rightarrow \text{move-action-frame-to-attn} \rightarrow \text{move-attn-frame-home} \rightarrow \\ \text{(action-frame-in-hand)} \rightarrow \text{(action-frame-color 'green)} \rightarrow \text{(attn-frame-table-below)} \rightarrow \\ \text{move-action-frame-to-attn} \rightarrow \text{place-object-at-action-frame} \end{array} \right\rangle \tag{7.7}$$

clear-home $\Rightarrow$

$$\left\langle \begin{array}{r} \text{move-action-frame-to-attn} \rightarrow \text{move-action-frame-to-stack-top} \rightarrow \\ \text{(frames-vertically-aligned)} \rightarrow \text{grasp-object-at-action-frame} \end{array} \right\rangle \tag{7.8}$$

In coding this plan several things became evident. First, managing the attention variables was very important. It was easy to accidently build plans that iterated between rules that simply shifted the visual attention around (e.g. between check-home and look-for-green). This emphasizes the utility of sequences. Visual pointers were only moved in an established context; otherwise they were left in canonical locations. Notice I added a primitive *look-away* which simply ensured the markers were not focussed on any blocks[3].

Second, stacking takes more state than the simple grab-green. Another ‘external’ deictic variable is added, *home*, the place the critical pile is being made. The state of the block stack in the home location indicates the next task for the robot, whether to pick up a green block, a red block, or whether the stack is complete. This may explain why Whitehead [1992] never returned to this task; his reinforcement learning system was already challenged by the number of free variables in the original task. Ballard et al. [1997] cite Whitehead’s difficulties as a possible explanation for the small number of ‘visual markers’ or pieces of free visual state humans seem to have evolved.

In addition to the stack location and the two visual markers, without the mechanism of a subgoal a fourth marker would have been necessary. In order to gauge the state of the stack, a marker must check its contents. If that marker detects that the stack contains a

[3]I never encountered this difficulty in any other domain, though the rest of my domains were more conventionally reactive with animal-like problems. The problem of getting ‘caught in a loop’ is also frequent in children who are trying to master complex procedures (and I have also seen it in chimpanzees). I am not certain yet what characterizes the sorts of problems that can generate loops: perhaps it is the unnatural regularity of the perceptual stimuli (the blocks) or the discrete nature of the possible acts.

green block, this can trigger the competence to grasp a green block, which uses both visual markers. If there were no separate competence, another marker would be needed to make regular comparisons to the top of the home stack. This demonstrates an interchange of control state and deictic state, again with considerable combinatorial ramifications.

7.4.3 The Copy Demo

The copy demo is a famous early experiment in robotics conducted at MIT [Horn and Winston, 1976, Chapman, 1989]. The principle is simple — if the robot sees a stack of blocks, it should try to replicate that stack. The code for this task is actually slightly simpler than for green-on-green, if one discounts the replications to account for all three colors. The extra complexity of tracing two stacks is compensated for by not having to represent the goal entirely in the control structure. Also, since the competence and simulation mechanism was by this time fully debugged, the original replication of the demo took less than three hours to write and debug.

$$
\textbf{copy-demo} \;\Rightarrow
$$

$$
\uparrow\left|\begin{array}{r l}
\text{(frames-horizontally-aligned) (attn-frame-goal)} & \Rightarrow + \\
\text{(action-frame-home) (attn-frame-color } \bot\text{) (action-frame-color } \bot\text{)} & \Rightarrow \textit{goal} \\
\text{(frames-horizontally-aligned) (frames-color-synched)} & \Rightarrow + \\
\text{(attn-frame-goal) (action-frame-home)} & \Rightarrow \textbf{check-next-goal} \\
\left\langle \begin{array}{r l}
\text{(frames-horizontally-aligned)} & \Rightarrow + \\
\text{(attn-frame-color red) (action-frame-color } \bot\text{)} & \Rightarrow \text{look-away } \textbf{grab-red} \\
\text{(frames-horizontally-aligned)} & \Rightarrow + \\
\text{(attn-frame-color blue) (action-frame-color } \bot\text{)} & \Rightarrow \text{look-away } \textbf{grab-blue} \\
\text{(frames-horizontally-aligned)} & \Rightarrow + \\
\text{(attn-frame-color green) (action-frame-color } \bot\text{)} & \Rightarrow \text{look-away } \textbf{grab-green}
\end{array} \right\rangle & \\
\text{(object-in-hand)} & \Rightarrow \textbf{lose-block} \\
\text{(object-in-hand } \bot\text{) (attn-frame-home) (attn-frame-shape 'block)} & \Rightarrow \textbf{clear home} \\
\bot & \Rightarrow \textbf{start-over}
\end{array}\right. \tag{7.9}
$$

The priorities again:

copy-demo-goal
check-next-goal
add-blue, add-red, add-green, place-block
lose-block
clean-home
start-over

If the the stack has not been replicated, then check the next goal, or if that has been done, get the appropriately colored block, unless you are holding one already, in which case drop it, or unless you need to clear the goal stack (if you've previously made an error), or in the worst case, reset your visual pointers to a reasonable place to start another action.

This all requires a couple of sequences:

check-next-goal $\Rightarrow$

$$\left\langle \begin{array}{r} \text{move-action-frame-to-attn} \rightarrow \text{action-trace-up-from-attn} \rightarrow \\ \text{move-attn-frame-to-action} \rightarrow \text{move-action-frame-home} \rightarrow \textbf{synch-up}\rangle \end{array} \right\rangle \tag{7.10}$$

start-over $\Rightarrow$

$$\left\langle \text{ move-attn-frame-goal} \rightarrow \text{move-action-frame-home}\rangle \right\rangle \tag{7.11}$$

The copy-demo task required yet another special place: the goal stack. It also requires grab-*color* and lose-block as above (7.5 and 7.3), and also a small new competence for horizontally synchronizing the visual markers.

sync-action-up $\Rightarrow$

$$\left\uparrow \begin{array}{l} \begin{array}{r} \text{(frames-horizontally-aligned)} \Rightarrow + \end{array} \\ \left\langle \begin{array}{r} \text{(attn-frame-goal) (action-frame-home)} \Rightarrow \textit{goal} \\ \text{(action-frame-color } \perp\text{) (attn-frame-goal)} \Rightarrow \text{move-action-frame-home} \\ \text{(frames-horizontally-aligned } \perp\text{)} \Rightarrow + \end{array} \right\rangle \\ \text{(action-frame-home) (attn-frame-goal)} \Rightarrow \text{move-action-frame-up} \end{array} \right. \tag{7.12}$$

Again, much of **sync-action-up** is redundant, in response to some of the bugs in the beta VRP. Its main purpose is simply to move the action frame to the level of the attention frame.

The replications for color are a consequence of my staying as faithful as possible to the original constraints of Whitehead's project so that additional state requirements would be clear. They could be replaced with a single extra deictic variable, the-color-I'm-attending-to. This example consequently shows clearly the way internal, external, and control state can trade off or be used to compensate for one another.

These plans also have the longest action patterns or sequences of any I've since built. This is because the constraints of the project made building more natural operators impossible, and because the nature of the constraints made many situations extremely perceptually ambiguous. Under BOD, one would normally make the behaviors supporting the primitives much more powerful to simplify the control. These programs thus illustrate clearly perceptual ambiguity (e.g. the block color) being compensated for by control state.

7.5 Perceptual State

Perception is a psychological abstraction for the level at which processed sensory information is used for cognition. The abstraction is clearest in humans: what we perceive is what we remember and report. This is famously open to illusion and distortion. What we sense must be present in environment, e.g. light and sound. Perception is driven by expectation and therefore dependent both on learning and on context [Carlson, 2000]. We can learn to discriminate differences without having any explicit knowledge of what those differences are [Bechara et al., 1995a]. In some cases, this learning requires no feedback, simply exposure to the appropriate stimulus [Sundareswaran and Vaina, 1994].

Perception makes a similar contribution to reducing plan complexity to that of deictic representation. Here, the reduction is not only by collapsing similar plans into one, but by substantially increasing the power and information content of plan elements. Arguably, the main contribution of reactive, behavior-based AI has been a change in the basis for action selection in an intelligent agent. Plans relying on complex internal models of the world have been replaced by distributed actions relying directly on fast, cleverly specialized perceptual routines.

Perceptual state definitionally reduces reactivity, since it relies on "old" information. However, it plays the same important role as structured control state in that it preserves information already experienced to disambiguate control decisions.

7.6 Edmund — Examples of Perception and Specialized Learning in a Robot

The problems of perception are more intuitively obvious when the agent is working with physical sensors, like we animals do. The following examples are taken from a BOD system I used to control a mobile robot, shown in figure 7-1. This work was conducted in The Laboratory for Cognitive Neuroscience and Intelligent Systems of the University of Edinburgh. The first example deals with determining from multiple, inaccurate sensors how the robot can best proceed forward. It was conducted in 1996–7 and has previously been reported [Bryson and McGonigle, 1998, Bryson, 2000b]. The second deals with combining multiple strategies, several using learning, for navigating around a complex space. This work was conducted in 1997–8, and has not been previously reported.

7.6.1 The Robot

The robot I used is shown in Figure 7-1 is a commercially available Nomad 2000. The robot has 3 different drive velocities to control: its speed, its direction, and the rotation of its sensing turret. The sensing turret has 16 faces, each with a sonar located at the top (about a meter off the ground) and an infrared sensor near the bottom (about a quarter of a meter above the ground). Below the turret is the wheel base which is surrounded by 20 bump sensors, staggered across two rings of ten sensors each.

The infrared sensors are useful from 2cm to 30cm away from the robot; the sonar sensor are useful for distances greater than 20cm. However, both sorts of sensors have difficulties. Infrared works by bouncing light, and thus gives much stronger readings (which look nearer) for light and reflective objects. Sonars work by bouncing sound, and may be fooled by a number of factors, including refraction away from the robot prior to reflection to the sensor, or reflection into a different sensor than emitted the initial signal. Sonar readings therefore can be very erratic, while infrared readings can be systematically distorted. Further, neither sensor necessarily provides coverage for the entire height of its facet of the robot.

Bump sensors are relatively accurate, but require hitting an obstacle hard enough to register through a fairly strong rubber bumper. By this point the robot is usually thoroughly engaged with the obstacle (its bumper is effectively gripping it), and needs to move directly opposite from the impact to extract itself. The robot also has odometric sensors which

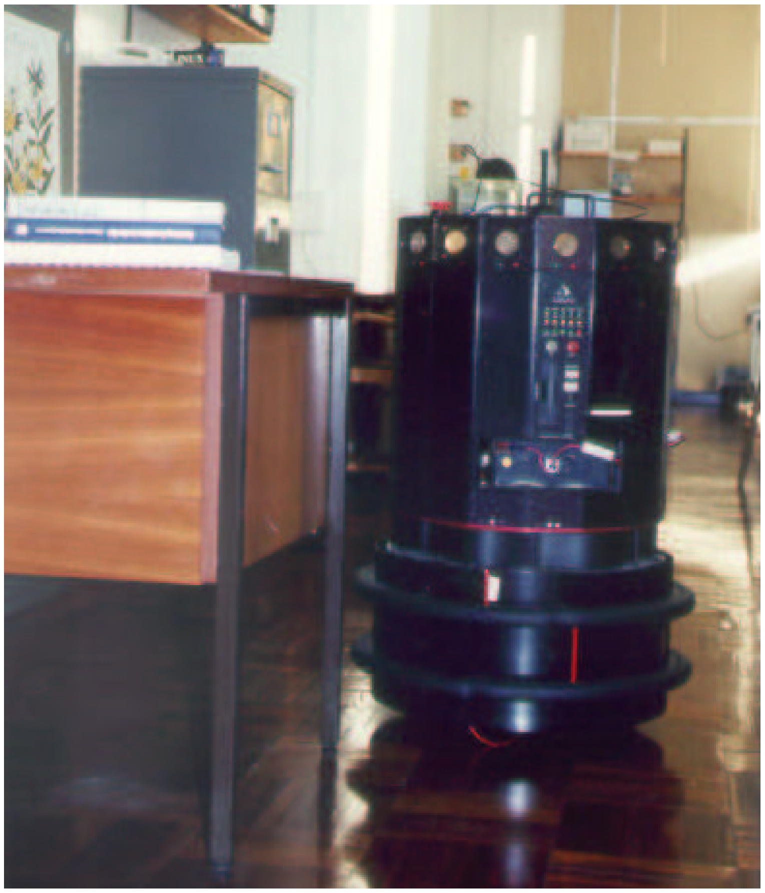


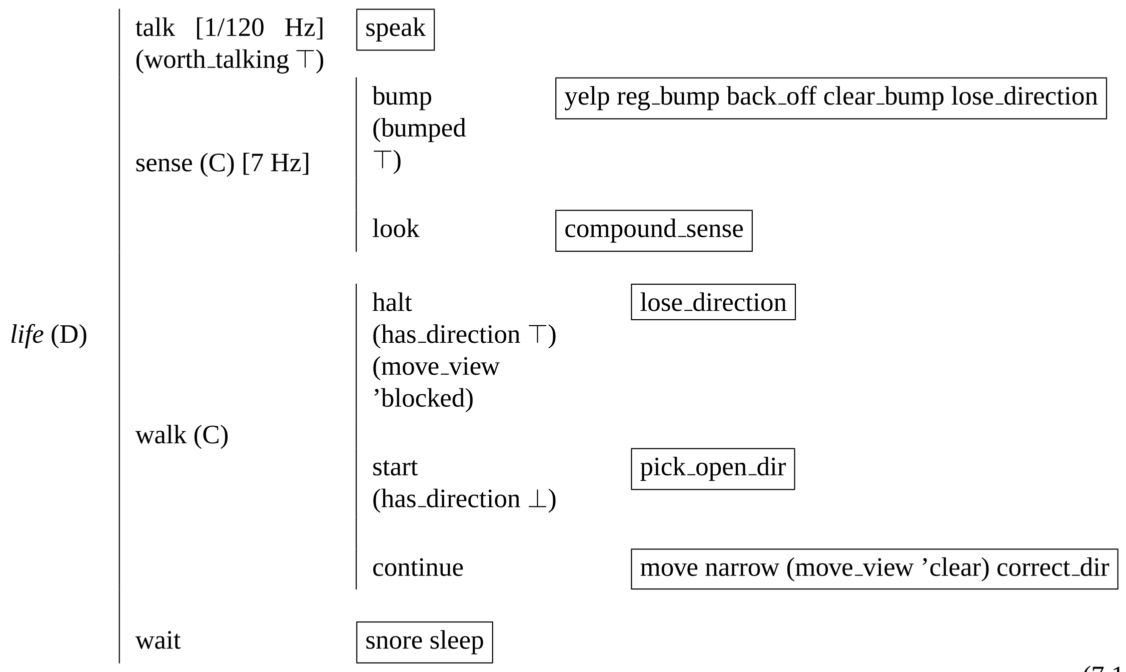
Figure 7-1: The Nomad 200 robot running in a laboratory at the University of Edinburgh. In this picture it has just encountered a desk — the surface is too low for the sonar sensors to detect when proximate, and the leg too dark for the infra-red sensors, so the robot had to learn about the desk with its bumpers.

determine how far and in what direction it has moved; these are also inaccurate over any great distance, both because of slippage and because the robot tends to slowly rotate anti-clockwise at a rate of a few degrees every 50 meters.

7.6.2 Example: Modular Learning and Perception

Plan 7.13 (in Figure 7-2) is the reactive plan for controlling the robot as it moves forward. This uses the same representation as plan 4.2, except that since it is a real-time agent, scheduling is in Hertz.

In this section, we will be concentrating on the primitives in the **sense** competence and in the **continue** action pattern (under the **walk** competence). However I will briefly introduce the whole plan. In several aspects, such as the bump action pattern and the representation underlying compound_sense and move, it bears a strong resemblance to the much simpler Plan 6.6. **Talk** provides information about the battery level if the charge level has fallen by a 10% increment (e.g. it's at 69% now and last time talk was called it was in


life (D)
talk [1/120 Hz] (worth_talking ⊤)
speak
sense (C) [7 Hz]
bump (bumped ⊤)
yelp reg_bump back_off clear_bump lose_direction
look
compound_sense
walk (C)
halt (has_direction ⊤) (move_view ’blocked)
lose_direction
start (has_direction ⊥)
pick_open_dir
continue
move narrow (move_view ’clear) correct_dir
wait
snore sleep

(7.13)

Figure 7-2: The plan for just keeping the Nomad moving in a crowded environment.

the 70s.) **Wait** should never trigger, but is there for debugging purposes, and was indeed called a number of times. Having a lowest-priority drive that conspicuously announces itself (in this case, by saying “Z Z Z” through the robot’s speech card) keeps the program running so a debugger can check internal state. If no element of a drive collection triggers, like any other BRP the drive collection simply terminates.

The behaviors used to support **sense** and **continue** are shown in Figure 7-3. Again, bold face indicates primitives that occur in the POSH action-selection scripts. The main behaviors are Direction, which determines speed as well as orientation of motion, and C-Sense, which represents the robot’s best estimate of how much free space it has around it.

The C-sense state is a simple vector of 16 integer values (one for each face) representing inches to the next obstacle in that direction. Every time **compound_sense** is called (roughly 7 times a second) this representation is updated. C-Sense combines information from the robot’s three sensor systems, essentially taking the shortest value from each. The exception to this is the infra-red reading, which has a maximum value of 15 inches. Infra-red readings of greater than 12 inches are ignored. Sonar readings are accurate to a much greater distance, but are inaccurate under 5 inches. Also, the sonar is situated at the top of the robot, and the infra-red near its base, so the two systems sometimes report different obstacles that occur at different heights.

Sonar is susceptible to a number of confounding effects which can make the readings grossly inaccurate. A simple, fairly robust solution employed on this robot is to not believe a reading that changes radically unless it persists for at least half a second. Given that sensing occurs at 7 Hz, this requires a three-event perceptual memory. This is represented

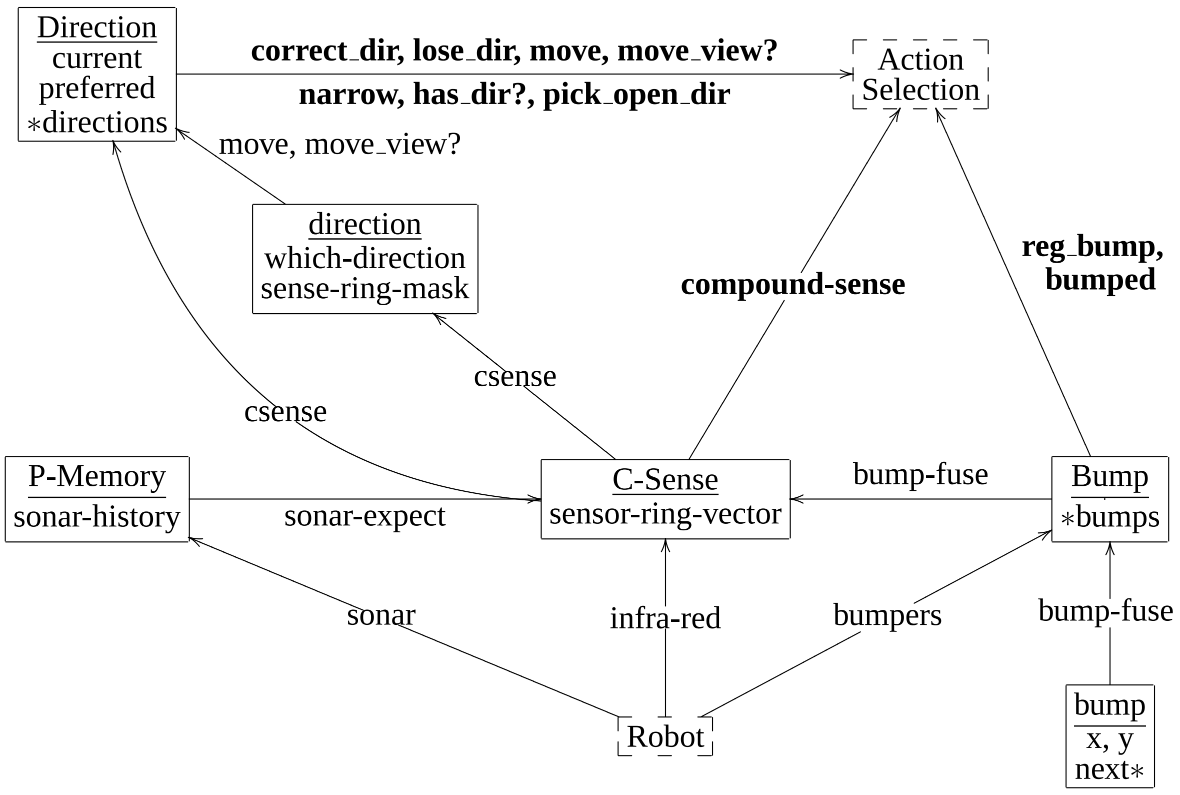


Figure 7-3: The behaviors involved in moving forward. Notation as in Section 2.2. See text for explanation.

and processed in P-Memory.

Bumps are a more difficult problem. As already described in Section 6.5, touch sensors necessarily require storing state until the obstacle is successfully navigated around. For the reasons described just above in Section 7.6.1, this can actually take several minutes. The behavior Bump retains records of impacted obstacles until the robot has moved several feet away from them, rather than for a fixed time. It also calculates the original location of the bump (given the bumper number and the robot's radius) and the face and distance on which to appropriately represent that object's location. Of course, bumps are necessarily points while obstacles are not, but the robot's tendency to move a good distance around any known obstruction generally clears the full obstacle. If not, it may be represented by multiple impacts.

Since the robot's libraries were written in C++, I took advantage of the concepts of class and instance variable state. Thus there can be any number of bump instances representing the actual impacts, while Bump propper keeps track of them. Similarly, there is a direction for each face of the robot. The direction behaviors include permenant state representing a mask applied to the C-Sense vector when that direction is controlling behavior. Velocity is determined by the distance to the nearest obstacle, but obstacles to the sides of the direction of motion are discounted, and to the rear are ignored. Similar discounts apply for steering corrections around obstacles. Direction keeps track of the current 'intended' direction of motion, and also the actual direction. Actual direction may be changed by **move** in order to avoid an obstacle. If the view is clear, **correct_dir** gradually transfers the current direction back to the preferred one.

7.6.3 Example: Episodic Memory and Navigation

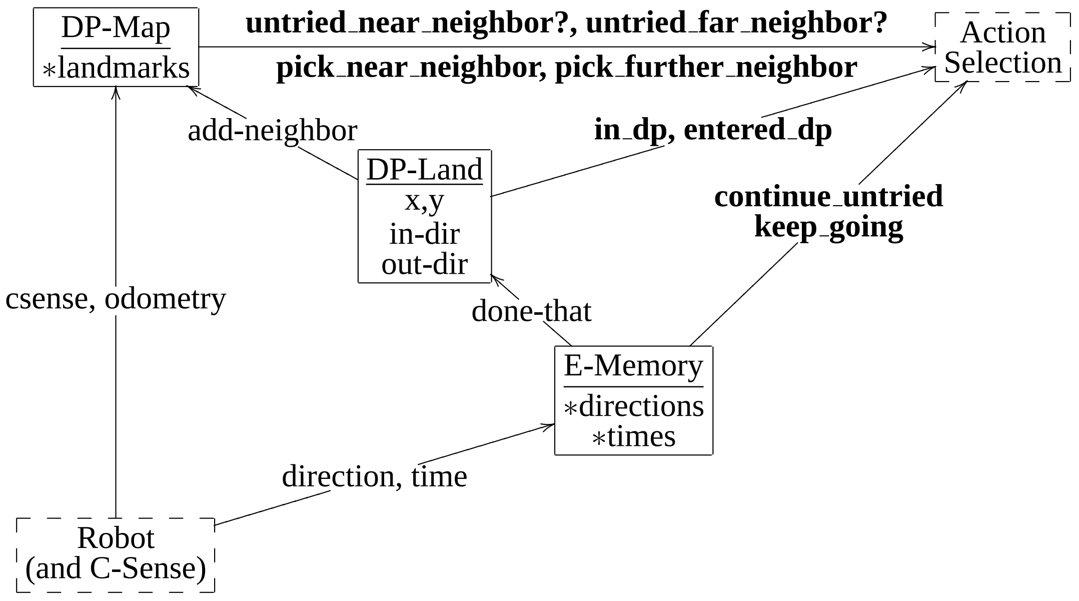


Figure 7-4: Behaviors supporting the plan for more informed navigation.

Plan 7.13 (in Figure 7-2 allows the robot to be fully autonomous from the control standpoint, and very easy to herd around a room in practice. However, my ambition was to make the robot able to learn tours through the laboratory through a few simple demonstrations or instructions. I achieved this goal for relatively small tours, but scaling it to long-term navigation of the entire multi-room laboratory space required further perceptual routines for recognizing locations and recalibrating orientation. Location recognition proved straightforward using sonar signatures, but the work was never completed due to lack of development time. This section shows only the additions and changes made to the wandering agent's architecture that allowed the robot to learn and follow a short route.

The behaviors developed for the small-tour version are shown in Figure 7-4. From a BOD standpoint, these behaviors are compositionally the same as those in Figure 7-3. However because their state changes more slowly and persists longer, their adaptations coincide better with the conventional definition of 'learning'. The robot learns a map over the lifetime of the agent and also has short-term episodic recall to help it select an appropriate direction.

The new behavior DP-Land is for recognizing decision points, locations where there are clearly more than two directions (forward and back) the robot could move in. As Plan 7.14 (Figure 7-5) shows, the robot 'loses direction' any time it either finds it can no longer move in at least approximately its currently intended direction, and whenever it enters a new decision point. When the robot does not have a direction, it attempts to remember a direction that previously worked from somewhere nearby this location. If it hasn't already

$$
\text{walk (C)}\ \left|\ \begin{array}{ll}
\begin{array}{l}\text{halt}\\ \text{(has\_direction }\top\text{)}\\ \text{(move\_view}\\ \text{'blocked)}\end{array} & \boxed{\text{lose\_direction}} \\
\text{cogitate\_route (C)} & \left|\ \begin{array}{ll} \begin{array}{l}\text{enter\_dp (in\_dp }\bot\text{)}\\ \text{(entered\_dp }\bot\text{)}\end{array} & \boxed{\text{lose\_direction greet\_dp}} \\ \begin{array}{l}\text{leave\_dp (in\_dp }\top\text{)}\\ \text{(entered\_dp }\top\text{)}\end{array} & \boxed{\text{dismiss\_dp}} \end{array}\right. \\
\text{pick\_direction (C)} & \left|\ \begin{array}{ll} \begin{array}{l}\text{look\_up}\\ \text{(untried\_near\_neighbor}\\ \top\text{)}\end{array} & \boxed{\text{pick\_near\_neighbor}} \\ \begin{array}{l}\text{keep\_going}\\ \text{(continue\_untried }\top\text{)}\end{array} & \boxed{\text{pick\_previous\_direction}} \\ \begin{array}{l}\text{desperate\_look\_up}\\ \text{(untried\_far\_neighbor}\\ \top\text{)}\end{array} & \boxed{\text{pick\_further\_neighbor}} \end{array}\right. \\
\begin{array}{l}\text{start}\\ \text{(has\_direction }\bot\text{)}\end{array} & \boxed{\text{ask\_directions}} \\
\text{continue} & \boxed{\text{move narrow (move\_view 'clear) correct\_dir}}
\end{array}\right. \tag{7.14}
$$

Figure 7-5: Changes in the *walk* competence for more informed navigation. See Section 7.6.4 for comments on slip-stack operation and this competence.

recently tried moving in that direction and failed, then that direction is chosen. Otherwise, if the robot has no clear record of why it stopped, it may have been halted by a sonar apparition or some other transient problem, so it may try continuing in its current direction. Or it may try a further afield neighbor.

If the pick_direction competence fails, then the robot asks for instruction. Every time the robot selects a direction, it records the direction, its current approximate location, and the current time in E-Memory, episodic memory. This is trimmed after a fixed number of events (16) which is greater than the number of tries the robot would ever attempt for one scenario before asking directions. Any decision or instruction which is successful (persists for at least a few inches of motion) is stored in long-term memory located in the behavior DP-Map.

7.6.4 Learning from the Robot Experiments

I have argued elsewhere [Bryson et al., 2000] that robots are sometimes over-rated as an experimental platform, particularly relative to other generally accessible complex platforms such as virtual reality or standardized simulations. However, it is impossible to deny their power both as engaging demonstrations, but also as 'intuition pumps' [Dennett, 1987]. I had two of the key insights of this dissertation while working on this robot.

Perceptual Learning and Behavior Decomposition

The robot examples above illustrate several different motivations for behavior decomposition — that is, for why one element of memory may be distinct from another.

- The original signal is very different (e.g. light vs. sound vs. an impact).
- The memory needs to record events that occur at different rates.
- Old memories needs to decay or be forgotten at different rates.
- Patterns from the memory emerge at different rates.
- Different representations are simplest or most minimal.

It was during one of my reimplementations of the POSH architecture (from PERL 5.003 to C++) that I came to fully equate persistent variable state with perception *and* with object-style representation. Although I had suggested that different rates of adaptation might be key to behavior decomposition several years earlier [Bryson, 1995], the importance of fully organizing a behavior library along these lines, and the intimate relationship between learning and perception, only struck me while I was reimplementing my own work on robot sensor fusion.

There is no substantial difference, from this BOD perspective, between transient perceptual state and lifetime learning. The term 'learning' seems to me to be arbitrarily applied somewhere along a continuum involving how frequently and to what extent state changes over time.

Maintaining More Action-Selection Context

Plan 7.14 will not work under the strictly 0-stack version of the POSH slip-stack documented in Section 4.6.2. Since this is the version of POSH I had running on the Nomad, Plan 7.14 is actually fudged. In fact, both the cogitate_route and pick_direction competences had to be wrapped inside of the DP-Map behavior. Pick_direction, for example, was expressed as a perceptual primitive that reported whether it had successfully found a viable stored direction, and a dummy action which formally 'picked' the already-selected direction. My intuition was that in fact a competence should be able to 'quickly' fail if none of its elements could fire, and let the next element of its parent competence fire. A year later I found a mechanism, the action scheduler of Ymir [Thórisson, 1999] which inspired my current implementation of POSH action selection, described in Section 4.6.3. The work leading to this insight is described in Section 12.3.2.

7.7 Meta-State and Learning to Learn

The final form of state is state about other state. In the context of the reactive hierarchical control architecture and modularized behavior, meta-state can be reduced to two forms: state about control state, and state about perceptual memory.

As I stated earlier, BOD gives no special support to this form of learning. Such development falls outside the remit of this dissertation, which focuses on engineering rather than autonomous approaches to development. Nevertheless, this is a fairly obvious area for research. Chapter 9 does demonstrate the modeling of a process for learning control structures similar to a BRP. In this section, and again in Chapter 11, I discuss possible extensions capable of meta-learning.

Work on social and cultural learning is very closely related to the specialized learning approach championed here. In BOD, the bias that enables learning derives from engineering; in social learning it derives from behavioral predispositions that facilitate absorbing the products of cultural evolution. BOD is a substitute for genetic evolution, but an obvious goal for autonomous agents would be to be able to absorb mimetic evolution directly from the surrounding culture as humans and other animals do [Whiten, 2000, de Waal, 2001].

Control state can be divided into two elements: the structure that indicates the flow of activation, and the activation itself. A great deal of work has been done on varying the way in which activation levels spread between behaviors. In particular, the ethological literature describes modules competing for attention by evaluating their own activation level; several ethologically inspired architectures do precisely this [Maes, 1989, Correia and Steiger-Garção, 1995, Blumberg, 1996, Cooper et al., 1995]. Blumberg [1996] has achieved probably the most coherent behavior in this genre by combining a strategy of resource-based intelligent constraints and mutual inhibition between competing behaviors. Blumberg also provides a mechanism for reconnecting the flow of control in order to implement conditioning. Under our typology, this strategy would be meta-learning, though conditioning could also be implemented on the perceptual level. Unfortunately, such strategies are extremely difficult to control whether by hand or by machine learning [Tyrrell, 1993]. In general, learning has only been successful in very constrained spaces with few behaviors.

When developing BOD, I experimented with flexible priorities within BRPs, but eventually replaced these structures with scheduling for the drive collections and fixed retry limits for competence elements. I found it easier to control the simulation of continuous valued internal drive levels [e.g. Lorenz, 1973, Tu, 1999] using perceptual state. Section 10.5 will demonstrate a model of this.

The actual structure of the action sequences and competences in POSH control were originally designed to support another form of meta-learning. I hoped to use evolutionary programming style processes to find new control sequences. (See for a similar theory [Calvin, 1996].) The easy problems for this approach are how to generate new thoughts and how to test and store a winning solution. The hard problems are how to tell when a new plan is needed, and how to recognize which solution wins. These are problems the case-based learning community have been researching for some time (see for example [Hammond, 1990, Ram and Santamaria, 1997].)

I now consider it unlikely, again due to the size of the search space, that such a strategy would be practical for useful complex agents. I consider social learning much more

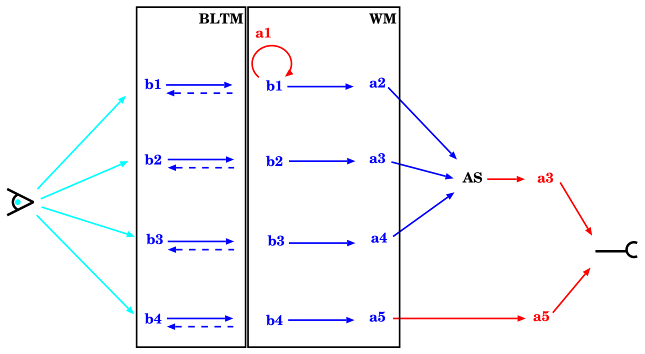


Figure 7-6: (Compare to Figure 2-1 on page 35.) A system capable of learning behaviors must 1) represent them on a common substrate in a 2) allow them to be modified. Here behaviors are represented in a special long-term memory (BLTM) *and* in a plastic working memory (WM) where they can be modified. During consolidation (dashed lines) modifications may either alter the original behaviors or create new ones.

likely to be a productive area of future research. Other areas of meta-level learning that might be interesting and productive to explore include implementing Norman and Shallice [1986]'s notion of deliberate action. Norman and Shallice's theory holds that deliberate control is triggered by special circumstances. Deliberation then monitors and modifies routine processes, possibly by slowing them or providing additional sensory information. Another, possibly related, area is the consolidation of episodic memory into usable skills [see Karmiloff-Smith, 1992, Wilson and McNaughton, 1994, Hexmoor, 1995, and below].

7.7.1 Meta-Learning and Distributed Representations

Behavior-oriented design requires the use of complex algorithms and specialized representations. Therefore I have argued that agent behavior is best developed in object-oriented languages. However, this representation may limit the autonomous learning of new behaviors. Learning new skill modules is clearly desirable, and has been the focus of significant research (see [Demiris and Hayes, 1999] for a recent example and review.) However, to date, most efforts on these lines would qualify as specialized learning *within* a single representation system, which is one skill module or behavior from the perspective of BOD.

If we could instead represent behaviors and homogeneous, distributed substrate such as artificial neural networks (ANN), we might more easily be able to produce this sort of learning. Consider Figure 7-6. In this figure, representation of the skill modules has been split into two functional modules: the Behavior Long-Term Memory (BLTM) and the Working Memory (WM). The working memory allows for rapid, short-term changes not only for perceptual memory, but also in the representation of the behaviors. The BLTM provides a relatively stable reference source for how these modules should appear when activated. Skill representations might be modified due to particular circumstances, such as compensating for tiredness or high wind, or responding to a novel situation such as using chopsticks on slippery rice noodles for the first time.

In this model, the adjustments made in plastic, short-term memory also affect the long-term memory. This sort of dual- or multi-rate learning is receiving a good deal of attention in ANN currently (see [French et al., 2000, Bullinaria, 2000, McClelland et al., 1995]). Depending on long-term experience, we would like this consolidation to have two possible effects. Let's imagine that b_2 has been modified in working memory in order to provide an appropriate expression of a_2. If the same modifications of b_2 prove useful in the near future, then they will be present for consolidation for a protracted period, and likely to effect the permanent representation of b_2. However, if the modifications are only sometimes applicable, we would like a new behavior ${b_2}'$ to become established. This process should also trigger perceptual learning, so that the two behaviors can discriminate their appropriate context for the purpose of action selection. Also b_2 and ${b_2}'$ would now be free to further specialize away from each other.

Unfortunately, the practical constraints for this model are similar to those described just above for evolving control structures. Also, ANN research in supporting modularity is only in its infancy [Wermter et al., 2001]. It is difficult to predict which strategy might become practical sooner.

7.8 Summary: Designing Learning

In summary, developing complex intelligent behavior requires not only modularized behavior, but also modularized learning. Agent programs should constrain and specialize each learning task as much as possible. Control state records both static knowledge of procedure and dynamic decisions on current approach. Deictic variables simplify control state by providing a point of reference through which procedures can be generalized across contexts. Perceptual and motor memory allows control to be determined by factors that appear only across time rather than in the immediate environment. Meta-state allows the agent to learn from and change its own behavior. In BOD, however, there is little difference between meta-state and perceptual learning. Where it is used, it has been designed and supported in exactly the same way. We will see an extended example of this in Chapter 9.

Chapter 8

Behavior-Oriented Design

8.1 Introduction

Chapters 4–7 have described the elements of the BOD agent architecture in some detail. I have also discussed the trade-offs that need to be made to build the simplest agent possible. The simpler an agent, the more likely it is to be successful. This means limiting as much as possible both the developer's search for the correct code (or, during maintenance, for bugs), and also the agent's search for the correct solution for any activity it must learn or plan.

The BOD methodology was introduced in Chapter 2; in this chapter it is presented more formally. I first describe the essential procedures: the initial process of specifying and decomposing the agents goals into BOD architecture representations, and the ongoing, iterative process for building and maintaining a complex agent. I then discuss some of the practical aspects of BOD development, such as documentation and tool use. Finally, I briefly discuss how this approach relates to other high-level approaches. Chapter 9 will provide a detailed example of a BOD development process.

8.2 The Basic BOD Development Process

8.2.1 The Initial Decomposition

The initial decomposition is a set of steps. Executing them correctly is not critical, since the main development strategy includes correcting assumptions from this stage of the process. Nevertheless, good work at this stage greatly facilitates the rest of the process.

The steps of initial decomposition are the following:

1. Specify at a high level what the agent is intended to do.
2. Describe likely activities in terms of sequences of actions. These sequences are the the basis of the initial reactive plans.
3. Identify an initial list of sensory and action primitives from the previous list of actions.

4. Identify the state necessary to enable the described primitives and drives. Cluster related state elements and their primitives into specifications for behaviors. This is the basis of the behavior library.

5. Identify and prioritize goals or drives that the agent may need to attend to. This describes the initial roots for the POSH action selection hierarchy.

6. Select a first behavior to implement.

Sections 9.3.1 and 12.2.1 contain extended examples of this process.

The lists compiled during this process should be kept, since they are an important part of the documentation of the agent. Documenting BOD agents, which is done primarily through well-organized source code, is covered in Section 8.4.1 below.

In selecting the first behavior, it is often a good idea to choose a simple, low-level priority that can be continuously active, so that the agent doesn't 'die' immediately. For example, on the mobile robot in Section 7.6.2, the bottom-most priority of the main drive hierarchy was 'wait', a function which keeps track of the time and snores every 30 seconds or so. This sort of behavior gives the developer a clear indication that the robot's control has not crashed, and also that none of its interesting behaviors can currently trigger.

Depending on the project, it may make sense to start with a competence rather than worrying about a drive collection. Examples of projects I started like this are the blocks-world experiments in Section 7.4 and the transitive inference experiments in Chapter 9. Both of these projects:

- required building a simulation, so truly basic behaviors needed to be tested,

- were built in conjunction with debugging a new implementation of POSH control, and

- emphasized a cognitive capacity that is expressed in isolation, rather than the balancing of conflicting goals over a period of time.

8.2.2 Iterative Development

The heart of the BOD methodology is an iterative development process

1. Select a part of the specification to implement next.

2. Extend the agent with that implementation:

   - code behaviors and reactive plans, and
   - test and debug that code.

3. Revise the current specification.

BOD's iterative development cycle can be thought of as sort of a hand-cranked version of the EM (expectation maximization) algorithm [Dempster et al., 1977]. The first step is to elaborate the current model, then the second is to revise the model to find the new optimum representation. Of course, regardless of the optimizing process, the agent will continue to grow in complexity. But if that growth is carefully monitored, guided and pruned, then the resulting agent will be more elegant, easier to maintain, and easier to further adapt.

Reactive plans grow in complexity over the development of an agent. Also, frequently multiple reactive plans are developed for a single platform, each creating agents with different overall behavior characteristics, such as goals or personality. All working plans should be preserved, each commented with the date of its development, a description of its behavior, and a record of any plans from which it was derived. This library forms a testing suite, and should be checked regularly when changes are made to the behavior library.

Even when there are radically different plan scripts for the same platform or domain, there will generally only be one behavior library — one set of code. Of course, each agent will have its own instance or instances of behavior objects when it is running, and may potentially save their run-time state in its own persistent object storage. But it is worth making an effort to support all scripts in a single library of behavior code. In fact, testing policy advocated above necessitates this.

Testing should be done as frequently as possible. Using languages that do not require compiling or strong typing, such as lisp or perl, significantly speeds the development process, though they may slow program execution time. "Optimize later", one of the modern mantras of software engineering, applies to programming languages too. In my experience, the time spent developing an AI agent generally far outweighs the time spent watching the agent run. Particularly for interactive real-time agents like robots and VR characters, the bottle-necks are much more likely to be caused by motor constraints or speech-recognition than by the intelligent control architecture.

8.3 Making It Work

8.3.1 Revising the Specifications

The most interesting part of the BOD methodology is the set of rules for revising the specifications. In general, the main design principle of BOD is *when in doubt, favor simplicity.* A primitive is preferred to an action sequence, a sequence to a competence. Similarly, control state is preferred to learned state, specialized learning to general purpose learning or planning. Given this bias, heuristics are then used indicate when the simple element must be broken into a more complex one.

A guiding principle in all software engineering is to reduce redundancy. If a particular plan or behavior can be reused, it should be. As in OOD, if only part of a plan or a primitive action can be used, then a change in decomposition is called for. In the case of the action primitive, the primitive should be decomposed into two or more primitives, and the original action replaced by a plan element. The new plan element should have the same name and functionality as the original action. This allows established plans to continue operating with only minimal change.

If a sequence sometimes needs to contain a cycle, or often does not need some of its elements to fire, then it is really a competence, not an action pattern. If a competence is actually deterministic, if it nearly always actually executes a fixed path through its elements, then it should be simplified into a sequence.

The main point is to be empirical. If one approach is too much trouble or is giving debugging problems, try another. It is important to remember that programmer experience is one of the key selective pressures in BOD for keeping the agent simple. BOD provides at least two paths to simplicity and clarity, with cyclic development and some trial and error we expect the programmer to determine which path is best for a particular situation [Parnas and Clements, 1986, Boehm, 1986]. This is also why modularity and maintainability are key to BOD: programmers are to be encouraged to change the architecture of an agent when they find a better solution. Such changes should be easy to make. Further, they should be transparent, or at least easy to follow and understand, when another team member encounters them.

8.3.2 Competences

A competence is effectively developed out of the sequence of behaviors that might need to be executed in a worst-case scenario. The ultimate (last / goal) step is the highest priority element of the competence, the penultimate the second highest and so on. Triggers on each element determine whether that element is able to fire at this instant, or whether a lower priority one must first be fired in hopes of enabling it.

Competences are really the basic level of operation for reactive plans, and a great deal of time may be spent programming them. Here are some indications gathered from competences that the specification needs to be redesigned:

- *Complex Triggers*: reactive plan elements should not require long or complex triggers. Perception should be handled at the behavior level; it should be a skill. Thus a large number of triggers may indicate the requirement for a new behavior or a new method on an existing behavior to appropriately categorize the context for firing the competence elements. Whether a new behavior or simply a new method is called for is determined by whether or not more state is needed to make that categorization: new state generally implies a new behavior.

- *Too Many Elements*: Competences usually need no more than 5 to 7 elements, they may contain fewer. Sometimes competences get cluttered (and triggers complicated) because there are really two different solutions to the same problem. In this case, the competence should be split. If the two paths lead all the way to the goal, then the competence is really two siblings which should be discriminated between at the level of the current competence's parent. If the dual pathway is only for part of the competence, then the competence should contain two children.

Effectively every step of the competence but the highest priority one is a subgoal. If there is more than one way to achieve that subgoal, trying to express both of them in the same competence can split attention resources and lead to dithering or 'trigger-flipping' (where two plan elements serve only to activate each other's precondition). The purpose of a competence is to focus attention on *one* solution at a time.

Nilsson [1994, p. 142] emphasizes more formally for his teleo-reactive BRPs that each action should be expected to achieve a condition (prerequisite) of a higher plan step. He defines the *regression property,* which holds for a plan where, for every element, the above is true. He also defines *completeness* for a BRP as being true if the conjunction of all the BRP plan steps' releasers is a tautology. Since BOD considers chaining between competences as a reasonable design pattern, I do not necessarily recommend the regression property. Also, since BOD emphasizes relying on the POSH control hierarchy to specialize any particular plan's circumstances, I do not strongly recommend maintaining completeness in plans. Nevertheless, these concepts are useful to keep in mind, particularly when debugging BRPs.

The 'complex trigger' heuristic brings us back to the question of trading off control complexity and behavior state discussed in Chapters 6 and 7 above, particularly in Section 6.4.2. To reiterate, it is fairly easy to tell when deictic representation will be useful, since it is typically a simple variable stored in order to remove redundant control code. Determining when to build a full-fledged categorizing behavior and when to add control code instead is more complicated.

8.3.3 Prioritization and Drive Collections

Getting the priorities right in a POSH system can also be non-intuitive. Clear cut priorities, like "*Always* run if you see a cat chasing you" are fairly simple, but even in the case of the ALife rodent in Section 4.5.1 things aren't necessarily that clear cut. How certain do you need to be that there's a cat? How often should you look for cats?

Standard BRP prioritization works fine for any prioritization that is always strictly ordered. But it quickly became obvious that there needs to be a mechanism for attending to things that aren't very important on spare cycles. For this purpose I have also provided a scheduling system in the Drive Collection.

Scheduling in POSH systems is inexact — the current implementations at least use course-grained and best-effort scheduling. Too many things may be scheduled per second with no direct indication that they are failing to execute. This is to some extent an artifact of the reactive nature of the system — we expect some events and behaviors to arrest the attention of the agent. An initial schedule can be computed with the help of simple benchmarking. Facilities for benchmarking are built into the control of the C++ POSH implementation — for the CLOS version, I use a commercial profiler instead.

Profiling helps determine the constraints on the number of elements that can run per second, which allows estimates for the rate at which various numbers of drives can be executed. For example, on the Nomad robot in Section 7.6.2, sensing is fairly expensive and limits the number of cycles for the architecture to about 340Hz. However, if the robot stops moving and reduces its sense sample rate, the cycle rate increases by an order of magnitude. This suggests that the robot should engage in phased periods of activity, for example switching between sense-intensive work like exploration and compute-intensive work like map learning. This strategy is found in most mammals (e.g. Wilson and McNaughton

[1994]), and is one of the motivations for the slip-stack / competence-chaining facility in POSH action-selection.

There are other ways to correct prioritization problems. One is switching elements between competences and drives at the control script level. Elements in drives can be scheduled the most reliably. However, an element which is only required in a particular context may waste cycles if it is scheduled as a drive. The AP for handling bumps in Plan 7.13 was initially added into the competence for motion, ‘walk’, but this overlooked the fact some of the competing BRP steps in walk were long and the bumpers would not be checked between their elements once they were triggered. Consequently, ‘bump’ was moved to the top level drive. ‘Compound_sense’ was subsequently moved to the same drive, so that both could be limited to 7Hz, because the hardware checks involved in sensing had relatively long durations and were proving a bottleneck. On the other hand, the perceptual routine for watching for landmarks was switched *into* ‘walk’ since it was only needed in the context of motion, and could rely on perceptual memory if it was called less frequently.

One problem with using scheduling is that the *entire drive* is only triggered at the scheduled rate, which can be a problem if what is really wanted is to be able to switch attention to that drive until a problem is solved. Because of this problem, the AP for disengaging the robot after it has bumped into something in Plan 7.13 is actually all expressed as a trigger, not an action (triggers are executed atomically, but like all APs terminate on the first sense or action that fails). Some researchers use systems inspired by hormone levels to effectively latch a particular priority ordering for a period of time [e.g. Grand et al., 1997, Cañamero, 1997, Breazeal and Scassellati, 1999a, Frankel and Ray, 2000]. Such a strategy can be used under BOD by creating a behavior for each ‘hormone’ or ‘emotion’ level, then using sense preconditions based on those behaviors (see Section 10.5 for an example).

8.4 Supporting Development and Maintenance

The entire point of BOD is to make it as easy as possible for a designer to make an agent work correctly. This means leveraging any strategies and tools that reliably increase productivity. *Reliably* here means not only that the tools or strategies must work correctly when used properly, but that they work well when they are used in the ways programmers are likely to use them. The three strategies I highlight in this section really are worth the effort of implementing, because the cost of using them is very low, even negligible, and the payoff is very high.

8.4.1 Document the Agent Specification in Program Code

The concept of self-documenting code is old and well-argued. The primary argument for having documentation being a functioning part of a project is that this is the only way to ensure that the documentation will never get out of synchronization with the actual project. The primary argument against it is that code is never really that easy to read, and will never be concisely summarized.

Because of the way BOD is structured, it is particularly straight-forward to make the most important parts of the system well documented. I *strongly* advise using the following guidelines.

Document the Plan / Behavior Interface in One Program File

Remember the **what**s from Chapter 2? These are the primitives of the reactive plans, which must be defined in terms of methods on the behavior objects. For each behavior library, there should be one code file that creates this interface. In my implementations of POSH action selection, each primitive must be wrapped in an object which is either an *act* or a *sense*. The code executed when that object is triggered is usually only one or two lines long, typically a method call on some behavior object. I cluster the primitives by the behaviors that support them, and use program comments to make the divisions between behaviors clear.

This is the main documentation for the specification — it is the only file likely to have both current *and intended* specifications listed. This is where I list the names of behaviors and primitives determined during decomposition, even before they have been implemented. Intended reactive plans are usually written as scripts (see below.)

Each Behavior should have Its Program File

Essentially, every behavior is well commented automatically if it is really implemented as an object. One can easily see the state and representations in the class definition. Even in languages that don't require methods to be defined in the class declaration, it is only good style to include all the methods of a class in the same source file with the class definition.

Comment the Reactive Plan Scripts

This is the one suggestion that really takes discipline — or preferably a GUI tool to facilitate it. Every script should contain a comment with:

- Its name. Necessary in order to make it obvious if a plan is copied and changed without updating the comment. In that case, the name won't match the file name.
- What script(s) it was derived from. Most scripts are improvements of older working scripts, though some are shortened versions of a script that needs to be debugged.
- The date it was created.
- The date it started working, if that's significantly different. Since writing scripts is part of the specification process, some scripts will be ambitious plans for the future rather than working code.
- The date and reasons it was abandoned, if it was abandoned.
- Possibly, dates and explanations of any changes. Normally, changes shouldn't happen in a script once it works (or is abandoned) — they should be made in new scripts, and the old ones kept for a record.

When tempted not to save old scripts remember: those who forget history are doomed to repeat it. Also, old script histories make great source material for papers documenting a project's history (e.g. see Chapter 9). From a single historic script and the current interface file, it is easy to reconstruct what the behavior library must have contained at the time the script was constructed. If revision control is also practiced (see below) then there can be direct access to behavior files from the appropriate dates as well.

8.4.2 Use Revision Control

Again, revision control systems are well established software tools, so they will not be covered in depth here. I *will* strongly recommend that some system be used, particularly on multi-programmer projects, but even for single programmers. In particular, I recommend CVS[1]. This allows you to easily create a directory containing your latest working version of an agent (or any version) for a demo on a moment's notice, and without interfering with your current development directory. It also helps handle the situation when more than one version of the source code has been handled simultaneously. It works on any size project — I first used it working for a multi-national company with firewalls. It really is worth the time it takes to install the system.

8.4.3 Use Debugging Tools

Again, good tools are worth taking the time to learn to use, and sometimes to build. I prefer starting with languages or language packages that provide a decent level of debugging, rather than relying too much on customized agent toolkits. At a minimum, any developer should have a profiler, to find out (rather than guess) where bottlenecks are happening in code. Ideally, they should also have an interruptible debugger that can show program state, and preferably allow stepping through the program execution. Finally, class browsers are extremely useful tools that both help the developer quickly find code or functions in the language, and often in their own program.

I know of two very successful complex agent research projects that found logging actions at multiple, discrete levels (e.g. high level decisions, mid-level decisions, low-level actions) absolutely critical to debugging their code. These are the Ymir multi-modal dialog agent project [Thórisson, 1997] and the CMUnited robo-cup soccer teams [Riley et al., 2001]. I have rudimentary versions of such logging in my own system, though I personally find real-time notification of events and stepping control more useful. Certainly, building GUIs to help automate the most time-consuming and often needed aspects of development and maintenance can be well worth the time.

Again, there is an entire literature on agent toolkits which I will not even try to summarize here [though see Bryson et al., 2001]. This section has outlined what I consider to be the minimum amount of support useful to develop intelligent agents.

[1]The Concurrent Versions System, `http://cvshome.org`.

8.5 How BOD Relates to Similar Approaches

Finally, I will give a quick, high-level summary of the relationship between behavior-oriented design and the other design techniques that have to some extent inspired it. First, all of these approaches are high-level methodologies, none of them are algorithms. In most cases, it should be at least *possible* to build an agent under any of these approaches. The important question is, how easy (and consequently, how likely) it is to solve a particular problem using a particular strategy.

8.5.1 Behavior-Based AI (BBAI)

There are two main benefits of BOD over standard BBAI: BOD's use of hierarchical reactive plans, and BOD's methodology of behavior decomposition.

As I have argued frequently earlier, having explicit reactive plans built as part of the architecture greatly simplifies control. When one particular set of behaviors is active (say a robot is trying to pick up a teacup) there is no need to worry about the interactions of other unrelated behaviors. The robot will not decide to sit down, or relieve itself, or go see a movie unless it is at a reasonable juncture with the tea cup. On the other hand, it may drop the cup if something truly important happens, for example if it must fend off an attack from a large dog trying to knock it over. It is much easier to express this information in a reactive plan than to build complex mutual inhibition systems for each new behavior every time a behavior is added, as is necessary in conventional BBAI. In mutual inhibition or reinforcement systems, the control problem scales exponentially; with explicit plans, the problem scales linearly.

What BOD offers in terms of behavior decomposition over other BBAI methods is:

- A better place to start. Instead of trying to determine what the units of behavior are, the developer determines what information the agent is going to need. This is one of the chief insights from OOD.
- A better way to fix things. Unlike other BBAI approaches, BOD does not necessarily assume that decomposition is done correctly on the first attempt. It provides for cyclic development and neat interfaces between behaviors and control.

8.5.2 Object-Oriented Design (OOD)

Although BOD is to some extent based on OOD, it is not a fully object-oriented approach. OOD tends to be useful for passive reactive systems, but is used less frequently for designing systems that are actively internally motivated. The addition BOD provides over OOD is the reactive-plan component. This allows the expression of motivation and priority as part of the organization of behavior. It also separates the problem of organizing behavior through time from the problem of representing information about how the behavior is conducted. BOD applies techniques for building plans and decomposing behaviors that are analogous to, but not exactly the same as the OOD methodologies for designing object hierarchies. There is no equivalent, to OOD notions of inheritance in BOD [2], or to the class hierarchy. In BOD the behaviors are not hierarchical, the reactive plans are.

8.5.3 Agent-Oriented Design (AOD)

On the other hand, if *every* object given its own motivations and intentions, then that system would be called a Multi-Agent System, or MAS. Programming with MAS is sometimes now called Agent-Oriented Software Engineering [Ciancarini and Wooldridge, 2001].

I believe that AOD is overkill for creating a single complex agent, though some people are using it as a modular agent architecture. There are two problems with AOD in this context:

- Arbitration is often done by voting or some other distributed algorithm. This is not only potentially slow, but also involves somehow being able to determine for the entire system the relative value between individual agent's goals [Sandholm, 1999]. For a complex agent with a fixed number of component modules, by the time this prioritization is done, one has essentially done the work of creating a POSH plan, which will presumably then operate more quickly.

- Communication between agents is over-emphasized, and often relies on a homogeneous representation. Again, I favor customizing the interface as part of the design of the complex agent.

The AOD approach makes sense when a fluctuating number of agents are running on different hardware and owned by different people are trying to negotiate the solution to a well-defined problem or set of problems. But as far as developing intelligence for a single complex agent goes, it seems to be another instance of overly homogeneous representations, and overly obscure control.

8.5.4 'Behavior-Oriented Design'

Two authors before me have used the term Behavior-Oriented Design, both in 1994. Neither author has continued using the term to any great degree. One author, Nakata [1994], dealt with automated design using causal reasoning about the way a device is intended to work. This work is completely unrelated to my own. The other author, [Steels, 1994b], does work in the complex agent field, and was discussing the development of behavior-based robots. The Steels paper is primarily devoted to documenting his extremely impressive robot experimental enclosure, where robots competed or cooperated in generating then acquiring electric power. His only mention of the actual process of "behavior-oriented design" though is as part of a definition:

> A behavior-oriented design starts by identifying desirable behaviors and then seeking the subset for which behavior systems need to be developed.
>
> [Steels, 1994b, p. 447]

[2] Though see the use of drive-levels in Section 10.5.

This approach speaks to the decomposition of the parallel modules in his robot architecture. He speaks elsewhere of a “behavior-oriented as opposed to a goal-oriented design” [pp. 445, 451]. This phrase does not encompass his entire architecture or philosophy, which at that point included avoiding the use of action selection.

I view the work of this dissertation as a significant elaboration on this earlier work of Steels.

8.6 Conclusions

This chapter has gone into detail in describing the BOD methodological process, that is, how to put a BOD agent together. I have described what the specification for a BOD agent needs to contain. I have also described the importance of developing both the agent *and its specification* in order to continually optimize the agent for simplicity. This is necessary in order to keep the agent easy to extend, adapt and maintain; and also to keep control of the complexity of the problems the agent will be expected to learn or solve for itself. I have also described several important ancillary issues to supporting a BOD development effort, such as how to guarantee that the specification is kept up-to-date, and how to make best use of programmer time and resources. Finally, I have briefly related BOD to other related high-level software methodologies.

The next chapter is also devoted to explaining BOD methodology, but this time by example, not by instruction.

Chapter 9

Design Process Example: Modeling Transitive Inference in Primates

9.1 Introduction

In previous chapters I have shown complete working BOD systems but only toy examples to illustrate the design process. In this chapter I document a complete design process in order to illustrate the process of developing and scaling a BOD system while simultaneously controlling its complexity.

The system developed here is built to examine a particular competence in real animals — the ability to perform transitive inference (described below). As such, it has a relatively simple drive structure, since it models only two compatible goals (learning the task, and training the agent to learn the task). Also, because the experimental agents are situated only in an artificial-life simulation, there is no complex perceptual or motor control. Consequently, the behavior code is relatively brief, concentrating primarily on coordinating behavior, the learning task itself and reporting results. The full code appears in Appendix B.

The experiments in this chapter are also intrinsically interesting because they extend the current models of transitive-inference learning available in the psychology and biology literatures. Also, the capacity being learned is in itself very BRP-like. The difficulty both monkeys and humans have learning this task supports the emphasis in this dissertation on the design rather than learning of action selection mechanisms.

9.2 Biological Background: Learning Ordered Behavior

Transitive inference is the process of reasoning whereby one deduces that if, for some quality, $A > B$ and $B > C$ then $A > C$. In some domains, such as real numbers, this property will hold for any A, B or C. For other domains, such as sporting competitions and primate dominance hierarchies, the property does not necessarily hold. For example, international tennis rankings, while in general being a good predictor of the outcome of tennis matches, may systematically fail to predict the outcome of two particular players[1]

Piaget described transitive inference as an example of concrete operational thought [Piaget, 1954]. That is, children become capable of doing transitive inference when they become capable of mentally performing the physical manipulations that would determine the correct answer. However, since the 1970's, demonstrations of transitivity with preoperational children and various animals — apes, monkeys, rats and even pigeons [see for review Wynne, 1998] — has lead some to the conclusion that transitive inference is a basic animal competence, not a skill of operations or logic. In fact Siemann and Delius [1993] [reported in Wynne, 1998] have shown that in adults trained in the context of an exploration-type computer game, there was no performance difference between the individuals who formed explicit models of the comparisons and those who did not ($N = 8$ vs. 7 respectively).

9.2.1 Training a Subject for Transitive-Inference Experiments

P1	Each pair in order (*ED*, *DC*, *CB*, *BA*) repeated until 9 of 10 most recent trials are correct. Reject if requires over 200 trials total
P2a	4 of each pair in order. Criteria: 32 consecutive trials correct. Reject if requires over 200 trials total
P2b	2 of each pair in order. Criteria: 16 consecutive trials correct. Reject if requires over 200 trials total
P2c	1 of each pair in order. Criteria: 30 consecutive trials correct. No rejection criteria.
P3	1 of each pair randomly ordered. Criteria: 24 consecutive trials correct. Reject if requires over 200 trials total
T1	Binary tests: 6 sets of 10 pairs in random order. Reward unless failed training pair.
T2a	As in P3 for 32 trials. Unless 90% correct, redo P3.
T2	6 sets of 10 trigrams in random order, reward for all.
T3	Extended version of T2.

Figure 9-1: Phases of training and testing transitive inference [Chalmers and McGonigle, 1984]. When a subject reaches criteria, it proceeds to the next phase.

Training a subject to perform transitive inference is not trivial. Subjects are trained on a number of ordered pairs, typically in batches. Because of the end anchor effect (described below in Section 9.2.2), the chain must have a length of at least five items ($A \ldots E$) in order to clearly demonstrate transitivity on just one untrained pair (*BD*). Obviously, seven or more items would give further information, but training for transitivity is notoriously difficult. Even children who can master five items often cannot master seven[2]. Individual

[1]British sports vernacular has a word for such non-transitive relationships: the lower ranking team is called a "giantkiller".

[2]Interestingly, this is also true for even conventional sorting of conspicuously ordered items such as posts of different lengths [see e.g. McGonigle and Chalmers, 1996].

items are generally labeled in some arbitrary way, designed to be non-ordinal, such as by color or pattern. Additional control is normally provided by assigning different color or pattern orderings for different subjects.

The subjects are first taught the use of whatever apparatus they will be tested on – that is, they are rewarded for selecting one option. In the case I will describe below, the subjects learn to look under a brightly colored metal box for a reward. They are next exposed to the first pair *DE*, where only one element, *D* is rewarded[3] When the subject has learned this to some criteria, they are trained on *CD*. After all pairs are trained in this manner, there is often a phase of repeated ordered training with fewer exposures (e.g. 4 in a row of each pair) followed by a period of random presentations of training pairs.

9.2.2 Modeling and Effects in Successful Transitive Inference

Once a subject has been trained to criteria, they are exposed to the testing data. In testing, either choice is rewarded, though sometimes differentially rewarded training pairs are interspersed with testing pairs to ensure they are not forgotten. Subjects that have achieved criteria on the training phase tend to show significantly above-chance results for transitive inference. They also show the following effects [see Wynne, 1998, for review]:

- *The End Anchor Effect*: subjects do better (more correct and faster reaction times) when a test pair contains one of the ends. Usually explained by the fact they have learned only one behavior with regard to the end points (e.g. nothing is $> A$).

- *The Serial Position Effect*: even taking into account the end anchor effect, subjects do more poorly the closer to the middle of the sequence they are.

- *Symbolic Distance Effect*: again, even compensating for the end anchor effect, the further apart on the series two items are, the faster and more accurately the subject makes the evaluation. This effect is seen as contradicting any step-wise chaining model of transitive inference, since there should be *more* steps and therefore a longer RT between distant items.

- *The First Item Congruity Effect*: in verbal tests, pairs are better evaluated if the question is posed in the same direction as the evidence (e.g. If the evidence takes the form “A is bigger than B”, “is C smaller than A” is harder to answer than “is C bigger than A”.) This is considered possibly analogous to the fact that pairs closer to the rewarded end of the series are done better than those symmetrically opposite (e.g. in 5 items with *A* rewarded, *AC* is easier than *CE*).

The currently dominant way to model transitive inference is in terms of the probability that an element will get chosen. Probability is determined by a weight associated with each element. During training, the weight is updated using standard reenforcement learning rules. Normalization of weights across rules has been demonstrated necessary in order to

[3]Unfortunately, the existing psychological literature is not consistent about whether *A* or *E* is the ‘higher’ (*rewarded*) end. This paper uses *A* as high (as in grade scales) which seems slightly more common.

get results invariant on the order of training (whether *AB* is presented first or *DE*). Taking into account some pair-specific context information has proven useful for accuracy, and necessary to explain the fact that animals can also learn a looping end pair (e.g. $E > A$ for 5 items). Wynne's current ultimate model takes the form

$$p(X|XY) = \frac{1}{1+e^{-\alpha(2r-1)}} \tag{9.1}$$

$$r = \frac{V(X)+V(Z)+\gamma V(\langle X|XY\rangle)}{V(X)+V(Y)+2V(Z)+\gamma V(\langle X|XY\rangle)+\gamma V(\langle X|XY\rangle)} \tag{9.2}$$

where Z is the normalizing term, $\langle a|ab\rangle$ indicates context sensitivity, and α and γ are free parameters. Such models assume that reaction time somehow co-varies with probability of correct response.

9.2.3 Models of Errors in Transitive Inference

In one of the earliest of the non-cognitive transitivity papers, McGonigle and Chalmers [1977] not only demonstrated animal learning of transitive inference, but also proposed a model to account for the errors the animals made. Monkeys, like children, tend to score only around 90% on the pair *BD*. McGonigle and Chalmers proposed the *binary-sampling theory*. This theory assumes that:

- The animals consider not only the items visible, but also from any items that might be *expected* to be visible. That is, they take into account elements normally associated with the presented stimuli, particularly elements that occur between stimuli.
- The animals consider only two of the possible elements, choosing the pair at random. If they were trained on the pair, they perform as trained; otherwise they perform at chance, unless one of the items is an end item (*A* or *E*), in which case it will perform appropriately (select or avoid respectively).
- If the animal 'selects' an item that is not actually present, it cannot act on that selection. Instead, this selection reenforces its consideration of that item, which tends to push the animal towards the higher valued of the items displayed.

Thus for the pair *BD*, this model assumes an equal chance the monkey will focus on *BC*, *CD*, or *BD*. Either established training pair results in the monkey selecting *B*, while the pair *BD* results in an even (therefore 17%) chance of either element being chosen. This predicts that the subjects would select *B* about 83% of the time, which is near to the average actual performance of 85%.

The binary-sampling theory is something of a naive probabilistic model: it incorporates the concept of expectation, but is not very systematic. It also fails to explain the symbolic distance effect. However, it motivated McGonigle and Chalmers to generate a fascinating data set showing the results of testing monkeys (and later children [Chalmers and McGonigle, 1984]) on *trigrams* of three items. The binary-sampling theory predicts that for the trigram *BCD* there is a 17% chance D will be chosen (half of the times *BD* is attended to), a 33% chance *C* will be chosen (all of the times *CD* is attended to) and a 50% chance of *B* (all of *BC* plus half of *BD*). In fact, the monkeys showed 3%, 36%, and 61%, respectively.

This data was used by Harris [1988] to create an even better model of the performance of the monkeys, both modeled as a group and as individuals. Harris attempted a production-rule model. This model involves only two assumptions.

1. The subject learns a set of rules of the nature "if A is present, select A" or "if D is present, avoid D".

2. The subject learns a prioritization of these rules.

This process results in a *rule stack*, where the first is applied if possible, if not, the second, and so on. Only four rules are needed to discriminate the five items. Further, there are only 8 discernible possible rule stacks a subject can learn which will solve the initial training task[4].

Adding the assumption that in the trigram test cases, an 'avoid' rule results in random selection between the two remaining items, the Harris and McGonigle [1994] results show no significant difference from the monkey results on conglomerate data. For example, over all possible trigrams, the stack hypothesis predicts a distribution of 0, 25% and 75% for the lowest, middle and highest items. Binary sampling predicts 3%, 35% and 63%, and logic of course 0%, 0% and 100%. The monkeys showed 1%, 22% and 78%. Further, individual performance of most monkeys could be matched to a particular stack.

9.2.4 The Proposed Model

The model of transitivity we will build is influenced by all three models discussed above. It is also influenced by the neurologically based models of action selection [Redgrave et al., 1999] and reenforcement learning [see Gillies and Arbuthnott, 2000, for a review] in the basal ganglia (see Chapter 11). The basal ganglia appears to control movement primarily through selective context-based inhibition of activity moving from perceptual and motivational streams towards motor cortices.

My model proposes the following:

1. In the earliest phase of training, the subjects learn to grasp any viewed tin in order to get their reward.

2. In the large blocks of ordered pair training, subjects discover that they may only select one item, and that only one is rewarded. Consequently, they must inhibit both applicable grasping behaviors until they have selected one item.

3. As the size of the training blocks is reduced, subjects must also learn prioritization between neighboring inhibition rules. The interactions of these neighbors is generally sufficient to produce a total ordering, although the compartmentalization also allows for learning the additional *EA* pair.

[4]There are actually 16 possible stacks, but trigram experiments cannot discriminate whether the fourth rule is a select or an avoid rule.

4. The process of selecting between two rules of similar activation involves probabilistic attentive sampling of the alternative rules. Attention allows for searching for increased evidence and generally increases the activation of the rule. Where two competing rules are similarly weighted, this process is likely to be both longer and more arbitrary.

In this chapter, I focus on modeling the third of these proposals. The first two are relatively uncontentious, and the fourth is essentially an elaboration of the common assumption of the sequence-learning models [e.g. Wynne, 1998].

9.3 Behavior-Oriented Design of a Model Agent

In this section, I document the construction of a performing agent-based version of the above model. This section is intended primarily as a detailed example of BOD. However, some of the intermediate versions of the model are represented in the results shown in Section 9.4.

Although this is a real modeling problem, the agent produced is relatively simple, particularly with respect to its drive system. The model is of a single task in a controlled environment, and is not real-time. To see more interesting examples of drive collections, see the agents in Sections 4.5.1, 7.6 or 10.5. The primary goal of this section is to illustrate in detail the BOD *process*, not the capabilities of BOD agent architectures.

Because Harris and McGonigle [1994] reverse the priority of the standard labeling letters A-E, I chose to revert to color labels to describe the boxes. This reduces the possibility for confusion based on assumptions about ordering. For all the following examples, the correct transitive ordering is this: $red > white > blue > green > yellow$.

9.3.1 Initial Decomposition

Here are the steps of BOD initial decomposition (see Section 8.2.1) applied to the task of modeling a monkey learning transitive inference. The agent built here actually represents two agents in the real world: the monkey and the test machine. This is not really as strange as it might seem, since in the real world the test apparatus must be designed in such a way that it accounts for the behavior of the monkey. BOD can also be applied to multi-agent systems (see Chapter 10), but in this case the behavior organization is better considered from the perspective of the system as a whole.

1. *High level description:* At the highest level of description, we want a system that on the one (apparatus) hand administers tests, and on the other (monkey) hand, makes choices and learns from them. In more detail, we want the agent to learn an ordering of selection rules for choosing one item when presented with pairs of stimuli, and if it learns to a particular criteria, then we want to test it with novel pairs and also triads of stimuli and see what it does.

2. *Prototype plans:* An individual trial should look something like this: set test, choose a box, be rewarded (if right) or notified (if wrong), learn from this experience, have

test-board cleared. At a higher level, the testing machine must follow a training procedure: learn each set of adjacent pairs to criteria, then demonstrate competence on each pair, in ordered sets following training, and then mixed randomly. If criteria is passed, then perform on all possible pairs (rewarding any behavior on the non-training pairs, correct behavior only on training pairs) and then on all possible triads (rewarding any behavior.)

3. *Prototype plan primitives:* The actions the system needs to be able to perform are setting the test, administering a reward, clearing the test, grasping a box and learning from an episode of test and reward. The monkey will need to be able to notice when a test has been set, distinguish between the boxes, and notice its learned preferences. The machine will need to be able to tell when to administer a test or a reward, and also what test or reward to administer.

4. *Prototype Behaviors* One obvious decomposition in this case is between the state needed by the monkey, and that needed by the apparatus. Further, some state on the monkey varies rapidly by context (e.g. where the monkey is looking or what it is holding) while other state adapts slowly over time (e.g. the priorities the monkey assigns its possible actions.) Since very different rates of state change are a key decomposition indicator, we will expect to have at least three behaviors: monkey (for the eyes and hands), sequence (for the priority learning), and apparatus (for the testing machine).

5. *Determine drives:* In this case, we are really only modeling a single high-level competence, although we will code it as a drive so that the system will run continuously while we are developing and debugging it.

6. *Select a first behavior to implement:* We will start by modeling the behavior in Harris system: a hard-coded reactive plan for transitive inference. This gives us a chance to debug our primitives, and to demonstrate the difference between encoding this knowledge and learning it.

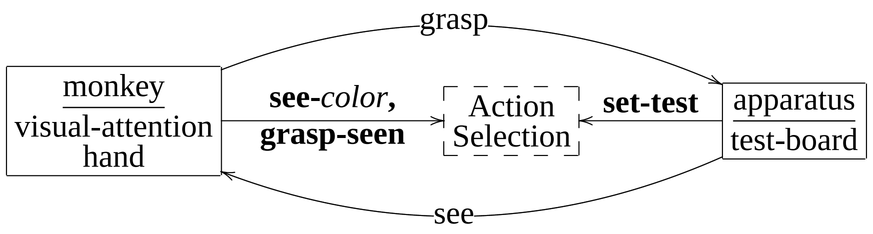
Once an initial decomposition exists, what remains is to iteratively build the agent. The following sections document my development of this agent. Each drawn model represents a plan script I debugged then saved as part of my test suite. A test suite of relatively simple plan scripts is useful for when major changes to the behavior library are made. Such changes could be caused by new insights about underlying representations, or the discovery of bugs. But most often, they are part of the development process — either the next added capability, or a new simplification of agent structure. All such changes are expected and supported in the BOD process.

9.3.2 Modeling an Expert

We begin by replicating a simplified version of Harris' system, which models skilled monkeys. We initially ignore the slightly more complicated avoid rules; we model the only correct solution using all select rules.

binary-test

The very first cut is just a competence (Figure 9-2). There are two behaviors that support it, representing the state in the monkey, and the state in the testing machine (**set-test** must be invoked by hand before running the competence.) (The monkey part of the agent is named Elvis after a particular Capuchin.) The primitive *set-test* must be called by hand before the competence can be tested.



(a) Behaviors

$$\textbf{elvis} \Rightarrow \left\langle \begin{array}{r} \text{(see-red)} \Rightarrow \text{grasp-seen} \\ \text{(see-white)} \Rightarrow \text{grasp-seen} \\ \text{(see-blue)} \Rightarrow \text{grasp-seen} \\ \text{(see-green)} \Rightarrow \text{grasp-seen} \end{array} \right\rangle \tag{9.3}$$

(b) Plan

Figure 9-2: The plan *binary-test*. This models only a simple, trained competence for making transitive choice. Subsequent diagrams omit the ‘grasp’ and ‘see’ arcs for clarity.

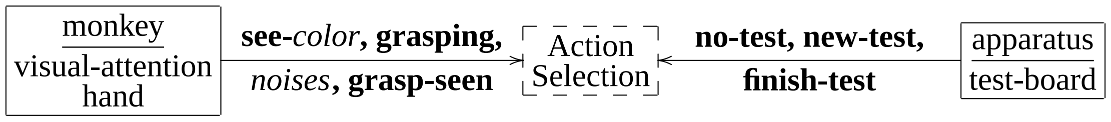
The competence here works effectively like a switch statement in C or a cond in lisp (or a production stack in Harris’ production-rule system.) The monkey grasps the highest priority color it sees. The **see-***color* primitives all map to a single method with a particular perceptual argument. The perceptual primitive also sets the visual attention, which is then used by **grasp-seen** to determine which object is grasped. Grasp-seen has the side-effect of removing the grasped object from the test-tray of the apparatus.

driven-b-test

The next step is to incorporate the elvis competence into an agent. The agent in Figure 9-3 sets a test if there’s currently no test, removes the test if the monkey is currently grasping something, and lets the monkey make its choice otherwise.

This plan gives an example of each POSH element type, though the action pattern is mostly gratuitous[5]. Notice that the drive collection, life, has no goal, so it never ends. The

[5] I used screeching for very high-level debugging — when this agent operates with all tool-wise debugging turned off, the only indication it is running successfully is the occasional screech at a red box.



(a) Behaviors

$$
\mathit{life} \Rightarrow \left\langle\!\!\left\langle \begin{array}{r l} (\text{no-test}) \Rightarrow & \text{new-test} \\ (\text{grasping}) \Rightarrow & \text{finish-test} \\ (\text{no-test } \perp) \Rightarrow & \textbf{elvis-choice} \\ \Rightarrow & \text{hoot} \end{array} \right\rangle\!\!\right\rangle \tag{9.4}
$$

$$
\textbf{elvis-choice} \Rightarrow \left\langle \begin{array}{r l} (\text{see-red}) \Rightarrow & \textbf{noisy-grasp} \\ (\text{see-white}) \Rightarrow & \text{grasp-seen} \\ (\text{see-blue}) \Rightarrow & \text{grasp-seen} \\ (\text{see-green}) \Rightarrow & \text{grasp-seen} \end{array} \right\rangle
$$

$$
\textbf{noisy-grasp} \Rightarrow \langle \text{screech} \rightarrow \text{grasp-seen} \rangle
$$

(b) Plan

Figure 9-3: The plan *driven-b-test*. This models an experienced monkey.

bottom-most drive should never trigger, it is included for debugging purposes. Screeching and hooting are nominally ascribed to the monkey behavior, though they require no state — they produce formatted output that alerts the experimenter.

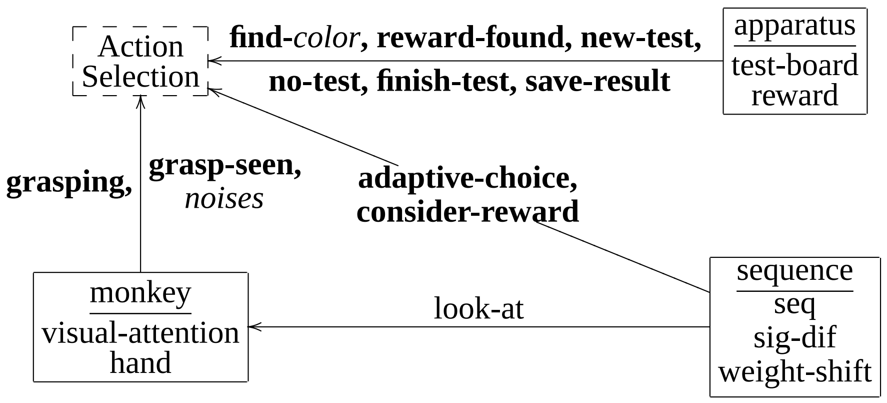
9.3.3 Learning an Ordering

prior-learn

The next step is to enhance the model by forcing the monkey to learn the ordering of the colors. This requires two significant changes:

- augmenting the test-behavior to reward the monkey appropriately, and
- adding a new behavior, sequence, to the system.

As external observers we would of course ascribe sequence to the monkey. However, notice that this is a separate object / behavior from the previous monkey behavior. This is in keeping with the BOD principle that behavior decomposition is dictated by representation and the rate of adaptation. Monkey's state is deictic and changes utterly on every testing episode. Sequence represents life-long learning. Its primary adaptive state is a vector, and its learning rule is akin to standard neural-network weight update rules. The other two associated pieces of state are parameters for this learning rule, which are fixed for each individual monkey.



(a) Behaviors

$$
\textit{life} \Rightarrow \left\langle\left\langle \begin{array}{r@{\;}l} \text{(no-test)} \Rightarrow & \text{new-test} \\ \text{(rewarded)} \Rightarrow & \textbf{end-of-test} \\ \text{(grasping)} \Rightarrow & \textbf{reward-monkey} \\ \text{(no-test } \perp) \Rightarrow & \textbf{educated-grasp} \end{array} \right\rangle\right\rangle \tag{9.5}
$$
$$
\Rightarrow \text{hoot}
$$
$$
\textbf{reward-monkey} \Rightarrow \left\langle \begin{array}{r@{\;}l} \text{(find-red)} \Rightarrow & \text{reward-found} \\ \text{(find-white)} \Rightarrow & \text{reward-found} \\ \text{(find-blue)} \Rightarrow & \text{reward-found} \\ \text{(find-green)} \Rightarrow & \text{reward-found} \end{array} \right\rangle
$$
$$
\textbf{educated-grasp} \Rightarrow \langle \text{adaptive-choice} \rightarrow \text{grasp-seen} \rangle
$$
$$
\textbf{end-of-test} \Rightarrow \langle \text{consider-reward} \rightarrow \text{save-result} \rightarrow \text{finish-test} \rangle
$$

(b) Plan

Figure 9-4: The plan *prior-learn*. Learning the priorities for different colors.

The apparatus uses an adapted version of the already-debugged transitive competence to determine whether the monkey has chosen correctly. Apparatus now requires extra perceptual skills in order to notice which color box was selected. The apparatus also has a new piece of state, 'reward', which is either nil, a peanut, or no peanut. (For the real monkeys, "no peanut" is an audible buzzer as well as a lack of reward following the choice.)

Figure 9-4 illustrates an example of trading off complexity in the behavior for complexity in the plan (see Section 6.5). 'Educated-grasp' is now a simple action pattern, which relies on sequence to determine which item to look at. Here the motivation for the tradeoff is not only simplification, but also the primary design goal for the agent: that it should learn appropriate prioritization. It is important to realize that although 'reward-monkey' looks like 'elvis-choice', it is in fact an entirely new competence in terms of ability for the system, if not in actual code.

By the principle of reducing redundancy (Section 8.3.1) competence 'elvis-choice' begged simplification. Of course, so does 'reward-monkey'. On the other hand, the arbitrary ordering of colors needs to be specified somewhere. It as easy specify this in the plan which is readily edited (if we ever wanted a different sequence), and, in this case, already debugged.

The sequence-learner contains three bits of state: a list of known objects with weights, a 'significant difference' and a 'weight shift'. The learning algorithm works like this:

- If the monkey chooses correctly, but its certainty was less than *significant-difference*, then it adds *weight-shift* to the weight of the winner, then renormalizes all the weights.
- If it is wrong, it shifts the weight in favor of the alternative, regardless of its certainty.

The sum of all the weights in a sequence is kept at 1.0; this is a standard strategy of modeling resource-constrained learning. If the monkey sees a new item it has never seen before, it adds this to the sequence list, giving it a weight equal to $1/N$ where N is the number of items currently in the list, reflecting a lack of expectation about where in the sort order the item will occur. This weight is derived from the other elements in proportion to their current weight (that is, each existing weight is multiplied by $(N-1)/N$).

The test machine is now in charge of both setting and rewarding the behavior. The new primitive 'find' searches the world for a colored box, then if it is found, a reward (or lack of reward) is given based on whether the machine is attending to the monkey's hand or the test-board. The end-of-test action-pattern calls actions in sequence from two different behaviors — sequence learns from the reward, then apparatus records the result and clears the test board.

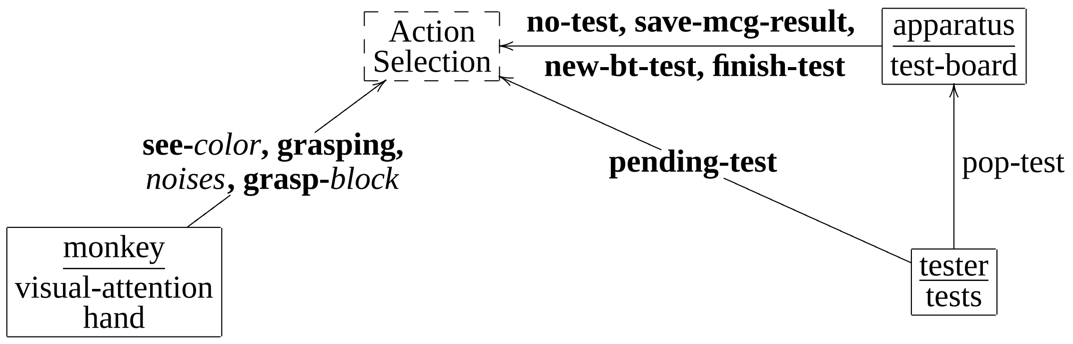
Results for the learning systems in this section are shown and discussed in Section 9.4 below. See Figure 9-8(a) for this particular plan. In this section, I will continue concentrating on demonstrating scaling under BOD.

fair-prior-learn

The figures I have been using to illustrate BOD systems do not show every aspect of a model's complexity. In particular, they do not show code for the behavior methods, but only their names. When scaling a model such as this one, some changes will only be in terms of method code. For example, the script **fair-prior-learn** only switches from training the agent on all possible pairs to only training it on adjacent pairs, in keeping with the standard animal procedure described in Section 9.2.1. The only impact on the plans in Figure 9-4 is to change 'new-test' to 'new-training-set'. Nevertheless, this was potentially a significant change to the training regime, so I ran another cluster of tests. With the monkey at this level of complexity, the performance was effectively identical to being trained on every possible pair.

9.3.4 Learning an Ordering of Rules

The full experimental results in the McGonigle and Chalmers [1977] transitivity tests indicate that both monkeys and 5-year-old children [Chalmers and McGonigle, 1984] learn



(a) Behaviors

$$life \Rightarrow \left\langle\!\left\langle \begin{array}{r} \text{(no-test) (pending-test } \perp\text{) hoot} \Rightarrow \textit{goal} \\ \text{(no-test)} \Rightarrow \text{new-bt-test} \\ \text{(grasping)} \Rightarrow \text{record-finish-test} \\ \Rightarrow \textbf{roger-choice} \end{array} \right\rangle\!\right\rangle \qquad (9.6)$$

$$\textbf{roger-choice} \Rightarrow \left\langle \begin{array}{r} \text{(see-red)} \Rightarrow \text{grasp-seen} \\ \text{(see-yellow)} \Rightarrow \text{grasp-other} \\ \text{(see-white)} \Rightarrow \text{grasp-seen} \\ \text{(see-blue)} \Rightarrow \text{grasp-seen} \end{array} \right\rangle$$

$$\textbf{record-finish-test} \Rightarrow \langle \text{save-mcg-result} \rightarrow \text{finish-test} \rangle$$

(b) Plan

Figure 9-5: The plan *roger-test*. This models a monkey with an avoid rule, and also an apparatus with a test regime.

not sequences of colors, but sequences of *behavior rules*. This fact is exposed when the primate or child is exposed to a choice of three colored boxes. As explained above, Harris and McGonigle [1994] showed a tight fit to a model where the subjects sometimes learn a rule like *avoid yellow* rather than *select red*. This fit assumes that if the highest ranking applicable rule is *avoid yellow*, the selection between the two other choices (in a triad) is at chance.

roger-test

The first step towards modeling learning rule rather than color ordering is modeling *having* such rules. Thus I returned to an earlier script, driven-b-test, and update it to represent a monkey that uses *avoid* rules. The monkey modeled happens to be a squirrel monkey named Roger.

The main monkey difference between roger-test and driven-b-test (Plan 9.5) is that the competence elvis-choice is replaced by the one in Figure 9-5, and the behavior monkey now supports the primitive ‘grasp-other’, as described above. However, in order to fully test the behavior, we also need to expose the monkey to triads. Thus I added a new behavior to apparatus, tester, which governs a regiment of training followed by testing. The first implementation is fairly simple — tester has a list of possible tests, and when that list is empty 'pending-test' is false. Notice *life* now has a goal, and terminates when the test is complete.

rule-learn

The final system will combine roger-test (Plan 9.7) with **fair-prior-learn** (effectively Plan 9.6). However, both to make debugging simpler and to more fully explore the model as a representation of a real monkey, we only add one behavior to a system at a time. Consequently, **rule-learn** does not incorporate tester from roger-test, but is instead is dedicated solely to learning orderings of rules.

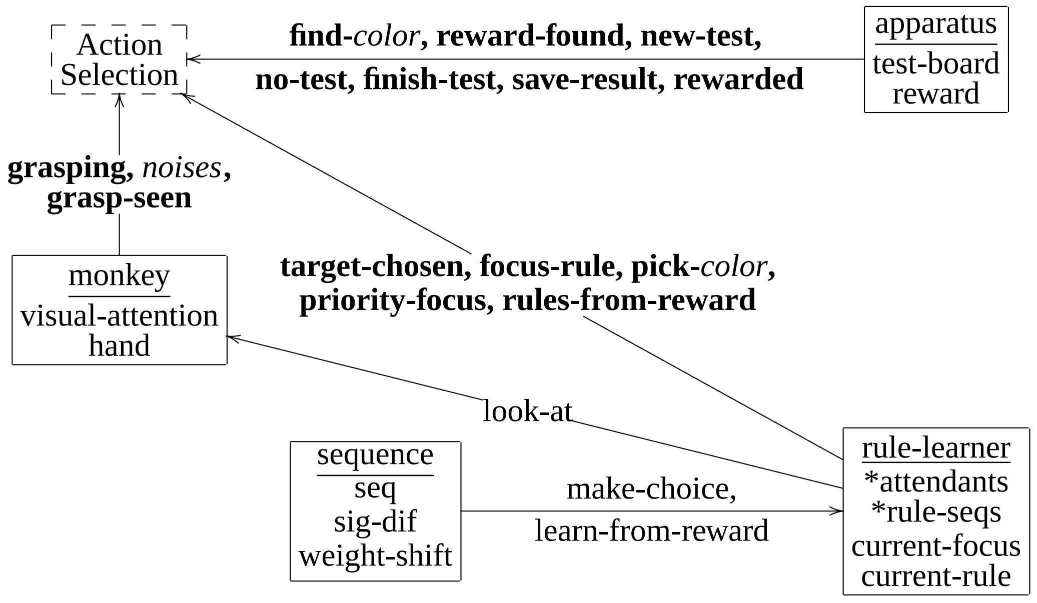
In the model in Figure 9-6, I assume (based on the results of Harris and McGonigle [1994]) that the learning of the select and avoid rule is independent for each possible context. That is, the monkey learns both which visual stimuli (colored box) is most important, *and*, for that stimuli, which rule is appropriate. This is consistent with the fact that some monkeys learn behavior that looks like "Select the 1st, Avoid the 5th, Select the 2nd", where the earliest mentioned rule has the highest priority[6].

There are two major embellishments in **rule-learn**. The first is a new behavior, rule-learner, which has four pieces of named state. Two are references to sequences, and two are deictic state for referencing the winning units in those sequences. In fact, 'rule-seqs' is really a list of sequences, one for each element in 'attendants'. The elements in 'attendants' correspond to the visual contexts, and are acquired the same way as the colors were acquired in **prior-learn**. I assume that the gross behaviors *select* and *avoid* have been acquired previously and are generally associated with colors. Thus, every time a new element is added to 'attendants', a new rule sequence is added as well, with the two rules preset to equal priorities (see also Section 9.2.4). Notice that there are now multiple instances of sequence behaviors. Each has its own distinct variable state, and is referenced differently, but has the same primitives. (See further the Section 7.6.2 commentary on directions.)

The second change is that 'educated-grasp' is now again a competence rather than an action pattern. This is not strictly necessary — 'pick-this' and 'pick-other' might have been simply 'pick' with reference to a deictic variable. However, given the large difference in the two procedures, I felt that segregating them improved the agent's clarity.

Other differences are somewhat less visible. For example, 'rules-from-reward' now affects two sequences, not one. This is implied by the fact it is now a method on rule-learner. In **rule-learn**, when making a choice, the monkey uses the rule-learner attendants' sequence to determine which color box it attends to, then the rule-seq associated with that color to determine which rule it applies. Learn-from-reward applies the learning rule to both sequences.

[6]As I explained earlier, the monkeys must actually learn four rules to disambiguate a five-item sequence, but the fourth rule cannot be discriminated using triad testing.



(a) Behaviors

$$\mathit{life} \Rightarrow \left\langle\left\langle \begin{array}{rl} (\text{no-test}) \Rightarrow & \text{new-test} \\ (\text{rewarded}) \Rightarrow & \textbf{end-of-test} \\ (\text{grasping}) \Rightarrow & \textbf{reward-monkey} \\ \Rightarrow & \textbf{educated-grasp} \end{array} \right\rangle\right\rangle \tag{9.7}$$

$$\textbf{reward-monkey} \Rightarrow \left\langle \begin{array}{rl} (\text{find-red}) \Rightarrow & \text{reward-found} \\ (\text{find-white}) \Rightarrow & \text{reward-found} \\ (\text{find-blue}) \Rightarrow & \text{reward-found} \\ (\text{find-green}) \Rightarrow & \text{reward-found} \end{array} \right\rangle$$

$$\textbf{educated-grasp} \Rightarrow \left\langle \begin{array}{rl} (\text{target-chosen}) \Rightarrow & \text{grasp-seen} \\ (\text{focus-rule 'avoid}) \Rightarrow & \text{pick-other} \\ (\text{focus-rule 'select}) \Rightarrow & \text{pick-this} \\ \Rightarrow & \text{priority-focus} \end{array} \right\rangle$$

$$\textbf{end-of-test} \Rightarrow \langle \text{rules-from-reward} \rightarrow \text{save-rule-result} \rightarrow \text{finish-test} \rangle$$

(b) Plan

Figure 9-6: The plan *rule-learn*. This learns the priorities of rules for different colors, as well as learning an ordering between these rules.

9.3.5 Real Training Regimes

As is shown in Section 9.4 below, **rule-learn**, unlike **prior-learn**, does not always converge when exposed only to randomly ordered adjacent pairs in the sequence. To complete our model, we must subject the monkey from **rule-learn** to a full training regime. I used the regime specified by Chalmers and McGonigle [1984] because of the detail in the description.

educate-monkey

Adding the mechanism for a training regime into a BOD agent is not strictly necessary. As was explained in Section 2.2.5 (see also Section 12.2), conventional software systems can be incorporated directly into an intelligent BOD agent. The training system, more algorithmic than 'intelligent', seems a prime candidate. Consequently, the new competence 'pick-test' grossly oversimplifies the complexity of the regime — there are actually nine phases of training and testing. The distinction between an (adjacent) pair and an n-gram (bigram or trigram) I document with separate primitives simply because it also documents the movement from training to testing.

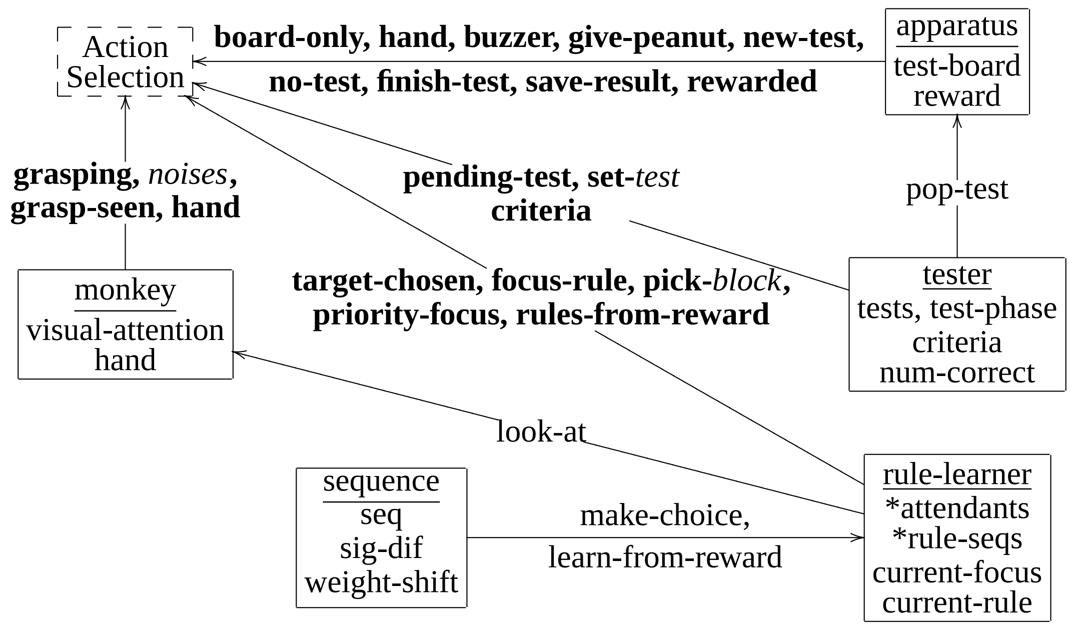
McGonigle and Chalmers [1977] also found it necessary to continue at least partially training the animals during testing. If a monkey is confronted by one of the adjacent pairs it has been trained on, and it performs incorrectly, then it is not rewarded. On all other test stimuli (all triads, and any non-adjacent pairs), the monkey is rewarded regardless of its choice. This explains why I needed to alter the reward competence for **educate-monkey**.

The results shown below were actually based on a script with one further refinement — it computes whether the monkey was *right* as well as whether it should be rewarded in order to simplify the analysis. Since there is little additional theoretical or pedagogical value in this improvement, the script (**educate-me+monk**) is included only in Appendix B.

9.4 Results

The previous section described the entire process of developing a BOD-agent model for learning transitive inference. Although the main motivation for going into detail on seven versions of the model was to illustrate the BOD process, several of the models provided interesting results. In this section I will not discuss the results of **binary-test**, **driven-b-test** or **roger-test**, because these non-learning models only replicated the work of Harris and McGonigle [1994]. While checking the replication was an essential part of development, it is not particularly interesting, since the operation of production-rule systems is well understood.

Similarly, **prior-learn** and **fair-prior-learn** were essentially replications of the studies documented by Wynne [1998], only with less precise mathematical models. However, these learning systems provide a basis for comparison for the final two models, so I begin by reviewing these systems.



(a) Behaviors

$$\begin{array}{r}
\text{(test-done) clean-up hoot} \Rightarrow \textit{goal} \\
\textit{life} \Rightarrow \left\langle\left\langle \begin{array}{r}
\text{(no-test)} \Rightarrow \textbf{pick-test} \\
\text{(rewarded)} \Rightarrow \textbf{end-of-test} \\
\text{(grasping)} \Rightarrow \textbf{selective-reward}
\end{array}\right\rangle\right\rangle \\
\Rightarrow \textbf{educated-grasp}
\end{array} \tag{9.8}$$

$$\textbf{pick-test} \Rightarrow \left\langle \begin{array}{r}
\text{(no-test } \perp) \Rightarrow \textit{goal} \\
\text{(criteria '3)} \Rightarrow \text{set-ngram} \\
\Rightarrow \text{set-pair}
\end{array}\right\rangle$$

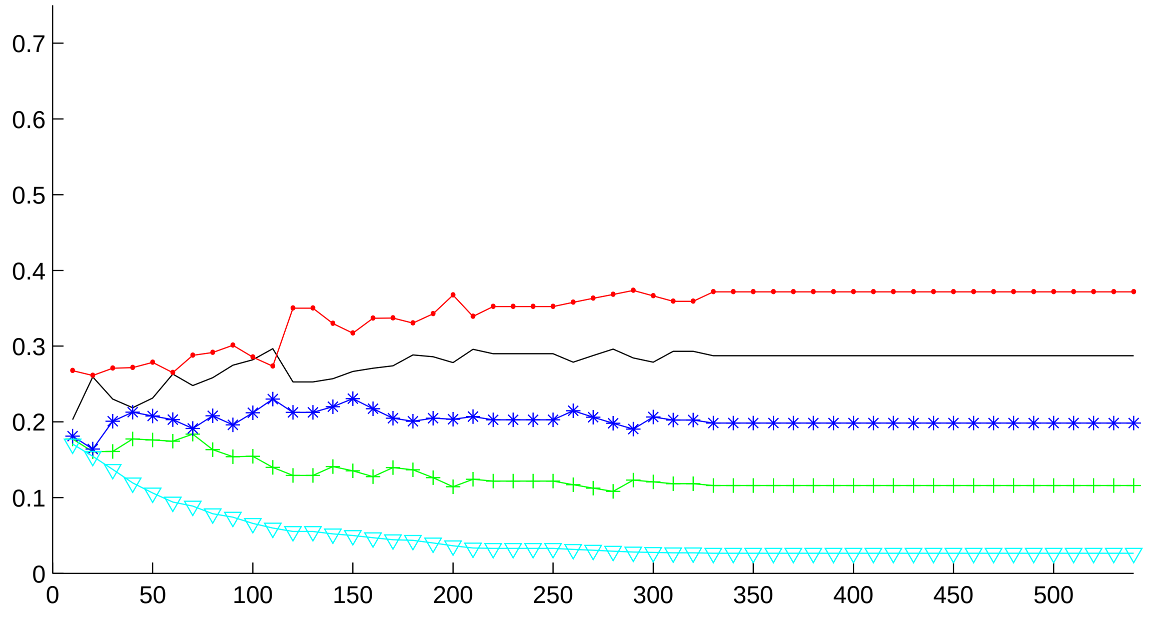
$$\textbf{selective-reward} \Rightarrow \left\langle \begin{array}{r}
\text{(board-only 'red) (hand 'white)} \Rightarrow \text{buzzer} \\
\text{(board-only 'white) (hand 'blue)} \Rightarrow \text{buzzer} \\
\text{(board-only 'blue) (hand 'green)} \Rightarrow \text{buzzer} \\
\text{(board-only 'green) (hand 'yellow)} \Rightarrow \text{buzzer} \\
\Rightarrow \text{give-peanut}
\end{array}\right\rangle$$

$$\textbf{educated-grasp} \Rightarrow \left\langle \begin{array}{r}
\text{(target-chosen)} \Rightarrow \text{grasp-seen} \\
\text{(focus-rule 'avoid)} \Rightarrow \text{pick-other} \\
\text{(focus-rule 'select)} \Rightarrow \text{pick-this} \\
\Rightarrow \text{priority-focus}
\end{array}\right\rangle$$

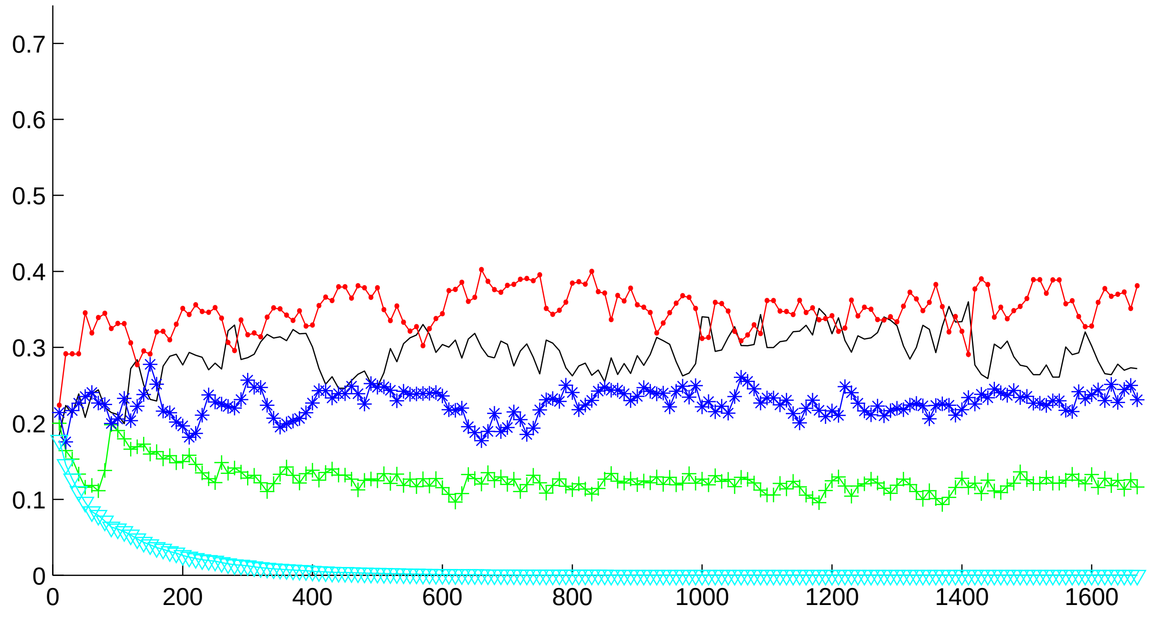
$$\textbf{end-of-test} \Rightarrow \langle \text{rules-from-reward} \rightarrow \text{save-rule-result} \rightarrow \text{finish-test} \rangle$$

(b) Plan

Figure 9-7: The plan *educate-monkey*. This augments *rule-learn* by making the training apparatus more sophisticated.



(a) **fast-prior-learn** finds a stable solution rapidly if one exists. Here ‘sig-dif’ was .08, and ‘weight-shift’ .02. The dotted line represents red (1st), the plain line is white (2nd), stars are blue (3rd), pluses are green (4th), and triangles yellow (5th).



(b) If there is no stable solution, then chance orderings of tests can drive one set of weights above another. Here **prior-learn** is running with ‘sig-dif’ at .12, (‘weight-shift’ is still .02).

Figure 9-8: Results from *prior-learn.*

9.4.1 Learning Colors, Not Rules

Figure 9-8(a) shows a typical result for **fair-prior-learn**. In either form of **prior-learn**, given that the weights sum to one, if the significant-difference set to .08 or less then a stable solution is found. In fact, although I gave **fair-prior-learn** its name because it was only exposed to training and not testing pairs, it actually trained more reliably and stabilized its weights much quicker than **prior-learn** (see Figure 9-9). This is no doubt due to the high incidence of significant (boundary) information provided in the training sets.

	prior-learn		*fair-prior-learn*	
	last err.	weights stbl.	last err.	weights stbl.
1	180	220	92	140
2	200	> 278	96	130
3	35	280	90	150
4	190	320	93	140
5	35	> 326	88	150

Figure 9-9: **prior-learn** vs. **fair-prior-learn**. Weights are only reported every 10 trials. Two trials ended (arbitrarily) before weights stabilized.

On the other hand, if the 'sig-dif' is greater than .1, then a stable solution for five items cannot be reached. Unless learning tails off, a 'hot hands'-like phenomena causes the weight of an item that has recently occurred in a number of training pairs to grow higher than the next-ranking element (see Figure 9-8(b)). In systems without stable solutions, this happens regularly.

These results give us a hypothetical explanation for individual difference in transitive task performance. Individual differences in stable discriminations between priorities can affect the number of items that can be reliably ordered. On the other hand, for fair-prior-learn, competing / unstable weights often represent the two top-priority colors. This violates the End Anchor Effect, so is not biologically plausible Of course, given that the bottom-most priority item successfully buries itself in a stable weight, one could imagine a dual-weighting system (such as proposed by Henson [1998]) that would anchor both ends.

9.4.2 Rule-Learning Results

With the added complication of rule learning, the monkey model can no longer learn the correct solution. In fact, **rule-learn** consistently learns either the solution shown in Figure 9-10 or a symmetric one with the 2^{nd} and 3^{rd} rules fighting for top priority. Although the priority between stimuli is not stable, the priority for rules *does* quickly stabilize. These solutions, although they look truly pathological, only make a mistake in one training pair — that of the two top priority stimuli. The ordering of these two rules is irrelevant, the error is made whichever rule fires.

Notice that this result *does* display the End Anchor Effect. The simulated monkeys quickly choose rules which avoid making errors on the two end points, but they choose

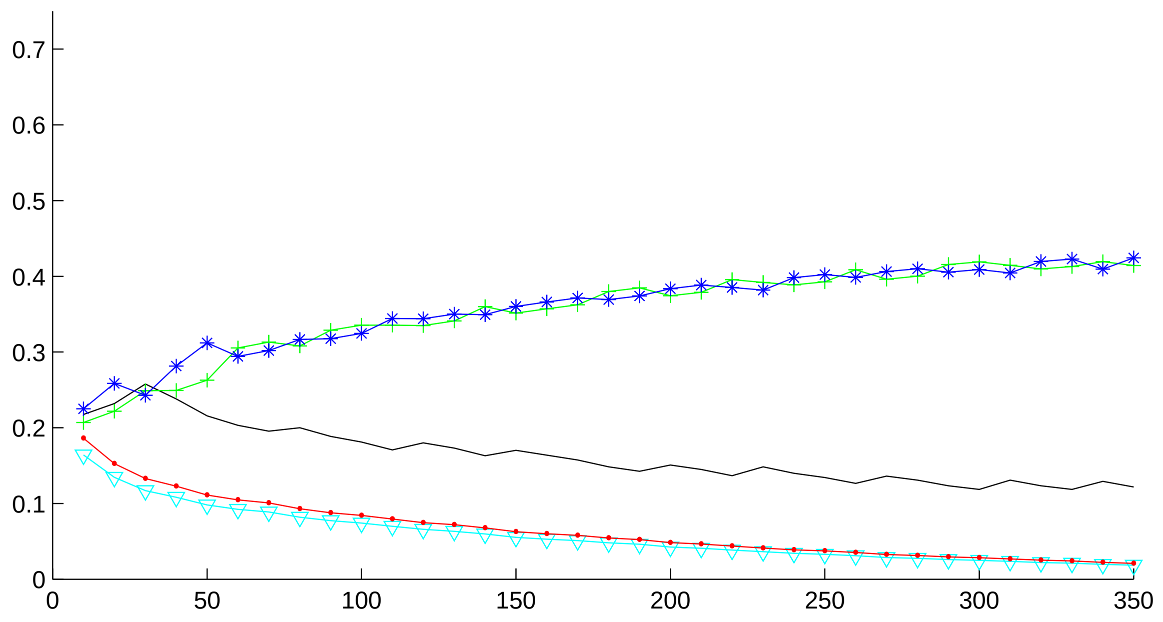


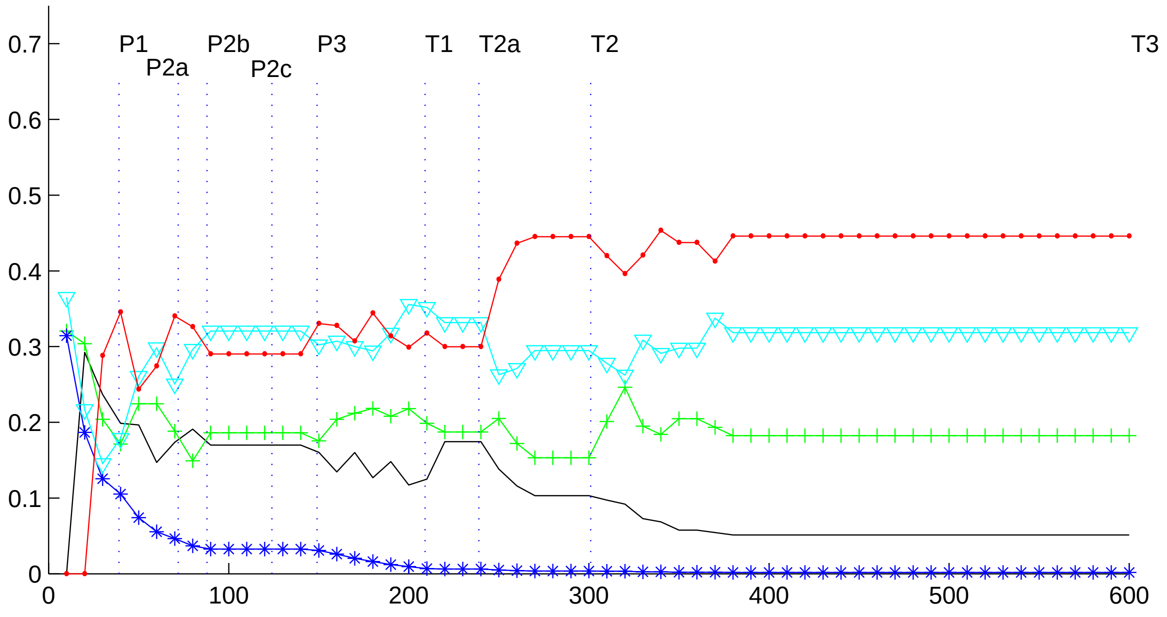
Figure 9-10: rule-learn fails to find a stable solution. The figure does not show the rule associated with each stimuli: they are *select* for green/pluses (4th), *avoid* for blue/stars (3rd), and *avoid* for white/no mark (2nd). This agent confuses only one training pair, blue/green.

rules which do not lead to complete solutions. It also displays the Serial Position Effect – it confuses only central pairs. Further, a real monkey named Blue showing a consistent error between the 3rd and 4th item is shown by Harris and McGonigle [1994, p. 332]. Perhaps Figure 9-10 is a model of Blue's mind.

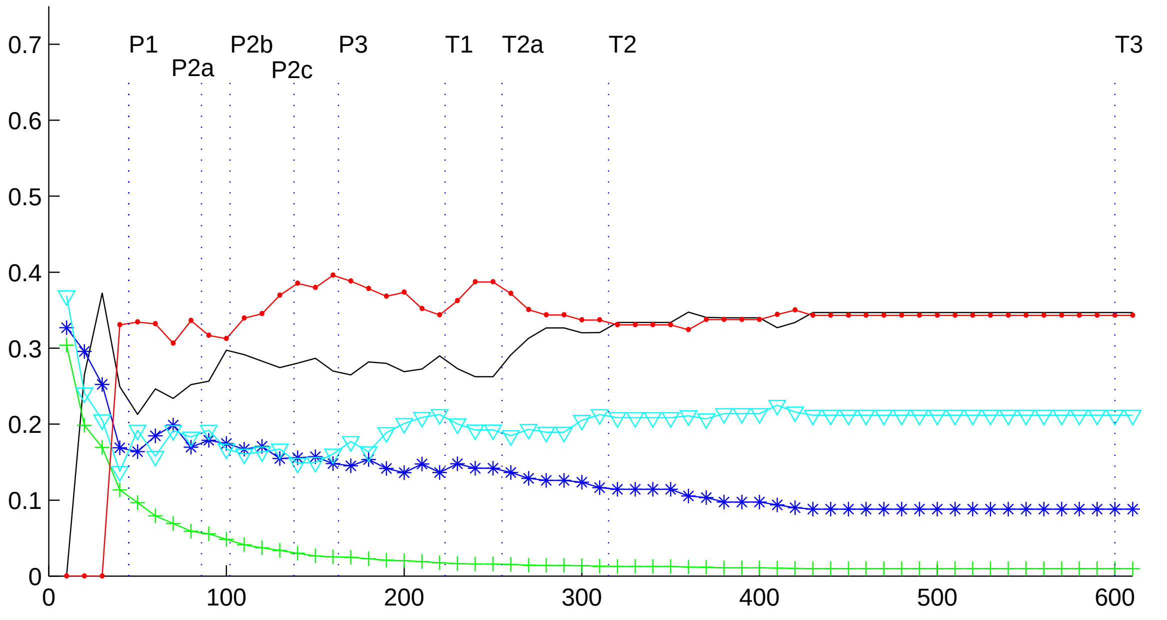
The training regiment implemented in **educate-monkey** (see Figure 9-1) was actually a more rigorous procedure than was applied to the monkeys, because children have more difficulty learning this task than adult monkeys do. It also is more rigorous than necessary for my simulated monkeys. Nearly all of them converge quickly, and the ones that fail to learn fail early, usually during Phase 2a.

Figure 9-11(a) shows an interesting run of **educate-monkey**, where, prior to testing, two rules are not fully ordered. However, since the rules concern the two opposite ends of the inference chain, either ordering is actually correct.

Figure 9-11(a) also shows how rule use can provide for stable solutions within resource constraints that would be too tight for stability in **prior-learn**. Because some rules would never normally compete with each other, they can share priority space, producing solutions such as those shown during phases P2c and most of P3, or the end of the third portion of training. Figure 9-11(b), where the sig-dif is also .12, also shows two rules that do not interfere with each other share a priority level until testing. However, after trigram testing, pressures similar to those in Figure 9-8(b) coupled with the lack of negative reinforcement result in a stable weight configuration being found that is no longer as correct. Of course, in trigram testing, all selections are rewarded, so from the agent's perspective, the solution is satisfactory.



(a) With the training regime of *educate-monkey* in place, most runs are able to find a correct solution. Vertical lines and labels mark the *end* of the labeled phase. This agent shows further learning occurring during test phases T1 and T2. It learns to *select* red/dots (1^{st}), *avoid* yellow/triangles (5^{th}), and *avoid* green/pluses (4^{th}). This monkey was particularly stupid: sig-dif .12, weight-shift .06.



(b) Another *educate-monkey* agent. This agent learned to *select* red/dots (1^{st}), *select* white/plain (2^{nd}), and either *avoid* yellow/triangles (5^{th}) or *select* blue/stars (3^{rd}). This monkey also had sig-dif .12, but weight-shift was .02.

Figure 9-11: Results from *educate-monkey*.

9.5 Conclusions and Discussion

The primary emphasis of this chapter is illustrating the BOD development process with an extended example. The flow of design is shown from a system of one competence and two simple behaviors, to a POSH action-selection tree with a drive collection, three competences and an action pattern, arbitrating for six behaviors, several of which are adaptive. I have also presented the first learning model of the rule-learning hypothesis proposed by Harris and McGonigle [1994], and demonstrated a number of interesting agents which show at least qualitative resemblance to various individual monkeys, and thus supply hypotheses about the learning process and beliefs underlying the monkeys' behavior.

The results section contains only the beginning of an analysis for this system, and further, the agent itself could continue to be scaled. To fully implement the model proposed in Section 9.2.4, I would need to alter 'priority-focus' to be more probabilistic when the model is uncertain. To make the model more biologically plausible, the basic priority representation probably requires two weight vectors instead of one, as suggested in Section 9.4.1. Making either of these improvements is fairly trivial: testing the agents and the analysis of the results is the most time consuming part of the process. (In fact, a batch tester added on top of tester and apparatus should probably be the next stage of my development!)

My model of the transitive-inference task is particularly interesting with respect to this dissertation. It extends the discussion of tradeoffs between plans and behaviors begun in Section 6.5, by showing how an adaptive behavior (rule-learner) can learn what is essentially a BRP. At the same time, the difficulty for both children and monkeys in learning this task underscores my emphasis on the importance of bias in learning, and of design in AI.

Chapter 10

Another Example: Modeling Social Interactions in Primate Colonies

10.1 Introduction

In Chapter 9, I gave an extended review of a BOD project, though of a relatively simple agent system. Although the primate competence being modeled was significant and interesting, the simulation itself was not in real-time. Also, although I was modeling two real-world agents (a monkey and a machine) I built only one BOD agent.

This chapter is the reverse: it is a very brief introduction to some preliminary work being done on a very interesting complex agent domain. Although the model is still under development, I include its current state here, because it demonstrates several important features not shown elsewhere in this dissertation:

- The use of BOD in a multi-agent context.

- A real-time drive system implemented under the newer POSH implementation (see Section 4.6.3). I have already demonstrated the earlier real-time POSH control on a robot (in Chapter 7). However, since this chapter uses the same implementation as in Chapter 9, the code is more readily comparable (see then Appendix.)

- The conventional level-based modeling of emotional motivations. Existing models by other researchers along these lines were already reviewed in Section 8.3.3. This chapter demonstrates how to incorporate this important idiom of the agent literature into BOD.

10.2 Biological Background: de Waal's Theory of Primate Societies

The work in this chapter is being done in collaboration with Jessica Flack of Emory University. Here is her summary of the phenomena we are modeling:

> One of the most interesting questions in the study of animal societies is how individuals negotiate their social relationships. This question of how conflict among lower level units (individual group members) is regulated in the formation of higher level units (societies) has been described as the fundamental problem in ethology, [Leigh, 1999]. Although research on non-human primate societies indicates that there are a variety of mechanisms — such as aggression, social tolerance, and avoidance — by which conflict is managed or resolved [de Waal, 2000], it is not well understood how and why the expression of these mechanisms varies across and even within social systems. For example, there is tremendous variation across the macaque genus in terms of how conflict is managed despite similar patterns of social organization. Aggression in some species is common and severe while in others, it is extremely frequent but rarely escalates to levels that produce injuries [de Waal and Luttrell, 1989]. Corresponding to this variation in the degree to which aggression is employed to settle conflicts of interest is variation in the degree of social tolerance by dominant individuals of subordinate ones, particularly in the context of resource acquisition, and variation in the degree to which relationships damaged by aggression are repaired via reconciliation [de Waal and Luttrell, 1989]. Although it appears that this co-variation in conflict management mechanisms varies in predictable ways across species, it does not appear that the co-variation is species-specific. Rather, the variation seems to be emergent from patterns of social interaction among individuals, and self-reinforced through social learning.
>
> [Bryson and Flack, 2001]

10.3 Extending BOD for a Multi-Agent Simulation

The first significant technical challenge to this work has been creating a multi-agent implementation of a BOD system. Of course, this would be easy enough to do with multiple fully independent agents, but for the reasons described in Section 8.4.3, I wanted to run all of the BOD agents under a commercial debugging environment on a single machine.

Fortunately, the current version of the POSH architecture was already implemented in a version of common lisp that supports multiple processes [Xan, 1999]. The approach I took was to dedicate one process to each agent, and rely on the platform to perform the time sharing between agents. Each agent has its own instances of both its behaviors and its current control stack (the action scheduler). Currently they share the same POSH reactive plan, but we intend to explore heterogenous communities in the future. However, since the action-schedule holds current instances of the POSH control objects in order to keep track of control state, there is currently no reason to replicate the template POSH plan structure for each agent.

10.4 Using BOD to Model Drives and Emotions

10.4.1 Modeling Conventional Emotion and Drive Theory

The topic of emotions is losing its taboo both in artificial intelligence and in the animal sciences. Nevertheless, emotions seem necessarily an emotional subject, often raising unusually passionate responses, both in support and in criticism for systems and theories. The primary goal of the present model is to explore models of state and behavior underlying the complex social activity in non-human primates. As a byproduct of this work, we must also integrate emotional responses into complex agent control.

By all indications, the various phenomena we know as emotions characterize a set of behaviors that evolved at significantly different points in our ancestral history [LeDoux, 1996]. Emotions are effectively an intervening variable used to explain categories of species-typical behaviors that are related not only by the behaviors and the environmental contexts in which they tend to be displayed, but by expressive body postures in the behaving animal. These emotion “variables” have a real, biological correlate: relatively slow and diffuse chemical transmissions within the brain (and the rest of the organism) which create a significant internal context for the brain’s operation, affecting both perception and action selection.

Our agents are designed to model animal behaviors that humans readily describe as emotional — the interactions between individuals in a primate colony. Our current model shows the animals oscillating between two “drives”, the desire to groom, and the desire for privacy. Grooming is an interesting behavior, associated with bonding between animals and a calming effect on the recipient. Although most primates seem to derive pleasure from grooming, they normally engage in this behavior relatively infrequently. Frequency of grooming tends to increase in times of certain social stresses.

The ‘desire for privacy’ in this model stands in for a number of other ways primates spend their time, such as foraging and napping. We model only seeking isolation from other agents for simplicity’s sake. For monkeys living in a community, one monkey’s desire to groom can interfere with another’s desire for privacy. There are a number of possible solutions to such conflict [de Waal, 2000], but for our pilot study we are only modeling two: *tolerance* and *aggression*. Aggression is of course associated with two other emotions, anger and fear.

10.4.2 Representing Emotions under BOD

The emotional responses of the agents in our simulation are represented exactly as any other behavior — through a combination of reactive plans representing particular orderings of actions (action patterns) and behaviors that determine how and in which way these actions are expressed. That emotional responses should be continuous with normal action selection makes sense in light of current understandings of emotions. Damasio [1999], for example, suggests that essentially any species-typical behavior *is* an emotional response, because emotions are central to motivation.

To begin with, I have made a general-purpose behavior, drive-level, for representing the current level of activation for a drive. This becomes one of the pieces of state in behaviors that follow the sort of pattern of activation of emotional responses. Methods in these behaviors help determine the motivation level (see Figure 10-1). For example, the desire for isolation increases slowly in the presence of other animals, and decreases slowly when the animal is alone. Fear on the other hand increases radically in the context of a direct threat, and more slowly in the context of a fight between other nearby agents. It decreases slowly in isolation, or more quickly when being groomed out of the context of danger.

Emotional body postures are very much abstracted in our ALife simulation: they are simply colors for the agents. Their expression is also controlled by the emotion behaviors.

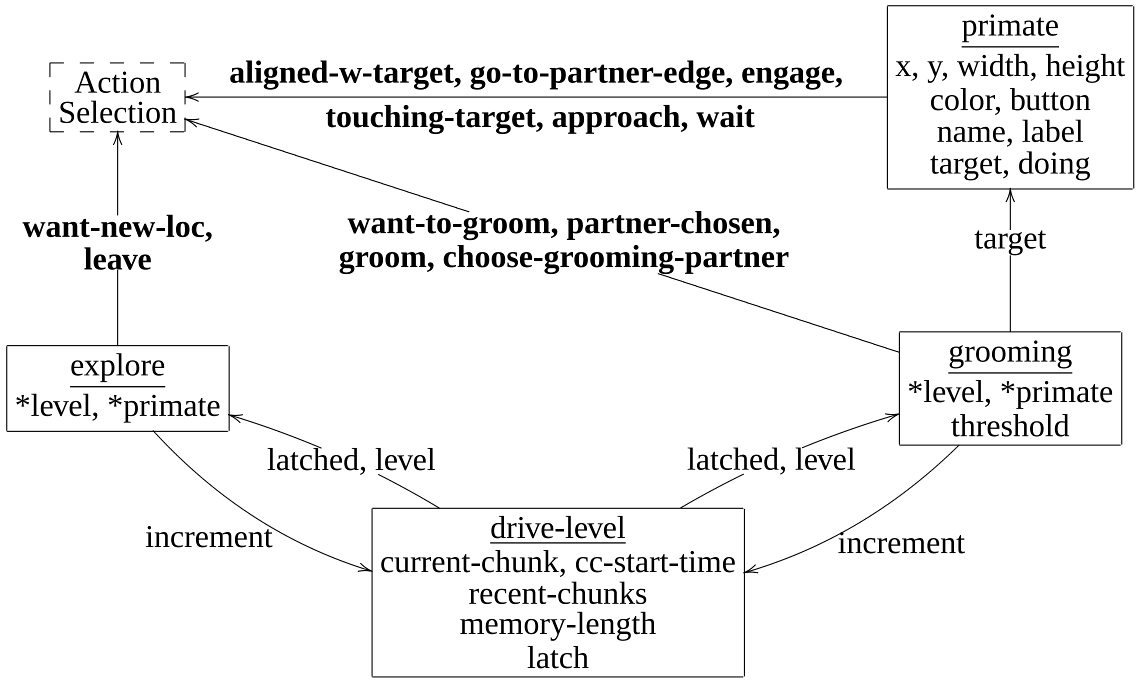


Figure 10-1: The behaviors supporting *move-n-groom*. The top two rows of primate 'state' are really characteristic of any object in the simulation.

10.5 A Model Showing Grooming Oscillating with Privacy

The current system controlling our primates is as follows. First, there are currently four behaviors. The first two, grooming and explore, are fairly simple behaviors as described above, controlling latent variables that might be identified with emotions or drives. The third is the drive-level behavior already described. The fourth, primate, has the relatively large responsibility of handling the primates' 'bodies' — it controls navigation of the agents around their enclosure.

If the simulations of the primates were particularly complex, primate would probably be decomposed into more behaviors. However, the most important output of our system is a simple list of events in the colony such as is produced by primatologists, since this is the information that is being compared to existing models and data. For the purpose of debugging, we also have a GUI representation of the agents, but they are represented by simple buttons, with color indicating their expression, and ASCII characters indicating their identity and some gestures. The observing function that produces the log of activity actually has more information than can be gleaned by observing the GUI, but not more than could be gathered by field workers observing real primates. Although there is of course less noise from ambiguities arising in the field in determining the intended object of a gesture in a crowded location, in general the level of reporting is plausible because it records only expressed behaviors.

life *D*	grooming *C* (want-to-groom)	(partner-chosen) (aligned-w-target)	**groom**
		(partner-overlap)	**go-to-partner-edge**
		(partner-chosen) (very-near-target)	**engage**
		(partner-chosen)	**approach**
			choose-grooming-partner
	wandering *C* ()	(want-new-loc) (target-chosen) (touching-target)	**forget-target**
		(want-new-loc) (target-chosen)	**leave**
		(want-new-loc)	**choose-isolated-place**
		()	**wait**

Figure 10-2: Reactive plan supporting the coordination of the ALife agents. See text for explanation.

Figure 10-2 shows the current reactive plan for coordinating potential conflicting actions between these behaviors. Each primate moves between three possible states — trying to groom, trying to be alone, and just sitting around. The current drive collection determines that the desire for grooming outweighs the desire for isolation, if it is operating. But it is equally possible to guard the competences with sense predicates based on the *relative* differences between drives. What **move-n-groom** shows is that prioritization drive levels can be combined for determining high-level action selection.

10.6 Preliminary Results and Future Work

We are still working on building the behaviors of our primates, and have thus not yet begun quantitative analysis of our results. However the transcript in Figure 10-4 shows a brief episode in the experience of a colony. When the animals want to be alone, they move towards a location (in 2-D space), when they are attempting to groom they move towards

CH	Charlie (all names should be unique in first 2 letters)
CH-	Charlie gesturing right
=CH	Charlie touching left
^CH	Charlie gesturing above
vCHv	Charlie touching below

Grey	- neutral: normal motion or sitting
Pink	- displaying
Red	- angry (fighting)
Orange	- frightened (screeching)
Lavender	- aroused
Purple	- mounting
Blue	- playing
Green	- grooming

Figure 10-3: Labels and colors indicating the state and identity of a particular agent in the simulated colony. The only colors in current use are grey and green.

each other. Mutual grooming as seen between Murray and Jean is at this stage coincidental — the agents do not yet deliberately cooperate. For example, Roger, who is not particularly in the mood to groom, but not particularly concerned with being isolated, ignores Alice. George, on the other hand, wants to move to an isolated spot, and is being chased rather pathetically by Monica who repeatedly tries to sit down by him as he moves. The agents shown here do not yet have any simulation of a 'theory of mind' — they cannot notice when their company is not wanted, or even that they are failing to meet a goal. Of course, both of these are well within the capacity of BOD representations, but current development efforts have been concentrating on the more general MAS aspects of this simulation, such as creating better debugging tools.

The next steps for this model will be to make the agents aware of the actions of others, and to model tolerance and aggression / fear. At this point, we will review the effects of varying parameters for such a system on the extent to which individual agents are able to meet their individual goals (e.g. amount of time spent alone or grooming.) Once we have this baseline of performance established, we will begin modeling intervention behaviors and social ordering.

10.7 Conclusions and Discussion

This chapter has given a brief glimpse not only into multi-agent BOD, but also into modeling conventional drive theory in a BOD system and the relationship between emotions and action selection. Further, the code for this chapter (see Appendix C) shows the functioning of real-time drives in the most recent implementation of POSH, and shows another example of a behavior library and a POSH script history.

```
George   APPROACH (0 54)               485591
 Ringo   APPROACH (0 67)               487575
Murray       WAIT (374 9)              487881
Monica       WAIT (45 236)             491908
  Jean   APPROACH Murray               497864
  Jean      ALIGN Murray               500125
  Jean      GROOM Murray               500254
 Alice   APPROACH Roger                500275
 Alice      GROOM Roger                503282
Murray   APPROACH Jean                 505554
  Jean   APPROACH Murray               505772
Murray      GROOM Jean                 506143
  Jean      GROOM Murray               506237
Monica   APPROACH George               509439
Monica      ALIGN George               510684
Monica   APPROACH George               510972
Monica      ALIGN George               513842
Monica   APPROACH George               513958
```

Figure 10-4: Part of a sample observation log for the simulated primate colony. “Approach” indicates walking, “wait” sitting or foraging, “align” engaging (sitting closely) for interactions, and “groom” is obvious. See comments in text.

The primate project is the first example of a MAS being built of BOD components. Thus, besides being a platform for exploring models of primate social organization, I expect this research to also be useful for examining the differences between modular intelligence of a single agent and the multi-agent intelligence of a community. Primates are interesting examples of successful communities, because they use fairly unstructured, low-bandwidth communication between agents.

Chapter 11

Extending BOD toward Modeling Neural Systems

11.1 Introduction

One useful and interesting application of complex agent technology is creating functioning models of naturally intelligent systems. In this chapter, I examine the extent to which this is possible under the BOD methodology. I also discuss what extensions to the current standard for BOD agents would be useful for modeling biological intelligence.

In Chapter 3 I argued that there is convergent evidence in the agent architecture literature supporting the utility and necessity of the structural attributes of BOD agents. If, as is often claimed [Brooks, 1991a, Hendriks-Jansen, 1996, Bryson, 2000b, e.g] biological agents face similar problems and constraints as complex artificial agents, and if the thesis of Chapter 3 is true, then animals might also be expected to have somewhere control structures. This chapter also examines evidence that this is in fact the case.

Besides examining the relationship between BOD and biological intelligence directly, this chapter also examines relationship between BOD and artificial neural networks (ANN). Although ANNs are vast simplifications of real neural systems, they have been a useful technology for helping us think about and model highly distributed systems of representation, control and learning. This work has proven useful both in science, by providing models, paradigms and hypotheses to neuroscientists; and to engineering, by providing adaptive control and classifier systems. Integrating ANN with agent software architectures may also further both science and engineering. BOD brings to ANN an understanding of modularity, specialized learning, and timing and synchronization issues. ANN brings to BOD a homogeneous adaptive representation which might in theory extend the sorts of adaptability possible under BOD.

ANNs can not so far be used for control systems that attempt to replicate the behavioral complexity of *complete* animals. One reason, as we discussed in Chapter 6, is that the complexity of such systems effectively requires decomposition into modules and hierarchy. This requirement is not theoretical, but practical. In theory, monolithic systems may be Turing complete; but whether attacked by design or by learning, in practice complex control requires decomposition into solvable subproblems. As we've seen, modularity is a key

$$\begin{tikzcd}
 & BOD \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & \\
BNN \arrow[rr, Leftrightarrow] & & ANN
\end{tikzcd}$$

Figure 11-1: Advances in artificial neural networks (ANN) have been both inspired by and used for better understanding neuroscience (BNN). This chapter shows that similar relationships can exist between BOD and both fields.

feature not only of BOD agents, but of many other architectures for complete complex agents.

This chapter begins with a discussion of modularity as found in mammalian brains. I the use this as the background for describing mapping between the features of BOD and other agent architectures and known or theorized functioning in mammal brains. Next, I return to the discussion begun in Chapter 6 of the limitations of BOD adaptivity, and discuss what extensions to BOD would be needed in order to model all of the adaptivity exhibited by natural intelligence. I conclude with a more practical sublist of biologically-inspired features of agent architectures that I consider useful and ripe for widespread implementation.

11.2 Modularity in Nature

There are at least three types of modularity in mammalian brains. First, there is *architectural modularity*. Neuroanatomy shows that the brain is composed of different organs with different architectural structures. The types and connectivity of the nerve cells and the synapses between them characterize different brain modules with different computational capabilities. Examples of architectural modules include the neocortex, the cerebellum, the thalamus, the hippocampus, periaqueductal gray matter and so forth: the various organs of the fore, mid and hindbrains.

Second, there is *functional modularity*. This modularity is characterized by differences in utility which do not seem to be based on underlying differences in structure or computational process. Rather, the modules seem to have specialized due to some combination of necessary connectivity and individual history. Gross examples include the visual vs. the auditory cortices. Sur et al. [1999] have shown at least some level of structural interchangeability between these cortices by using surgery on neonate ferrets. There is also other convincing and less invasive evidence. For example, many functionally defined cortical regions such as V1 are in slightly different locations in different people [Nestares and Heeger, 2000]. Many people recover capacities from temporarily debilitating strokes that permanently disable sections of their brains, while others experience cortical remaps after significant alterations of there body, such as the loss of a limb [Ramachandran and Blakeslee, 1998]. This evidence indicates that one of the brain's innate capabilities is to adaptively form functionally modular organizations of neural processing.

Thirdly, there is *temporal modularity*. This is when different computational configurations cannot exist contemporaneously. There are at least two sorts of evidence for temporal modularity. First, many regions of the brain appear to have local "winner take all" connection wiring where a dominant "impulse" will inhibit competing impulses [Hebb, 1949, Grossberg, 1999]. This neurological feature has been used to explain the fact that humans can only perceive one interpretation of visually ambiguous stimuli at a time [Pöppel, 1994]. Second, many cells in the brain are members of more than one assembly, and can perform substantially different roles in only subtly different contexts [e.g. in the hippocampus Kobayashi et al., 1997, Wiener, 1996]. Brain cell recording experiments showing that individual cells are associated with different stimuli and/or behavior, and indeed are members of different ensembles, depending on the animal's current context [e.g. Skaggs and McNaughton, 1998]. This sort of temporal modularity is not yet well understood, but it could have implications for individual differences in intellectual task performance such as insight and metaphoric reasoning.

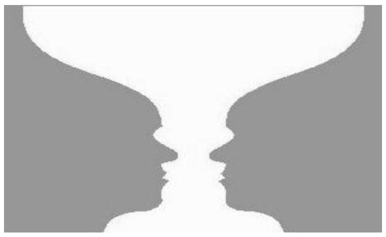


Figure 11-2: An ambiguous image. This figure can be seen either as a vase or as two faces, but not both at the same time. From [Gleitman, 1995, p. 213]

The presence of these forms of modularity in mammalian brains motivates a modular architecture in two ways. First, if we are interested in modeling the brain as a matter of scientific interest, we will need to be able to replicate its modularity. Second, the presence of modularity in the best examples of intelligent control available is that further evidence that modularity is a useful means of organizing behavior. Evolution is not a perfect designer — the mere presence of a solution in nature does not prove it is optimal. However, given the extent and complexity to which the brain has evolved, it is at least worth treating the utility of its features as hypotheses.

11.3 Mapping BOD Features to Mammal Brain Structure

Chapter 3 shows that complete agent architectures have converged on three sorts of architectural modules in order to support complex, reactive behavior. Skill modules and hierarchically structured reactive plans are used to focus attention on behaviors likely to be useful in a particular circumstance and provide temporal ordering for behavior. Environment-monitoring or alarm systems switch the focus of action-selection attention in response to highly salient environmental events. In BOD, behaviors correspond to skill modules, and the POSH control structures handle both plans and attention switching.

If this sort of organization is necessary or at least very useful for intelligent control, then it is also likely to be reflected in the organization of animal intelligence. This section relates these principles to what is known of mammal brain architecture.

11.3.1 Skill Modules

In the previous section we discussed modularity in mammalian brains. Using that terminology, we consider BOD behaviors to correspond roughly to functional modularity, particularly in the neocortex, and perhaps to some extent to temporal modularity. The strength of this correspondence various, however, both is a factor of BOD application and of brain region.

Consider a behavior for grasping a visual target. In a BOD agent, it would be likely to incorporate information that and mammals comes from the retinas, visual and associative cortices, motor pre-planning and motor coordination. It would also need to exploit somatic and proprioceptive feedback from the grasping limb, though some of this complexity might be masked by interfacing to other specialist modules. This end-to-end processing, encompassing both perception and action, contrasts with most understandings of brain modularity. Much of functional cortical modularity in mammals tends to be more general purpose or modality specific, for example the usual understanding of the visual, auditory, somatic cortices.

On the other hand, very recent research [e.g. Graziano and Gandhi, 2000, Graziano et al., 2001] shows that the premotor and motor cortices actually *do* represent not only motion, but also multi-modal input specific to various complete motor primitives, such as feeding, scratching and ducking blows or projectiles. These complex behaviors, triggered with single-cell simulation, indicates that animals is complex as monkeys not only use BOD-like modules, but represent them hierarchically. Further, the temporal modularity in the parietal cortex and the hippocampal formation, which is also multi-modal, is obviously not strictly parallel. Such temporally asynchronous and context-specific biological modularity motivates those BOD behaviors which have no associated parallel processes, but which are only active when supporting affected primitives.

Of course, most artificial agent models are not strictly biological, but this does not preclude a researcher interested specifically in biological modeling from using BOD to do so. As I have reiterated many times in this dissertation, BOD-like behavior decomposition is motivated primarily by expedience for the software engineer. Modularity in BOD serves primarily to support an orderly decomposition of intelligence into manageable, constructible units. However, for the researcher interested in modeling the brain directly, BOD can easily be used with known or theorized cortical modularity as a blueprint for skill decomposition.

11.3.2 Action Selection

The basal ganglia has been proposed as the organ responsible for at least some aspects of action selection [Mink, 1996, Gurney et al., 1998, Redgrave et al., 1999, Prescott et al., to appear]. In a distributed parallel model of intelligence, one of the main functions of action selection is to arbitrate between different competing behaviors. This process must take into account both the activation level of the various ‘input’ cortical channels and previous experience in the current or related action-selection contexts.

The basal ganglia is a group of functionally related structures in the forebrain, diencephalon and midbrain. Its main ‘output’ centers — parts of the substantia nigra, ventral tegmental area, and pallidum — send inhibitory signals to neural centers throughout the brain which either directly or indirectly control voluntary movement, as well as other cognitive and sensory systems [Middleton and Strick, 2000]. Its 'input' comes through the striatum from relevant subsystems in both the brainstem and the forebrain. Prescott et al. [to appear] have proposed a model of this system whereby it performs action selection similar to that proven useful in complex agent architectures.

Arbitrating between subsystems is only part of the problem of action selection. Action patterns must also be sequenced with appropriate durations to each step. The duration of many actions is too quick and intricate to be monitored via feedback, or left to the vagaries of spreading activation from competing but unrelated systems [Lashley, 1951, Houghton and Hartley, 1995]. Further, animals that have had their forebrains surgically removed have been shown capable of conducting complex species-typical behaviors — they are simply unable to apply these behaviors in appropriate contexts [Gleitman, 1995]. In particular, the periaqueductal grey matter has been implicated in complex species-typical behaviors such as mating rituals and predatory, defensive and maternal maneuvers [Lonstein and Stern, 1997]. However, there appears to be little literature as to exactly how such skills are coordinated. There is also little evidence that learned skills would be stored in such areas. We do know that several cortical areas are involved in recognizing the appropriate context for stored motor skills [e.g. Tanji and Shima, 1994, Asaad et al., 2000]. Such cortical involvement could be part of the interface between skill modules and action selection.

11.3.3 Environment Monitoring

Our proposal for the mammalian equivalent to the environment monitoring and alarm systems is more straight-forward. It is well established that the limbic system, particularly the amygdala and associated nuclei, is responsible for triggering emotional responses to salient (particularly dangerous, but also reproductively significant) environmental stimuli. Emotional responses are ways of creating large-scale context shifts in the entire brain, including particularly shifts in attention and likely behavior [Damasio, 1999, Carlson, 2000]. This can be in response either to basic perceptual stimuli, such as loud noises or rapidly looming objects in the visual field, or to complex cortical perceptions, such as recognizing particular people or situations [Carlson, 2000, Bechara et al., 1995b]. Again, there can be no claim that this system is fully understood, but it does, appropriately, send information to both the striatum and the periaqueductal grey. Thus the amygdaloid system meets our criteria for an alarm system being interconnected with action selection, as well as biasing cortical / skill-module activation.

11.3.4 Discussion

We can only begin to produce a mapping of BOD agent attributes to neural subsystems, primarily because the workings of neural subsystems are only beginning to be understood, but also because of differences in decomposition strategies. The primary function of a BOD architecture is to facilitate a programmer in developing an agent. Consequently, complexity is kept to a minimum, and encapsulation is maximized. Evolution, on the other hand, will eagerly overload an architectural module that has particular computational strengths with a large number of different functions. Nevertheless, we have identified several theories from neuroscience that are analogous to the features of BOD agents.

I would like to describe one more interesting biological analog to the mental architecture constructed under BOD. The structure is in fact a neat inverse, which is not surprising because it is not a theory of control, but of perception — specifically Rensink's theory of visual attention and comprehension [Rensink, 2000]. Rensink proposes that the visual scene is essentially covered with *proto-objects* which are monitored in parallel by the vision system. Only one item is fully attended to at any given time. That item is constructed of approximately four "fingers" of attention which bind proto-objects into the attended, fully represented object. Only attended objects can appear in episodic memory, or be associated with time. Proto-objects may however communicate location and gist, particularly on the level of priming. This is a striking inversion of the BOD model, where a single point of control attention (often focused on 3–7 BRP plan steps) governs the expression of behavior generated by a number of semi-autonomous behavior modules.

11.4 Adaptivity in Modular Systems

As we discussed can Chapter 6, BOD is designed to support specialized learning. We can design learning systems for the knowledge the agent is destined to know, and indeed, this is the dominant form of learning exhibited by mature agents in nature. The tradeoff for BOD being a good way to designing agents is that it is not particularly suited to other forms of learning — what in Chapter 6 I called *meta-learning*, learning plans or behaviors. I will now begin to discuss possible strategies within BOD and extensions to BOD for addressing these concerns.

Given that action selection requires structure, a natural extension of the standard BOD architecture would allow an agent to learn new reactive plans. There are at least three means by which this could be done. The most commonly attempted in AI is by *constructive planning*. This is the process whereby plans are created by searching for sets of primitives which, when applied in a particular order to the current situation, would result in a particular goal situation [e.g. Fikes et al., 1972, Weld, 1999]. Another kind of search that has been proposed but not seriously demonstrated is using a genetic algorithm (GA) or GA-like approach to combine or mutate existing plans [e.g. Calvin, 1996]. Another means of learning plans is to acquire them socially, from other, more knowledgeable agents.

Constructive planning is the most intuitively obvious source of a plan, at least in our culture. However, this intuition probably tells us more about what our consciousness spends time doing than about how we actually acquire most of our behavior patterns. The capacity for constructive planning is an essential feature of Soar and of most three-layer-architectures; however it is one that is still underutilized in practice. We suspect this will always be the case, as it will be for GA type models of "thinking", because of the combinatoric difficulties of planning and search [Chapman, 1987]. Winston [1975] states that learning can only take place when one nearly knows the answer already: this is certainly true of learning plans. Search-like algorithms for planning in real-time agents can only work in highly constrained situations, among a set of likely solutions.

Social or mimetic learning addresses this problem of constraining possible solutions.

Observing the actions of another intelligent agent provides the necessary bias. This may be as simple as a mother animal leading her children to a location where they are likely to find food, or as complex as the imitation of complex, hierarchical behavioral patterns (in our terminology, plans) [Whiten, 2000, Byrne and Russon, 1998]. This may not seem a particularly promising way to increase intelligence, since the agent can only learn what is present in its society, but in fact, it is. First, since an agent uses its own intelligence to find the solution within some particular confines, it may enhance the solution it is being presented with. This is famously the case when young language learners regularize constructed languages [Bickerton, 1987, Kirby, 1999]. Secondly, a communicating culture may well contain more intelligence than any individual member of it, leading to the notion of cultural evolution and mimetics [Dennett, 1995]. Thus although the use of social learning in AI is only beginning to be explored [e.g. Schaal, 1999], we believe it will be an important capacity of future artificial agents.

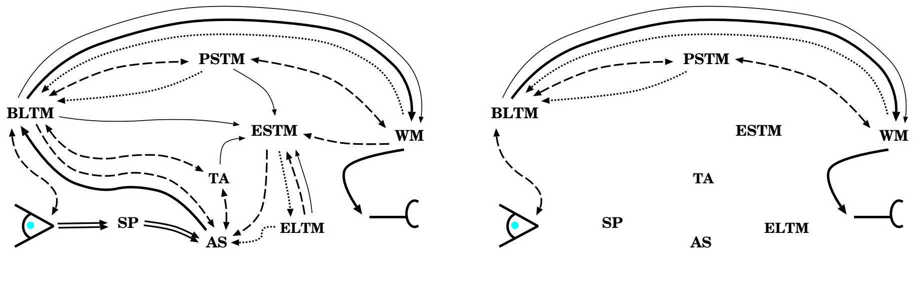
Finally, we have the problem of learning *new* functional and/or skill modules. Although there are many PhD theses on this topic [a good recent example is Demiris, 1999], in the taxonomy presented in this paper, most such efforts would fall under the parameter learning for a single skill module or behavior. Learning full new representations and algorithms for actions is beyond the current state of the art for machine learning. Such a system would almost certainly have to be built on top of a fine-grain distributed representation — essentially it should be an ANN. However, again, the state of the art in ANN does not allow for the learning and representation of such complex and diverse modules.

11.5 Requirements for a Behavior Learning System

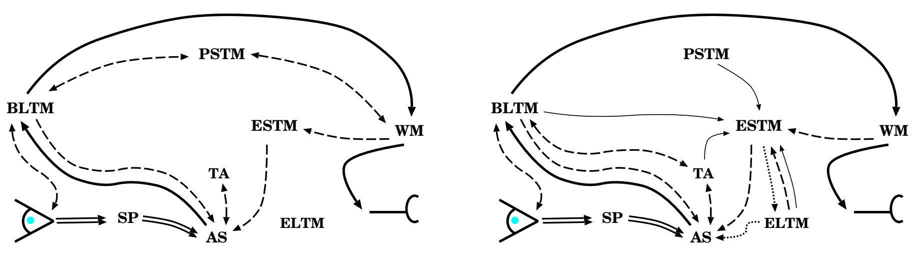
If the current state of the art were not an obstacle, what would a system capable of *all three* forms of adaptivity described in the previous section look like? I believe it would require at minimum the elements shown in Figure 11-3(a). This section explains that model.

Consider first the behavior or skill module system, Figure 11-3(b). The representation of BOD behaviors has been split into two functional modules: the Behavior Long Term Memory (BLTM) and the Perceptual Short Term Memory (PSTM). The persistent representation of the behaviors' representations and algorithms belong in the former, the current perceptual memory in the latter. There is further a Working Memory (WM) where the representation of the behaviors from the BLTM can be modified to current conditions, for example compensating for tiredness or high wind. In a neurological model, some of these representations might overlap each other in the same organs, for example in different networks within the neocortex or the cerebellum. The behaviors of course contain both perception and action, though notice the bidirectional arrows indicating expectation setting for perception [see e.g. Jepson et al., 1996, Hinton and Ghahramani, 1997].

The full path for expressed action is shown in Figure 11-3(c). This takes into account both standard action selection and environment monitoring. Here, with the learning arcs removed, we can see recommendations flowing from the behavior system to action selection (AS). Action selection also takes into account timing provided by a time accumulator (TA, see below) and recent action selections (decisions) stored in episodic short term memory (ESTM). Expressed action takes into account current perceptual information in PSTM as



(a) Complete Model (b) Skill Modules



(c) Expressed Behavior (d) Learning Action Selection

Figure 11-3: An architecture for allowing adaptation within skill modules, of new plans, and of new skill modules. Icons for sensing and action are on the lower left and right respectively. Dashed lines show the flow of information during an active, attending system. Dotted lines are pathways for consolidation and learning. The heavy solid line is the path of expressed behavior; the double line represents the constant perceptual pathway for environmental alerts. The fine lines indicate references: the system pointed to references representations in the system pointed from.

well as the current modulated version of the behaviors in WM.

I have also provided a separate path for basic perceptual reflexes such as alarm at loud noises or sudden visual looming. The module for recognizing these effects is labeled SP for Special Perception. In nature this system consists of subcortical perception systems like the superior and inferior colliculi, but also has connections to the cortical system, so that reflexive fear responses can be developed for complex stimuli. However, is probably important to isolate the fundamental system from possible modification by the skill module learning system.

To make action selection adaptive (Figure 11-3(d)) we provide first a time accumulator (TA) as proposed by Pöppel [1994] and Henson [1996] and episodic short term memory (ESTM) as postulated by a large number of researchers (see [McClelland et al., 1995] for experiments and review.) Episodic long term memory (ELTM) is included for good measure — as consolidated experience, it might also represent other forms of semantic memory, or it might actually be homologous with BLTM.

Finally, in keeping with [Teyler and Discenna, 1986, McClelland et al., 1995], this model assumes that many of the modules make reference to the state of other modules rather than maintaining complete descriptions themselves. This is considered an important attribute of any system which needs to hold a large number of things which are learned very quickly, because it allows for a relatively small amount of state. Such reference is considered important in computer science as a means to reduce the probability of conflicting data sets, and is also a likely feature of evolved systems, where existing organization is often exploited by a variety of means.

11.6 Future Directions: From Neuroscience to Complex Agent Architectures

Fortunately, implementing such a complex system is not necessary for most agent applications. In general, the adaptive needs of the agent can be anticipated in advance by the designer, or discovered and implemented during the process of developing the agent. We do, however, suspect that some of the systems being discovered and explored in neuroscience may soon become standard functional modules in agent architectures, in the same way that action selection and alarm systems are now.

One of the capacities often ascribed to the hindbrain, that of smoothing behavior, should probably be given its own functional module. This allows modules that create motor plans to operate at a relatively coarse granularity. It also allows for the combination of influences from multiple modules and the current situation of the agent without complicating those skill modules. The only current architecture I know of that explicitly has such a unit is Ymir [Thórisson, 1999], where the Action Scheduler selects the most efficacious way to express messages given the agent's current occupations (see further Sections 5.5.1 and 12.3.2). This sort of capacity is also present in a number of new AI graphics packages which allow for the generation of smooth images from a script of discrete events [Brand et al., 1997, Baumberg and Hogg, 1996]. The fact that such work is not yet the norm in robotics (though see [Schaal and Atkeson, 1994, Atkeson et al., 1997]) may be partially due to the fact that a physical agent can take advantage of physics and mechanics to do much of its smoothing [Bryson and McGonigle, 1998]. As robots attempt more complex feats such as balancing on two legs, providing for smoothed or balanced motions may well deserve dedicated modules or models similar to those cited above.

I also expect that sparsely represented records of episodic events (as in Section 7.6.3) will become a standard mechanism. Episodic records are useful for complimenting and simplifying reactive plans by recording state about previous attempts and actions, thus reducing the chance that an agent may show inappropriate perseveration or redundancy when trying to solve a problem. Further, as mentioned previously, episodic memory can be a good source for consolidating semantic information, such as noticing regularities in the environment or the agent's own performance [e.g. Smart, 1992]. These records can in turn be used by specialized learning systems for particular problems, even if a full-blown skill learning system has not been implemented.

11.7 Additional Humanoid Subsystems

Many researchers are currently working on emotion modules for complex agents. Emotions can be used to provide complex reinforcement signals for learning behavior [e.g. Gadanho, 1999], and to represent motivation level [e.g. Tu, 1999]. These functional considerations can be addressed from within BOD (see for example Chapter 10). Explicitly modeling human-like emotions may also be useful from an HCI standpoint [e.g. Breazeal, 2000], or to provide the agent with model necessary to empathically comprehend human social interactions [c.f. de Waal, 1996]. I am however skeptical of the need for or practicality of an independent emotion module for two reasons. First, there is a great deal of evidence that the basic emotions evolved independently at different times in our history. This suggests that a single emotion module might not be appropriate. Second, emotions are intimately involved in action selection. In vertebrates, emotions serve as specialized mechanisms for focusing attention, including by deactivating large sections of the cortex [Damasio, 1999]. In fact, Damasio [1999] implies that any species-typical behavior pattern is effectively an emotional response. This suggests that it is impossible to separate emotions from motivation in action selection. Consequently, I believe emotions are best modeled within the existing BOD framework.

An even more contentious area of research is that of consciousness an explicit knowledge. None of the models presented in this dissertation make claims regarding which of their aspects might be conscious, or what of their information might be explicitly known. This is not due to lack of interest or capability. Rather, it was simply not necessary feature of these particular models. Notice also that there is a lack of data: it is often unclear how much consciousness affects even human action selection [Bechara et al., 1995a, Dennett and Kinsbourne, 1992]. Nevertheless, I can imagine modeling some of the current, well-specified theories of consciousness [e.g. Norman and Shallice, 1986, Dennett and Kinsbourne, 1992] using the methodology presented in this dissertation.

11.8 Conclusions

BOD hypothesizes the following:

1. most of intelligence is broadly modular,
2. arbitrating between modules requires a specialized mechanism for action selection,
3. complex behavior requires hierarchical and sequential structure for arbitration, and
4. switching attention from complex behavior to new salient features or events also requires a specialized mechanism, operating in parallel.

In this chapter I have shown that these hypotheses are reasonable for natural as well is artificial intelligence. I have shown some of the relationships between BOD architectures, ANN research and brain science, and suggested possible future work in all three areas. I hope that one day there will be as rich exchange between researchers in complex agent architectures and those in behavioral neuroscience as the currently is between ANN and neuroscience.

Chapter 12

Extending BOD toward Industrial Applications

12.1 Introduction

Could BOD be used in an industrial setting? I believe so for two reasons. First, BOD is designed to address the concerns of industrial development processes. BOD agents are designed to be modular and easy to maintain, easily decomposed for multi-programmer projects, and easy to integrate. The behavior structure can incorporate existing packages and solutions; the POSH control structures are designed to be as intuitive as possible for conventional, sequential programmers.

The second reason is that I have twice worked with it in industrial or semi-industrial research settings. Unfortunately, neither of these projects have reached completion, so I have classified them under 'future work'. The first was a blue-sky virtual-reality (VR) research effort for an entertainment company, which unfortunately was terminated when the company closed its digital research division. The second was a dialog tutoring agent still under development at the Human Computer Research Centre in Scotland. In this case I had to leave the project after only a few months, due to personal obligations.

These two projects may appear similar in that both deal with personified software agents. In fact, only the VR project involved building personalities similar to the artificial life projects demonstrated in this dissertation. The dialog agent is much more like conventional large-scale AI project. It requires integrating many disparate technologies into a functioning, coordinated system. The VR project also required coordination, but this time primarily between disparate teams of designers.

Many other industrial applications have similar needs. For example medical monitoring systems [e.g. Doyle et al., 1999], or intelligent environments [e.g. Coen, 1997]. Any problem where a system is required to prioritize conflicting goals or integrate multiple sources of information can potentially be viewed as a BOD agent.

The rest of this chapter describes the progress that was made on the two projects mentioned above. As such, they not only describe some exciting possibilities for future BOD agents, but also serve as two final examples of BOD methodology.

12.2 BOD and Tutoring Dialog systems

Dialog systems currently require an enormous amount of engineering, and typically result in relatively brittle systems. This section reports work exploring the use of reactive planning in general and BOD in particular for simplifying dialogue system design.

12.2.1 A BOD Decompositon of a Dialog Project

I will begin by considering as an example the problem of dialog management in a system such as TRAINS-93 [Allen et al., 1995]. This system was a major effort in addressing the complete problem of dialog, including having a system capable of planning and acting as well as discussing its plans and acquiring its goals verbally. The TRAINS system served as an assistant to a manager attempting to make deliveries of commodities, such as bananas and orange juice, to a number of different cities. In addition, various cities had various important resources, such as trains, cars, processing plants and raw commodities. These cities were connected by rail, so transport requires scheduling in both time and space.

To build a dialog system similar to TRAINS-93, we must first list a rough set of capabilities we expect the agent will have. In this case, we can use the existing system as a guide, and assume that the agent will eventually need the same set of speech acts as capabilities. While we are organizing the gross behavior of the agent, these speech acts will be simple primitives that merely indicate their place in execution by typing their name. This practice of implementing bare, representative functionality as a part of early design is called *stubbing* in OOD. Based roughly on TRAINS speech acts, the initial list of primitives is the following:

accept or **reject** a proposal by the dialog partner,
suggest a proposal (e.g. a particular engine or location for
    a particular task),
request information (e.g. a particular of the current plan),
supply-info in response to a request, and
check for agreement on a particular, often necessary
    due to misunderstandings.

Working from these primitives, we can construct a high-level plan for dialog management in just a few lines (see Table 12.1). Here, sensory checks for context are indicated by parenthesis. The primitive actions listed above are in bold face.

The highest level concern for this plan is simply whether the agent should take a turn, or whether it should wait quietly. Once it has decided to take a turn, the highest priority behavior is to fulfill any discourse obligations, including the obligation to try to understand the previous statement if it was not successfully parsed. If there are no existing obligations, the next highest priority is to resolve any inconsistencies in the agent's current understanding, indicated here by having a requirement not entailed by the task bound to some value. This indicates a need either for clarification of the requirement, or of the current task.

If there are no such inconsistencies, but there is an outstanding task to perform, then the next highest priority is to complete the task, which in the case of TRAINS usually involves

```
(if my-turn)
    (if request-obligation) (if check-request false) **reject**
    (if request-obligation) (if check-request true) **accept**
    (if inform-obligation) **supply-info**
    (if comprehension-failure) **check** *last-utterance*
    (if bound-non-requirement)
        (if requirement-checked) **check-task**
        **check-requirement**
    (if requirement-not-bound)
        **pick-unbound-req , suggest-req**
    (if (no task)) **request-task**
wait
```

Table 12.1: In this table, indentation indicates depth in the plan hierarchy. Notice that the action primitives generally assume deictic reference, where the perception primitive has set attention to a particular task or requirement.

assigning a particular resource to a particular slot in the problem space. Finally, if there is no task, then this agent, having no other social or personal goals, will seek to establish a new one.

This simple plan indicates a number of elements of state the agent is required to keep track of. These elements in turn indicate behaviors the agent needs to have established. To begin with, the agent needs to know whether it currently believes it has the turn for speaking. Although that may be a simple of bit of information, it is dependent on a number of perceptual issues, such as whether the dialogue partner is actively speaking, and whether the agent itself has recently completed an utterance, in which case it might expect the other agent to take some time in processing its information. The agent may also be capable of being instructed to wait quietly. Further, that waiting might also be time bounded.

12.2.2 Building a Behavior Library and Drive Structure

To a first approximation, the primitives used in the plan above can be arranged into behaviors as shown in Figure 12-1.

The constructive planning required by TRAINS-93 can also be replaced by a fairly short reactive plan (omitted for space) though still supplemented by an $A*$ search algorithm for finding the nearest resources. This suggests that a reasonable initial drive structure for the TRAINS-like dialog agent might be:

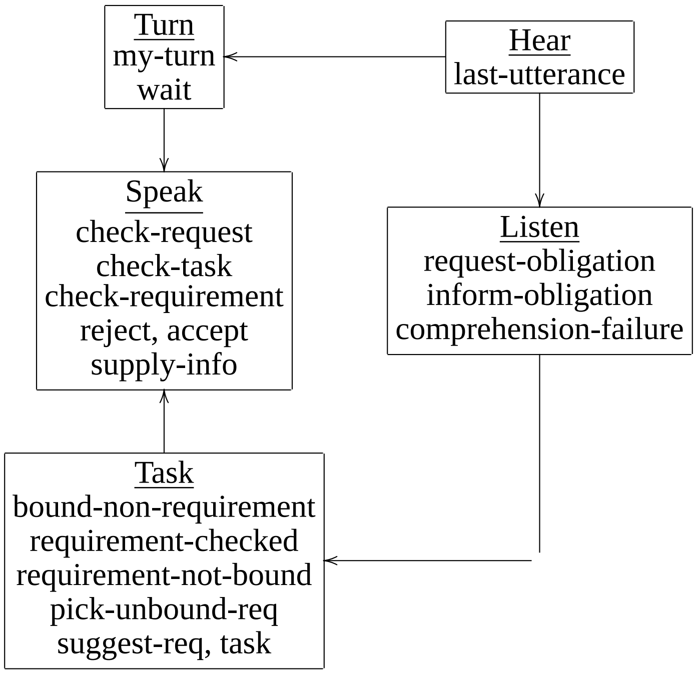


Figure 12-1: A first cut at a behavior decomposition for a TRAINS-93 type dialog agent. Unlike the other behavior diagrams in this dissertation, here the contents of the behavior are just the names of the primitives that that behavior will provide. This represents Step 4 of the BOD initial decomposition process (see Section 8.2.1.) Arrows still vaguely indicate the general information flow between behaviors.

$$\uparrow \left\langle \begin{array}{crl} \underline{\text{Priority}} & \underline{\text{Releaser}} \Rightarrow & \underline{\text{Action}} \\ 4 & (\text{if noise}) \Rightarrow & \textit{listen} \\ 3 & (\text{if need-answer}) \Rightarrow & \textit{think} \\ 2 & (\text{if my-turn}) \Rightarrow & \textit{take-turn} \\ 1 &) \Rightarrow & \text{wait} \end{array} \right\rangle \qquad (12.1)$$

This small plan serves as the parallel operating root of the action selection for the entire dialog agent. The plan that would eventually be derived from Table 1 would fit under the label *take-turn* above. The reactive plan for scheduling, including the call to the search algorithm, would fit under *think*. A drive structure like this allows another speaker to interrupt, since *listen* has the highest priority. The entire system still relies on the basic behaviors shown in Figure 1. The act of attempting to take a turn would set the flag for 'need-answer' if a problem requiring domain-specific planning has been encountered. Solving such a problem should unset the flag, so that turn-taking might again operate. Notice that the drive structure has no goal, so will never terminate due to success. Also, the lowest priority element has no precondition, so the drive might never terminate with failure, unless *wait* has a timer and a limit after which *wait* itself fails.

12.2.3 Scaling the System

The above system obviously hides a great deal of complexity: the problems of parsing the dialog input and constructing sensible output are completely untouched. On the other hand, a BOD system is sufficiently modular that these procedures may be primitives or 'black boxes,' since many AI systems for language parsing and generation have already been constructed.

The above analysis was actually preliminary work for organizing an even more complex dialog project. The intention would be to use BOD to organize dialog management for an even more complex system than the one shown above. The problem domain is tutoring basic electricity and electronics, and we hope to integrate systems that are capable of a wide range of behaviors for assisting students. Examples of desired behavior include analyzing incorrect answers in order to diagnose the learning failure, and providing multi-turn, Socratic method tutoring to lead the students to correcting their basic misconceptions. To be useful with real students, this system will need to be sufficiently reactive to allow the student to either solve the problem prematurely, and also be able to branch into a greater depth of explanation in response to a query or further errors from the student. The design specifications of this tutoring system are described further in [Core et al., 2000].

12.2.4 Specialized Learning in a Dialog Agent

BOD is designed to enable learning within a behavior; the rate at which state varies is one of the chief cues for what state should be clustered into a particular behavior. Machine learning techniques can be used for constructing a behavior, or at least part of its state. Another promising research direction would be to incorporate statistically acquired semantic lexicons [e.g Lowe, 1997] into a dialogue agent. This could quickly broaden the scope of the agent's ability to recognize conversational contexts. An agent with this lexicon could recognize entire classes of semantically similar sentences for any one programmed interaction.

Similarly, I would like to incorporate the statistically acquired mechanisms of natural language generation of Knight [Knight and Hatzivassilogon, 1995, Oberlander and Brew, 2000] into a dialog agent. This would allow varying the generative output of a dialog system to be appropriate for various audiences simply by training the mechanism on an appropriate corpus.

Ultimately, it would be interesting to attempt to learn dialog patterns directly from corpora as well. In this case, we could create a 'learning Eliza' with only basic turn-taking mechanisms built into the system. The system might be vacuous [Searle, 1980], but this might not be apparent in gossip-level conversations.

12.3 BOD and the Integration of AI characters into VR Entertainment

The evolutionary utility of play is considered to lie in enabling an individual to acquire and rehearse complex behaviours, as well as to learn appropriate situations in which to express them [Bekoff and Byers, 1998, Byrne and Russon, 1998]. This section addresses the problem of incorporating pre-packaged artificial intelligence into a creative play environment for a child.

12.3.1 Character Architecture and Constructive Narrative

Much research into agents for entertainment concentrates on the problem of combining the concept of a script with the notion of autonomous, reactive characters [Hayes-Roth and van Gent, 1997, Lester and Stone, 1997, André et al., 1998]. A better approach to constructive narrative eliminates this problem by changing the top level creative design from a *script* to a *cast of characters*. This simplifies the task of the player by removing the need for character addition, substitution, alteration, or removal. It has the penalty of removing a substantial element of narrative structure: a sequential order of events. However, this problem has already been addressed by the creators of role-playing and adventure games. Their solution is that plot, if desired, can be advanced by knowledgeable characters, found objects, and revealed locations. Structure is produced through the use of geographic space as well as character personalities. Personality traits such as loyalty, contentment or agoraphobia can be used to maintain order despite a large cast of autonomous character, by tying particular characters to particular locations. Developing such characters requires an agent architecture powerful enough to support this complexity. It also requires sufficient modularity to allow reasonably quick construction of behaviour patterns.

Most virtual reality agent architectures are fundamentally behaviour-based, and at least partially reactive [see Sengers, 1998, for a recent review and critique]. This is because the reactive, behaviour-based AI revolution of the late 1980s [Kortenkamp et al., 1998] was primarily the triumph of a design approach. Behaviour-based AI is simpler to design than a monolithic intelligence system because it allows the decomposition of intelligent behaviour into easy-to-program modules, with more localised control structures. Specifying that the intelligence should also be reactive removes the complex problems of learning and constructive planning from the agent. In spite of limiting the potential complexity of the agent's capabilities, the behaviour-based approach has been more successful in achieving interesting, believable characters than any fully human-specified or fully machine-learned approach simply because it empowers the human designer.

There are two sets of problems associated with using the established AI 'complex agent' architectures. One is getting the correct level of control for scripted personalities or behaviours. As I argued in Chapter 3, most hybrid architectures do not seem truly reactive enough to support the abrupt and frequent changes in context possible in a play scenario. Their 'reactive' elements are constrained to switching in new, complete plans during exceptional circumstances — for example if a fire alarm has sounded. When working with children, a more consistent sort of responsiveness is required in order to respond to unexpected assistance or interruption by the child. These events are more likely require movement within the same script than restarting or changing scripts. On the other hand, more purely reactive architectures make scripting coherent behaviour very difficult. In other words, constructive play is another domain that requires agent architectures with BRPs (see Chapter 4).

The other set of problems is associated with the technical difficulties of controlling a real-time multi-modal VR system. Very few AI architectures support the millisecond precision and modality coordination necessary for believable, engaging real-time interactions. These concerns are critical for all VR, but are particularly apparent when dealing with dialogue and gesture [Thórisson, 1998].

12.3.2 Spark of Life

Working with Kris Thórisson, I developed a solution for these problems, Spark of Life (SoL). SoL is essentially Ymir [Thórisson, 1996, 1999] extended with POSH action selection (see Section 5.5.1 for the details of the extension).

Ymir is a highly modular, hybrid architecture which combines features from classical and behaviour-based AI, and provides a system that can simulate in great detail the psychosocial dialogue skills of humans. Real-time, face-to-face dialogue encompasses a broad range of perceptual, cognitive and action requirements. Ymir addresses these phenomena, including natural language and multi-modal input and output (facial expression, gesture, speech, body language), load-balanced handling of time (from short reactive behaviours like fixation control to the execution of several seconds of multi-modal actions), and employs a modular approach that enables the creation of complex, human-like behaviour.

SoL consequently encompasses the following capabilities: multi-modal perception and action, real-time speech input and output, memory, and planning. SoL's modularity combined with robust, simple control makes it ideal for constructive play by allowing for easy additions and modifications.

12.3.3 Adapting BOD to SoL and Constructive Narratives

The SoL architecture provides the framework for the middle layer of our proposed three-layer design approach. AI facilitates the creation of a socially engaging world; however such a world also requires careful overall creative design, and a rich visual and behavioral structure. Because SoL is both behaviour based and has POSH action selection, it is an excellent platform for practicing BOD.

However, BOD has to be adapted somewhat for SoL. This is because Ymir, like many architectures including PRS and Soar, represents explicit knowledge in a single, general purpose knowledge base. Since SoL's VR timing skills hinge on particular Ymir's motor representation, the Motor Lexicon, SoL follows Ymir's representational framework, including knowledge representation. Modularity in knowledge can still be at least documented, in both Ymir and SoL, by dividing the knowledge base into a variety of Knowledge Areas. Also the Motor Lexicon has a hierarchical structure which is not only a form of self-documenting modularity, but can also be quite useful for incremental development. As the Motor Lexicon is elaborated, new actions can sometimes be automatically absorbed by the SoL/Ymir Action Scheduler, even if they are not specifically referenced in plans. However, the link between perception and action that is so explicit in standard BOD is no longer as clear under SoL.

12.3.4 The Responsibilities of AI Developers

Development of a constructive play VR world requires development on three levels:

1. a high, artistic design level for creating story and characters,
2. a middle, behaviour-based design level for creating personality in character agents, and
3. a low, VR design level for basic capabilities and appearances.

AI developers should not necessarily be expected to be sufficiently skilled artists that they can create the plots and characters needed for a fully engaging interactive play experience. AI attracts (and perhaps requires) developers with a hubristic belief in their own ability to replicate the thinking skills of others. However, good artists devote years of attention, and often their formal education, to perceiving and constructing the things that make a situation interesting, æsthetic and fun. The design process above places the AI developer as an intermediary between the artistic and the engineering aspects of the project. The AI developer is in the best situation to understand both requirements and restrictions of the overall project, and therefore has considerable responsibility for communication as well as developing solutions.

The AI expert is responsible for taking a set of motivations, goals, knowledge, personality quirks and skills, and creating an agent that will behave coherently according to these. In a rich virtual environment designed for free, creative play an autonomous character should be able to prioritise its goals and display its intentions. It should exhibit both persistence and resolution while at the same time being aware and opportunistic. In short, it should have a recognisable personality. Developing the initial set of character attributes, however, is not necessarily solely the task of the agent expert. It *is* necessarily the task of one or more creative artists. The artist's responsibility is to provide well formed and interesting characters, skills and situations, to design potential plots and plot twists. This is level 1 of the design process model. In this, as in most industrial design, it will be best if the artists work in a team with the agent developers, who can help the artists understand the limits of the agent's behavioral and expressive capabilities.

The agent developers are themselves constrained by the particular platform on which the artificial agent is to be implemented. In robotics these constraints come from the robot's hardware; in virtual worlds they come from the graphics environment in which the agent will be embodied. Creating this platform is level 3 of our design process. It is the responsibility of the AI developer to provide requirements for, and understand the constraints of, the underlying platform. Again, the character personality developer may or may not be the correct person to develop the agent's behavioral platform, depending on whether the platform in this context also provides the basic behaviours, or behaviour primitives, for the agents.

Drawing a line between levels 2 and 3 can be difficult. For example, it may make sense to put collision detection or motor smoothing into the 'world' (i.e. the graphics environment itself), either for more efficient performance of the system or for cleaner implementation and easier debugging. In nature, vertebrates have dedicated systems for providing such smoothing in their hindbrain [Carlson, 2000], as well as being able to rely on physics for smoothness and consistency. In a simulated world the division between an agent's own perception and the world itself may not be well defined. Implementations in level 3 can become a point of contention because on either side of the fence between graphics and AI, very different skill sets have been developed, and people working on each side may prefer very different solutions to the problems at hand.

Grossly, the levels of our design process model correspond to the different sides of SoL. The interface between levels 1 and 2 leads to specifications of personalities and drives, and the interface between levels 2 and 3 lead to the implementation of the behaviours. But as is emphasised under BOD, the design process has to happen iteratively. Many forms of technical constraint might only be recognised after development has begun. Further, as the system develops, it can provide considerable creative inspiration to the designers. Even more importantly, early users, particularly those coming from outside the project, will discover both shortcomings and unforeseen creative potential in the system. All of these sources of information should lead to periods of redesign and renegotiation between the various levels of the project. Further, personality may be demonstrated in subtle motions best provided in the behavioral level, or complex behaviour may require or suggest changes to the plans and drives. Thus all three levels of the design process must be available for cyclic development and reanalysis. The AI programmers working primarily at level 2 cannot be abandoned to try to satisfy potentially impossible constraints coming from isolated processes on either side of the project.

12.4 Case Study: Creating Characters for an Adventure Narrative

The design process described above was developed as part of a research effort at LEGO to create an interactive virtual reality entertainment package that allows children to engage in creative and constructive play within an established action/adventure framework. The project illustrates the design principles above, and gives indication of the efforts and difficulties involved. I will refer to the AI portion of this large-scale, multi-faceted research effort as the "castle character project". This effort included a detailed, relatively large virtual world with a castle situated on rolling hills, surrounded by a mountain range. A full moon hangs in the sky; the sun just under the horizon. Users can enter the world either through a desktop, or as fully embodied virtual (humanoid) LEGO characters with full body tracking and immersive glasses with displays.

12.4.1 High Level Design

In the case of the castle character project, much of the character content was predetermined, as it was a virtual version of an active product. The general appearance of the characters, an outline of their personalities, as well as their world, had been developed as a part of the marketing, but no stories had been created. The domain was a magic castle, inhabited by an evil knight and various magical entities. Much of the larger VR research effort was

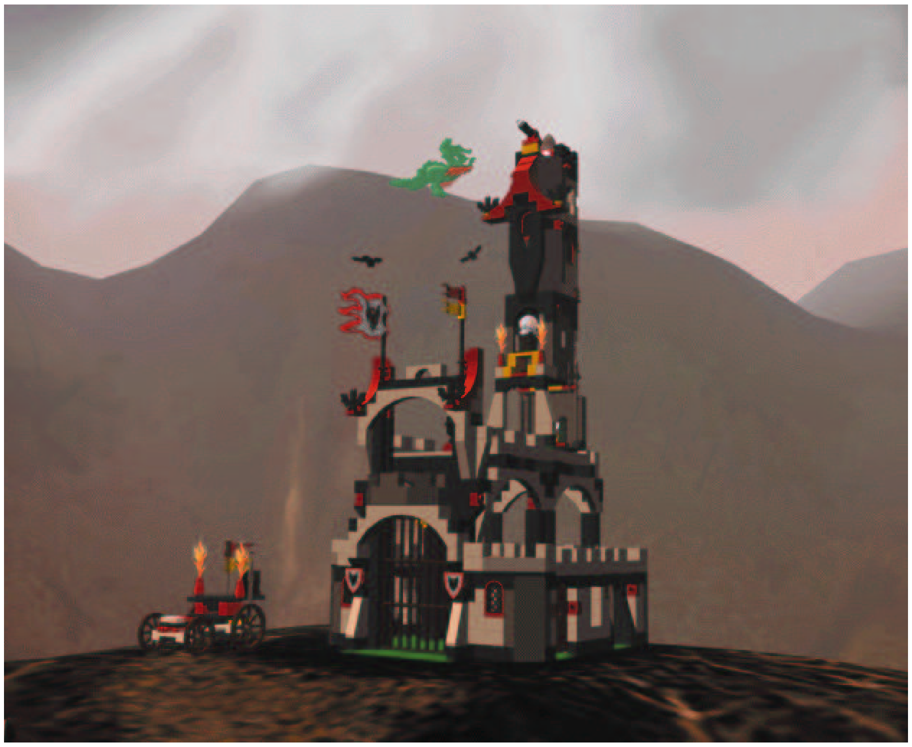


Figure 12-2: Still from the castle project, a real-time interactive virtual reality environment. Image ©1998 The LEGO Group

dedicated to ensuring that simply exploring the space would be intrinsically rewarding, but it was the introduction of moving characters that made the virtual experience become alive and magical. For example, there is a SoL character named Puff. Puff is a talking, flying green LEGO dragon. Puff can discuss the castle, or be encouraged to demonstrate his flying ability.

The first step toward creating an interesting narrative for a set of characters is to understand the constraints of the task and the system. One set of constraints comes from the character's environment, e.g. the size and features of open spaces: The castle world, though complex and interesting, is not very large relative to the size of the characters, so this constrains the characters motions inside the castle. This can be compensated by setting the most gross motion (such as large-character flying and sword fights) to the space surrounding the castle. Another set of constraints are those dependent on the expected users of the system. Because expected users were young, naïve to virtual worlds and, perhaps most importantly, only exposed to the system for a few minutes total, we considered it essential to make the characters interesting whether or not the user deliberately attempted to interact with them. The solution was to make the characters interact with each other as well. They were also designed to react to the visitor in their domain in a way that encouraged exploration, but not to be too forceful or too intrusive on the user's experience. To maintain interest, the characters should act and interact in such a way that they generate continuous change. There should be no steady state that the system of characters can reach if the user is being passive.

The constraints of the virtual environment and the pre-existing product meant that most of this change had to take the form of arrivals and departures, as well as a few gross gestures. This effect was achieved by designing characters with various incompatible goals. For example, a witch could frequently fly around the castle in a quest for intruders. When she found the intruder she would do little other than land nearby, slightly approach the stranger and cackle. However, her presence might attract other characters, some of whom might in turn repulse her (she was designed to fear flying bats). Having characters that are attracted by some situations, yet repulsed by either crowds or other characters, can help maintain the amount of free space needed for character motion. In addition, it limits the number of simultaneous interactions, and therefore the amount of confusion. This allows the designers to quickly focus the interest for the short-term visitor.

Notice that stateless 'reactive' social behaviours such as flocking [e.g Reynolds, 1987, Matarić, 1992] will not be sufficient — the characters here are doing more than being repulsed, attracted and avoiding obstacles. They are displaying personalities. A visitor can learn individual character's traits, and then manipulate these deliberately. Exploring the personality space of the characters in the world becomes part of the puzzle, and part of the fun.

12.4.2 Encoding Personality

After creating a rough description of the desired world, the next task is to develop a first-cut description of the reactive plans which will encode each character's personality. Starting from the descriptions of the characters set by the marketing department of the product, and keeping in mind the constraints determined in evaluating the task, each character was described in terms of three to five goals or drives. Further, the behaviour associated with achievement of these goals was visually described. This work was done by a team of in-house artists and external creative consultants, with the AI team participating both creatively and as technically informed resources.

Once the personality of the characters has been sketched, the next steps were as follows:

- Prioritising goals or gross behaviours and determining their necessary preconditions. For example, the witch described above has a goal of patrolling the castle from the air. This has a fairly high priority, but the motivation should be reduced by the performance of the act, so that in general she circles the castle only three times. She has a priority of landing in a room in which she has seen an intruder, once she no longer desires to fly. She also avoids bats.

- Determining necessary behaviour primitives and behaviour states. For example, the witch has to remember if she saw an intruder on her patrol. A bat might approach an intruder closer and closer over successive swoops. A state within the bat's swooping behaviour enables it to keep track of its current level of 'boldness,' which in turn determines its trajectory. Some characters can be made into friends by playing with them. These would have to remember how friendly they feel towards a particular person. Seeing the user, avoiding the walls of the castle, flying and landing are behaviour primitives required by all of these agents.

- Developing and testing the behaviour libraries and the scripts.

The architectural and methodological support we developed for this level has already been discussed.

12.4.3 Developing Perception and Action Primitives

In developing behaviour libraries, the task of the personality designer connects to the task of environment's architects. For the castle character project, some of the potential difficulties of this relationship were overlooked, and caused some of the greatest difficulties of the AI effort.

There are several possible approaches for building the basic movement primitives. One straightforward approach would be for the character developers to program the behaviours from scratch using models prepared by the graphic artists. There is a general problem for this approach: As mentioned earlier, AI programmers are not necessarily artists or students of natural motion. Animals have evolved complex motion behaviours, constrained by physical forces and structures not normally modelled on an artifact, particularly one designed to run in real time, so difficult to take into account. Animals are also constrained by habits of behaviour, whether general to a species or specific to an individual. Even if æsthetic motion primitives are achieved by an AI programmer, the process of programming them is likely to have been very time-consuming. Nevertheless, this was the main source of behaviors for the one SoL character — Puff, the flying and talking dragon.

Another potential source of behaviour primitives explored on the castle character project were the efforts of a team of animators already working on the project. The idea was to segment animations into sets of behaviours suitable as exemplars of various behaviour primitives. A continuous variety of behaviour could be derived from combining and connecting fixed sets of canned behaviours. Unfortunately, animations also proved slow and difficult to develop. More importantly, the format the animations were produced in was determined to be incompatible with the primary real-time virtual reality environment.

We also explored an intermediate solution: a purpose built animation tool for ‘quick and dirty’ animation segments stored in an appropriate format for the main VR engine. This technique was used for creating some of the most life-like motion on the castle, a guard that responded to an approaching camera / observer by turning and facing it. The intelligence behind this character was purely reactive, and did not use SoL, but it did show the promise of this technique. Motion capture of humans participating as puppeteers was the final source of ‘intelligence’ explored in the project. This could also have potentially served as a source of primitives for AI, but this alternative was not explored due to lack of time.

12.5 Conclusions

In the introduction to this dissertation (Chapter 1), I quoted part of the following:

> The distinctive concerns of software engineering are today [1995] exactly those set forth in Chapter 1 [of the original 1975 version of *The Mythical Man*

Month]:

- How to design and build a set of programs into a *system*
- How to design and build a program or a system into a robust, tested, documented, supported *product*
- How to maintain intellectual control over *complexity* in large doses.

. . . This complex craft will demand our continual development of the discipline, our learning to compose in larger units, our best use of new tools, our best adaptation of proven engineering management methods, liberal application of common sense, and a God-given humility to recognize our fallibility and limitations.

[Brooks, 1995, pp. 288–289]

BOD has been developed with these issues very much in mind. Most of this chapter has been a description of how to integrate BOD into two real-world, industrial projects. Both of these might be thought of as 'character based', though the tutor being built at Edinburgh has no conventional personality. However, *any* application that involves juggling priorities, chosing between possible actions, and / or regulating and interpreting multiple environments or contexts in parallel might profit from BOD design.

Chapter 13

Conclusions

In this dissertation I have presented Behavior-Oriented Design (BOD), a methodology for creating complex adaptive agents capable of addressing multiple, conflicting goals. BOD consists of an architecture and a design process. The architecture is modular; specialized representations are associated directly with code for acting and sensing. The specialized representations facilitate learning, the modularity facilitates design. Potential conflicts between the modules are resolved using reactive plans. The details of this reactive planning system are among the contributions of this dissertation, as is the literature research supporting the approach, and the experimental research demonstrating it.

The literature research is primarily presented in Chapters 3 (for artificial intelligence) and 11 (for natural intelligence). The experiments are primarily presented in Chapters 7 (simulated blocks world and real mobile robots), 9 (primate learning of transitive inference) and 10 (social behavior in primates). There are also references to working BOD systems not fully presented here in Chapters 4 and 12.

For more complete listings of contributions and chapters, please see Chapter 1. This last chapter concentrates on restating the most important lessons of this dissertation.

13.1 Design is Key to the Success of 'New AI'

One of the most important aspects of the reactive revolution of the late 1980's is often overlooked. The break-throughs in robotics associated with reactive and behavior-based systems are usually attributed to the loss of deliberate planning and/or explicit representations. The real contribution of the reactive paradigm was explained nearly a decade earlier: you can't learn something you don't practically already know [Winston, 1975], nor, by extension, plan something you can't nearly already do. The reason is simple combinatorics [Chapman, 1987, Wolpert, 1996b, McGonigle and Chalmers, 1998]. As evolutionary linguists and case-based reasoning researchers often try to tell us, what makes humans seem so much more intelligent that the other apes is not just our creative ability to plan, but our excellent methods of storing and transmitting solutions we manage to find [e.g. Hammond, 1990, Knight et al., 2000].

Reactive and behavior-based AI thus facilitate the advance of AI in two ways. First, by severely deprecating both planning and state (and consequently learning), the reactive approach increased by default the emphasis on one of the largest problems of AI and software in general: design. Second, the behavior-based approach made fashionable a proven software design methodology: modularity.

Yet the importance of human design for reactive systems still seems to be under-recognized. This is despite the extensive mention of design in the descriptions of the best known early architectures [e.g. Brooks, 1991b, Maes, 1990b]. Further, users tend to design agents by hand even for architectures like PRS [Georgeff and Lansky, 1987], which are intended to exploit productive planners, or like Soar [Newell, 1990], which are intended to learn.

Besides emphasizing the use of modularity, the behavior-based movement also made an important engineering contribution by emphasizing specialized learning [e.g Brooks, 1991b, pp. 158–9]. Specializing learning increases its probability of success, thus increasing its utility in a reliable agent. Similarly, modularity simplifies program design, at least locally, thus increasing the probability of correctness.

13.2 Learning is Important, but Not Sufficient

As the BOD emphasis on adaptive state and perception indicates, learning (or at least adaptation at many different time scales) is absolutely critical for an intelligent agent. However, a learning system alone is not enough to achieve AI. Even children, who are highly-honed learning machines, take years to acquire useful behaviors to any degree of proficiency. Consequently, even if building a child were within the ability of science and technology, any artificial system with a limited market window is best instilled from the beginning with as much knowledge as its designers can impart. Learning is necessary for giving an agent information its designer can't know in advance, such as the layout of its owner's home. It is also useful when it can save the designer development time. But the process of making learning work is a design process. Thus, even in learning systems, design is critical.

13.3 BOD Makes New AI Better

Behavior-oriented design maximizes the benefits of behavior-based AI by reemphasizing both the design process and modularity. BOD addresses the difficulties inherent in arbitrating between or coordinating behaviors using another specialized representation: POSH reactive plans. The critical aspects of POSH action selection are these:

- it supports basic reactive plans (BRPs), which allow for flexible though focussed action selection
- it limits stack growth and allows cycles in its hierarchy,
- it supports pseudo-parallelism and the changing of attention to higher priorities, and
- it restarts a plan hierarchy from its root if it terminates.

In BOD, POSH action selection provides all these features without eliminating either the autonomy or heterogeneity of the underlying behaviors. POSH plans communicate to the behaviors by an interface supported by the behaviors.

BOD leverages many of the design contributions of one of the most significant improvements in software engineering of the last 20 years: object-oriented design (OOD). For example, BOD addresses the BBAI issue of behavior decomposition (and the analogous issue of agent decomposition facing the MAS community) using the rule-of-thumb for object decomposition developed in the OOD community. Behavior decomposition can be determined by the adaptive requirements of the various primitive actions. BOD also heavily emphasizes iterative design.

The fact that BOD exploits OOD not only implies that libraries of behaviors can be easily and cleanly developed in any object-oriented language, but also that, in the future, BOD can continue absorbing the advances of the OOD community.

13.4 BOD Doesn't Require Changing Architectures

I have demonstrated that BOD can be used to greater or lesser extent with existing agent architectures. This means that BOD can improve existing projects, or new projects, that for whatever reason are implemented in other agent architectures or simply in object-oriented languages. So long as there is a neat way to express the critical idioms of action sequences and basic reactive plans, and so long as learning and behavior can be at least partly modularized, BOD can help keep agents simple and successful.

To get maximum benefit from the BOD development process, three conditions need to be met by the underlying architecture.

First, there needs to be reactive action selection capable of supporting three kinds of situations:

1. things that need to be checked all the time,
2. things that only need to be considered in particular contexts, and
3. things that reliably follow one from another.

POSH action selection supports these situations with drive collections, competences and action patterns, respectively.

Second, there should be ways to modularize the project's code and data, preferably with specialized representations for particular learning tasks. Further, there should be a way of tagging or grouping associated primitive actions and the data that supports them.

Finally, there should be enough structure to the development process that the designer or design team can come together regularly and reevaluate the current structure of the agent. This provides the opportunity for reexpressing the agent in a clearer way. Regular housekeeping is the only way to keep a project clean and coherent.

13.5 The Future

As I said in Chapter 1, my main motivation for conducting this research has been to create a platform on which I can build psychologically plausible agents and models. The motivation for me is building and exploring the agents and models themselves; BOD is just a necessary step in that process. I have no doubt that in the future both I and other people in the agent community will discover methodological insights that can significantly improve the current BOD process. I have already suggested (in Chapter 11) some additional mechanisms that may compliment its architecture. I hope that, as a community, we will be able to share these insights and improve our field.

Appendix A

POSH Code

This appendix shows the CLOS code for the current version of POSH action selection (see Section 4.6.3.) It also has additional code for making things work, like a GUI for debugging posh code. You may want to skip those sections and move to the next appendix and see some behavior libraries.

A.1 Basic POSH classes

This file defines everything but the drive collection.

```
;; posh.lisp

;; Joanna Bryson, June 2000, developed off the old edsol code


;; CONSTANTS
;; how long things stay on the bboard (for viewing) by default
;; 600000 is 10 minutes in harlequin on SGI or linux
;; 60000 is 10 minutes in lucid on suns
;; 2000 is 2 seconds (for running debugged code)
(defparameter DEFAULT-VIEWTIME 600000)
(defparameter DEBUG-SCHEDULE-TIME 600000)

#| Note1:  I am assuming a nil timeoutinterval indicates it never times out
   Note2:  If you change these, you must change the corresponding new-instance
           function(s)
|#

(defclass sol-generic ()
  (
   (name :accessor name :initarg :name :initform "name")
   ; preconditions should be easy to check quickly!
   (preconditions :accessor preconditions :initarg :preconditions
  :initform '())
   ; postactions post special messsages to the BBs on succesful completion
   ; Note this is *not* the mechanism for getting the default 'done, 'failed
   ; or 'canceled messages out; those are done by the schedulers.
   (postactions :accessor postactions :initarg :postactions :initform '())
   ; string name for blackboard
   (blackboardname :accessor blackboardname :initarg :blackboardname
   :initform "CB")
   ; The thing that really gets executed. Accessors defined below!
   (content :accessor content :initarg :content :initform nil)
   ; when absolutely this should be killed off the action scheduler
   (timeout :accessor timeout :initarg :timeout :initform -1) ;-1 is invalid
   ; for remembering chosen rep. if it came from a script file
   (timeout-rep :accessor timeout-rep :initarg :timeout-rep :initform nil)
   ; when relative to starting this should be killed off the action scheduler
   ; (timeoutinterval :accessor timeoutinterval :initarg :timeoutinterval
   ;                  :initform 1000) ;1s timeoutinterval
   (timeoutinterval :accessor timeoutinterval :initarg :timeoutinterval
    :initform DEBUG-SCHEDULE-TIME);10m t/o for debug
   ; this tells you what the ultimate root motivation for this behavior was
   (drive-name :accessor drive-name :initarg :drive-name :initform 'raw-prototype-no-
   ))

#|    GENERIC METHODS

(send-sol object content &optional timeoutinterval)   generate a solPut
(rec-sol object content &optional timeoutinterval)    generate a solGet
(ready object)                                  checks if preconditions are true
(fire object tim viewtime)                         execute the thing in object
(fire-postactions object tim viewtime result)      execute object's postactions
(new-instance object drive-name tim &key (postactions nil) (preconditions nil))
                           A new instance is what gets put on the schedule
                           when an action pattern is called.  See comment
                           at beginning of file.


(command object)                -|
(tag object)                     |
(action object)                  |    accessors into the content field
(value object)                   |
(timestamp object)              -|
                                 |

(find-bbitem-bb tag bb &optional result)     find a particular thing
(find-prereq-bb command tag bb) find a type (tag ~= action)
```

```
(bbitem-from-sol-generic ((sg sol-generic)          make a BB entry
  &key command timeout (timestamp nil))
|#

#|  GENERIC FUNCTIONS

These related to constructing datatypes...
[a bunch of accessors for "content" [argh!]]
(make-content &key command tag action (value nil) (timestamp -1))
(fix-content-action sg action)
(const-bbitem content timeout &optional (timestamp nil))

These two are really a method on one such type, a bbitem
(content-from-bbitem (bbi bbitem))
(sol-generic-from-bbitem (item bbitem))

These do important work
(check-bb &optional (viewtime DEFAULT-VIEWTIME))  maintain a bb --
                      transfer stuff to schedule, cancel outdated things
(sol-schedule (item bbitem) tim)        another bbitem method -- put a bbitem
                                        on the schedule
(action-scheduler &optional (viewtime DEFAULT-VIEWTIME))
                                             run through the schedule once,
                                             processing what you can, cancel
                                             outdated, etc.
(driven-action-scheduler drive-name  debug-stream &optional (viewtime))
                       action-scheduler but only attend to descendents of the
                       specified drive.
(reset)                            for debugging -- erase bboard and schedule
|#

;; this class is for sensing primitives -- always predicates
(defclass sol-sense (sol-generic)
  (
   ; this gets evalled when the sense is fired
   (sensep :accessor sensep :initarg :sensep :initform nil)
   ))

#|  SENSE FUNCTIONS

(fire object tim viewtime)                       execute the thing in object

|#
```

```
;; this class exists only to tag my collective classes below...
(defclass sol-edsol (sol-generic)
  (
   ; is this object currently active, or does it need to be activated?
   (active :accessor active :initarg :active :initform nil)
   ))

#|    SoL EDSOL METHODS

(cancel obj)       wipe out all childeren of obj on the schedule
(schedule-element object element prereqs postreqs tim)

|#

#|
  sol-action-pattern :
     a simple structure for encoding sequences or parallel sets of behaviors
     which is entirely triggered by a single environmental condition.  Once
     the pattern has started, it will complete in order unless there is a
     catestrophic failure of an element.  In this case, the remaining pattern
     dies.

Notes: elements are listed in firing order.  An element looks like an
       invocation: a list of a function name and any arguments it takes.
       If a set of elements are again clustered in a list, then they
       are fired in parallel.
       Examples: ((a) (b) (c)) -> sequence a, b, c.
                 (((a) (b)) (c)) -> a & b, then c
       Elements of a sequence should normally return a true value.
       False (nil) will cause the sequence to terminate, but the
       intention of an action pattern is that it should be fired as one
       unit, without cognitive checking between elements. (In fact,
       a failed action pattern might be the kind of exception that
       grabs cognitive attention, see Norman&Shallice.) Action-patterns
       are usually only aborted for failed pre-conditions of elements,
       rather than because an element's execution was supervised and found
       wanting.  For a flexible system that examines external state
       between chosing elements, see sol-competence.
|#

(defclass sol-action-pattern (sol-edsol)
  (
  (elements :accessor elements :initarg :elements :initform '())
  ; this is for remembering the printed version esp. senses
  (ap-guts :accessor ap-guts :initarg :ap-guts :initform '())
  ))

#|    SOL-ACTION-PATTERN METHODS

(fire object tim viewtime)  just calls activate for this class
(activate object tim)      create and schedule children / elements
                           "activate" is run from the scheduler, like "fire"
(new-instance object drive-name tim &key (postactions nil) (preconditions nil))
                           A new instance is what gets put on the schedule
                           when an action pattern is called.  See comment
                           at beginning of file.

check-parse    check all elements exist (run on roots after scanning
               a newly written script file) -- debugging tool
|#

#|
  sol-competence :
     provides for "reactive plans" --- prioritized sets of procedures
     or subgoals for attaining a particular goal, each with preconditions
     which determine their applicability.

Notes: I provide a log so you can tell how well a competence is doing as
       a whole.  Beginnings are labeled 'active, ends are labeled 'done
       if the goal was passed, or 'fail if no childeren were triggered.
       It's also possible control simply passed to one of the children, in
       which case there will be no terminating symbol in the log.
       This code doesn't use the log for anything, but you might.

       Triggers are for doing more complicated things than are
reasonable to send through the blackboards.  In other words, you may
never want to use them: it depends what abstraction level your
primitives are at.  But I prefer keeping the cognitive (? SOL anyway
:-) book- keeping seperate from the real / active perception.  |#

#|
competence-element...
Trigger is an object (often an action-pattern) or nil, see note above.
Element is fired if trigger succeeds, trigger is tested if
preconditions succeed and retries does not equal 0.  Tries is an
integer, the number of times this element gets to fire before it
fails.  Elements with tries < 0 never fail.
```

```
I think this logging stuff is too slow, and I'm not using it right
now... it should be composited, not stored raw.  JJB Jan 01
(defstruct competence-log command tag timestamp)
|#

(defclass sol-element ()
  (
  ; can this element fire?
  (trigger :accessor trigger :initarg :trigger :initform nil)
  ; what drive does it belong to? (at least when executing
  (drive-name :accessor drive-name :initarg :drive-name :initform 'not-assigned-sol-e
  ))

(defclass competence-element (sol-element)
  (
  (ce-label :accessor ce-label :initarg :ce-label :initform 'fix-me-cel)
  ; action is what actually gets executed
  (action :accessor action :initarg :action :initform nil)
   ; this determines how many times an element can be tried, ignored if neg.
  (retries :accessor retries :initarg :retries :initform -1)
  (competence :accessor competence :initarg :competence
      :initform 'not-my-competence)
  ))

(defclass sol-competence (sol-edsol)
  (
  ; Elements are listed in order of priority (highest first).
  (elements :accessor elements :initarg :elements :initform '())
  ; goal used to be the first element of a competence (with reward 100)
  ; in old edmund --- here I stick it in its own slot, but it should still
  ; be a competence-element.
  (goal :accessor goal :initarg :goal :initform nil)
  ; this is a place to swap real preconditions while waiting on a child
  (temp-preconditions :accessor temp-preconditions :initarg
      :temp-preconditions :initform nil)
  ; logbook is for noticing if a strategy isn't working, could also be handled
  ; elsewhere in episodic memory, but this allows ACT-R like learning or simple
  ; rules on retrying. (see note above JJB Jan 01)
  ; (logbook :accessor logbook :initarg :logbook :initform nil)
  ))

#|    COMPETENCE METHODS

ready          like regular ready, but checks triggers too
activate       put on the scheduler
```

```
deactivate     leaving the scheduler
fire           run

(new-instance object drive-name tim &key (postactions nil) (preconditions nil))
                           A new instance is what gets put on the schedule
                           when an action pattern is called.  See comment
                           at beginning of file.

check-parse    check all elements exist (run on root after scanning new script file)
truncate-log   remove log entries older than a particular timestamp.
|#

#|
  drive-collection :
     provides the root of behavior hierarchies: these are persistant
     decision modules that are always active, determining when they
     should create behavior.  I'm actually not modelling them on the
     action scheduler, but rather as the outer loop for the entire
     structure.

     There are actually two kinds of drives.  Regular drive-collections
     treat a frequency as a number of passes / (turns that are given up)
     between executions.  This is only useful for discrete time-step
     simulations.  Real-time-drive-collections consider their frequencies
     to be in hertz, and fire based on how much time has passed since the
     previous execution.
|#

; note "drive-name" for the name is defined in sol-element.
(defclass drive-element (sol-element)
  (
   ; this should hold an SG for firing when nothing else is being done by
   ; this drive.
   (drive-root :accessor drive-root :initarg :drive-root :initform 'fixme-de)
   (drive-root-rep :accessor drive-root-rep :initarg :drive-root-rep
   :initform 'fixme-dep)
   ; this determines how many times an element can be tried, ignored if neg.
   (frequency :accessor frequency :initarg :frequency :initform -1)
   ; this is for remebering how to write the frequency out
   (frequency-rep :accessor frequency-rep :initarg :frequency-rep
  :initform nil)
   ; this is for figuring out if it's time to fire now
   (last-fired :accessor last-fired :initarg :last-fired :initform 0)
   (drive-collection :accessor drive-collection :initarg :drive-collection
     :initform 'not-my-drive-collection)
   ))

(defclass drive-collection (sol-edsol)
  (
  ; Elements are listed in order of priority (highest first).
  (elements :accessor elements :initarg :elements :initform '())
  ; Generally, this should be nil, as you kill the agent if you let the
  ; drive-collection terminate.  But I keep it for non-persistant agents.
  (goal :accessor goal :initarg :goal :initform nil)
   ))

(defclass real-time-drive-collection (drive-collection)
  (
   ; this just treats "frequency" differently
   ))

#|    DRIVE METHODS

are actually all in another file --- see posh-run.lisp

|#

;;; Generic Methods

;; This structure is for what lives on the bulletin board --- it's
;; based around the "content" section of the sol-message.  This all
;; assumes content is like '(command (tag drive-name action value*) [timestamp])
;; where value* is any number of values, and [timestamp] is optional.
;; This has evolved a bit from the specification, but I found it
;; necessary to cart all that information around.

(defstruct bbitem command tag drive-name action value timestamp timeout)

;; Command can be one of '('request 'done 'cancel 'pending 'fail 'active)
#|
 'request is when something should be serviced, in my system this
  means needing to be scheduled.
 'done is a notification of an ordinary completion.
 'cancel means the item was canceled and removed from the schedule
  before completing -- this is done by timeout, or could be done with
  the "cancel" method.
 'fail means the command failed, usually some form of radical failure
  or else feedback from a perceptual system.
 'pending means its on the scheduler but hasn't been gotten to (not used
  in my implementation so far, but may help with  given expected delays)
 'active is for edsol / plan  meta-modules; while they are supervising
  their offspring they stay on the scheduler.
|#

;; fake accessors to get stuff from content (argh!  FIXME) Nasty accessors
;; replicated a bit just below....
(defmethod command ((sg sol-generic))
  (car (content sg)))
(defmethod tag ((sg sol-generic))
  (caadr (content sg)))
(defmethod action ((sg sol-generic))
  (cadadr (content sg)))
(defmethod value ((sg sol-generic))
  (cddadr (content sg)))
(defmethod timestamp ((sg sol-generic))
  (caddr (content sg)))

(defun make-content (&key command tag action (value nil) (timestamp -1))
  (unless (listp value) (setf value (list value)))
  (if (> 0 timestamp)
      (list command (cons tag (cons action value)))
      (list command (cons tag (cons action value)) timestamp))
  )

(defmethod fix-content-action ((sg sol-generic) action)
  (setf (cadadr (content sg)) action))

;; construct a bbitem from a content string
(defun const-bbitem (content drive-name timeout &optional (timestamp nil))
  (make-bbitem :command (car content) :tag (caadr content)
        :drive-name drive-name :action (cadadr content)
        :value (cddadr content) :timeout timeout
        :timestamp (if timestamp timestamp (caddr content)))
  )

;; construct content from bbitem
(defmethod content-from-bbitem ((bbi bbitem))
  (make-content :command (bbitem-command bbi)
                :tag (bbitem-tag bbi)
                :action (bbitem-action bbi)
:value (bbitem-value bbi)
:timestamp (bbitem-timestamp bbi)))

; (tag sg) for a competence element is the name of the competence
; element (name sg) is the name of the action.  Wasn't a problem when
; actions were just names, not functions, but now is unclear FIXME.
(defmethod bbitem-from-sol-generic ((sg sol-generic)
    &key command timeout (timestamp nil))
  (make-bbitem :command  command :tag (tag sg)
       :drive-name (drive-name sg)
       :action (action sg) :value (value sg)
       :timestamp (if timestamp timestamp (get-internal-real-time))
       :timeout timeout)
  )

(defmethod sol-generic-from-bbitem ((item bbitem))
  (make-instance 'sol-generic
 :name (bbitem-tag item)
                 :content (content-from-bbitem item)
 :drive-name (bbitem-drive-name item)
                 :timeout (bbitem-timeout item))
  )

(defmethod sol-generic-from-function ((func function)
      &optional (name 'anonymous-function))
  (make-instance 'sol-generic
 :name name
                 :content   (make-content
     :command 'request      ; sort of
     :tag name
     :action func
     :value nil))
  ) ; defmethod sol-generic-from-function

;; FINALLY, the new/local versions of send-sol and rec-sol

; only assume real time sent if real-time.  Should really fix all of
; posh.lisp to deal with non-real-time.  Notice, timeoutinterval has
; to be based on real time, not tim!
(defmethod send-sol ((sg sol-generic) content tim
     &optional (timeoutinterval DEFAULT-VIEWTIME))
  (declare (special bb))
  (push
   (const-bbitem
    content (drive-name sg) (+ (get-internal-real-time) timeoutinterval) tim)
   bb)
  )

; time doesn't matter here, only the tag in the content (see find-bbitem-bb)
; also ignoring drive -- not sure that's right
```

```
(defmethod rec-sol ((sg sol-generic) content tim &optional
    (timeoutinterval -1))
  (declare (special bb))
  (let ((cont (const-bbitem
       content (drive-name sg) timeoutinterval tim)))
    (find-bbitem-bb (bbitem-tag cont) bb))
  )

; this returns everything with a particular tag
(defun find-bbitem-bb (tag bb &optional (result nil))
  (let ((item (car bb)))
    (cond
     ((null item)
      result)
     ((eq (bbitem-tag item) tag)
      (push item result) (find-bbitem-bb tag (cdr bb) result))
     (t
      (find-bbitem-bb tag (cdr bb) result))))
  )

; Ymir has pre-labelled conditions for pre-reqs, but here I'm
; auto-generating things that aren't itemized in advance.

;; command is the same as in content / bbitem (see comments above)
(defstruct prereq command tag)

; find whether a single prereq is on a bboard
(defun find-prereq-bb (command tag bb)
  (let ((item (car bb)))
    (cond
     ((null item)
      nil)
     ((and (eq (bbitem-command item) command)
   (eq (bbitem-tag item) tag))
      item)
     (t
      (find-prereq-bb command tag (cdr bb)))))
  )

; find out if all an object's prereqs are happy
(defmethod ready ((sg sol-generic))
  (do* ((prereqs (preconditions sg) (cdr prereqs))
(thisreq (car prereqs) (car prereqs)))
       ((null thisreq) t)  ;; if done then return true
    (declare (special bb))
    (unless
(find-prereq-bb (prereq-command thisreq) (prereq-tag thisreq) bb)
      (return nil)))
  )

;;; Fake Ymir Mechanism Methods

; check-bb trims bb (send-sol adds to it) and also sends things to the
; action scheduler via the schedule.  "viewtime" is how long a message
; from this stays on the bb, default is 10 minutes.  Note that this is
; trimmed in real-time, regardless of whether the agent is running in
; real time.
(defun check-bb (&optional (viewtime DEFAULT-VIEWTIME))
  (declare (special bb))
  (do ((unexamined (cdr bb) (cdr unexamined))
       (item (car bb) (car unexamined))
       (tim (get-internal-real-time)))
      ((null item) bb)
    (cond
     ((and (bbitem-timeout item)              ; If has a timeout and...
           (> tim (bbitem-timeout item)))      ;   if it timed-out, drop...
      (if (eq (bbitem-command item) 'request)  ;   unless must show cancel.
          (push (make-bbitem
                 :command  'cancel :tag (bbitem-tag item)
                 :action (bbitem-action item)
 :drive-name (bbitem-drive-name item)
                 :value (bbitem-value item) :timestamp tim
                 :timeout (+ tim viewtime))
                bb))
      (setf bb (delete item bb :count 1)))
     ((or (eq (bbitem-command item) 'request)  ; If command, process...
          (eq (bbitem-command item) 'pending)) ;   and remove request.
      (sol-schedule item tim)
      (setf bb (delete item bb :count 1)))
     (t t)))                                   ; Otherwise, ignore
  ) ; defun check-bb

; The schedule is a list of objects to be executed... notice the
; "schedule" will keep getting changed as the scheduler runs over it!

; Put something on the schedule (convert it to a sol-generic object
; first if it isn't already.)  This goes for schedule-element as well!
; Changed June '00 -- no longer passing names around but objects;
; no longer have names defined in environment.  Will new instances make
; too much garbage?
```

```
(defmethod sol-schedule ((item bbitem) tim)
  (declare (special schedule))
  (let ((act (bbitem-action item)))
    (cond
     ((typep act 'sol-generic)     ; If a subclass of s-g, then... push a copy
      (push (new-instance act (bbitem-drive-name item) tim) schedule))
     ((or (typep act 'function) (typep act 'method)) ; If reg. func call then
      (push (sol-generic-from-bbitem item) schedule))   ; ...build an object.
     (t                                         ; Else complain
      (error (format nil "Unknown object type passed to sol-schedule: ~s"
     item))))
    )) ; defmethod sol-schedule

; and this executes them.  Notice some things (like drives and competences)
; should stay on the schedule, even though they've already been executed.
; They should just have lower priority for a while.

(defun action-scheduler (&optional (viewtime DEFAULT-VIEWTIME))
  (declare (special schedule))
  (do ((unexamined (cdr schedule) (cdr unexamined))
       (obj (car schedule) (car unexamined))
       (tim (get-internal-real-time)))
      ((null obj) schedule)
    (cond
     ((and (timeout obj)                         ; If has a timeout and...
           (> tim (timeout obj)))                ;   if it timed-out, cancel.
      (send-sol obj (make-content
     :command  'cancel :tag (tag obj)  :action (action obj)
     :value (value obj) :timestamp tim) tim viewtime)
      (setf schedule (delete obj schedule :count 1)))
     ((ready obj)                                ; Else, if it's ready, fire.
      (unless (equal (fire obj tim viewtime) 'preserve)
        (setf schedule (delete obj schedule :count 1))))
     (t t)))
  ) ; defun action-scheduler

; this is the action scheduler modified to work with drives.  This
; preserves info on drives not currently attended to, unless attention
; switches long enough that the behaviors time out.  The return value
; indicates whether anything currently on the scheduler belongs to the
; chosen drive. If not, the calling function will add the drive's root
; to the schedule, which will get executed on the next cycle.
(defun driven-action-scheduler (drive-name  debug-stream
&optional (debug-action nil)
(debug-non-action nil)
(viewtime DEFAULT-VIEWTIME))
  (declare (special schedule))
  (unless debug-stream (setf debug-action nil debug-non-action nil))
  (do ((unexamined (cdr schedule) (cdr unexamined))
       (obj (car schedule) (car unexamined))
       (tim (get-internal-real-time))
       (found-one nil))                  ; flag whether need restart from root
      ((null obj) found-one)
    (cond
     ((and (timeout obj)                         ; If has a timeout and...
           (> tim (timeout obj)))                ;   if it timed-out, cancel.
      (send-sol obj (make-content
     :command  'cancel :tag (tag obj)  :action (action obj)
     :value (value obj) :timestamp tim) tim viewtime)
      (setf schedule (delete obj schedule :count 1)))
     ((and (drive-member obj drive-name)      ; Else, if it's the right drive
      (ready obj))                            ;         and it's ready, fire.
      (setf found-one t)
      (if debug-action
  (format debug-stream "~% Firing action ~s (for drive ~s)"
  (name obj) drive-name))
      (unless (equal (fire obj tim viewtime) 'preserve)
        (setf schedule (delete obj schedule :count 1))))
     (t                                       ; not firing, but not old
      (if debug-non-action
  (if (drive-member obj drive-name)
      (format debug-stream "~% Action ~s not ready (for drive ~s)"
      obj drive-name)
    (format debug-stream "~% Action ~s wrong drive (for drive ~s)"
    (name obj) drive-name)))
      t)))
  ) ; defun driven-action-scheduler

; (defmacro drive-member (sg drive-name)
;   `(eq (drive ,sg) ,drive-name))
; s/b the above, but while debugging...
(defun drive-member (sg drive-name)
  (unless (drive-name sg)
    (error (format nil "Missing drive in object called ~s, ~s"
   (name sg) sg)))
  (eq (drive-name sg) drive-name))

; fire evaluates the action (which should be enough to run it) and then
; notes and records the return value.  This winds up on the bboard.
; If the return value was true, then postactions are also fired.
; Changed June 2000 to accept function action -- JB
(defmethod fire ((obj sol-generic) tim &optional (viewtime DEFAULT-VIEWTIME))
  (declare (special bb))
  (let ((won (apply (action obj)
 (value obj))))
    (if won
(progn (push (make-bbitem
      :command  'done :tag (tag obj) :action (action obj)
      :drive-name (drive-name obj)
      :value won :timestamp tim :timeout (+ tim viewtime))
     bb)
       (fire-postactions obj tim viewtime 'done))
      (progn (push (make-bbitem
    :command  'fail :tag (tag obj) :action (action obj)
    :drive-name (drive-name obj)
    :value won :timestamp tim :timeout (+ tim viewtime))
   bb)
     (fire-postactions obj tim viewtime 'fail)))
    won)) ; fire sol-generic


; I use these for letting the rest of a meta structure know what's going
; on / who's turn it is.
(defmethod fire-postactions ((obj sol-generic) tim viewtime result)
  (do* ((reqs (postactions obj) (cdr reqs))
(thisreq (car reqs) (car reqs)))
       ((null thisreq) t)  ;; if done then return true
       (if (eq (prereq-command thisreq) 'result)
   (send-sol obj (make-content :timestamp tim
  :command result :tag (prereq-tag thisreq)
  :action (prereq-tag thisreq)) tim viewtime)
 (send-sol obj (make-content :timestamp tim
:command (prereq-command thisreq)
:tag (prereq-tag thisreq) :action (prereq-tag thisreq))
  tim  viewtime)))
  )

;; useful to programers debugging, though not used in the code
(defun reset ()
  (declare (special schedule bb))
  (setf schedule nil)
  (setf bb nil))


;;; This used to be a sol-edsol methods, but I can't see a good reason, and
;;; made it generic --- Joanna, Jan 1999
; there was a "break" here that didn't seem to be triggering (though I
; didn't check hard) -- JB Nov 2000

; new instances keep the old tag, so you can follow what's happening...
(defmethod new-instance ((ap sol-generic) drive-name tim
 &key (postactions nil) (preconditions nil))
  (make-instance 'sol-generic
 :blackboardname (blackboardname ap)
 :drive-name drive-name
 :name (name ap)
 :postactions (append (postactions ap) postactions)
 :preconditions (append (preconditions ap) preconditions)
 :timeoutinterval (timeoutinterval ap)
 :timeout (+ tim (timeoutinterval ap))
 ; add a plus for debugging...
 ; :content (make-content :tag (list (tag ap) '+)
 :content (make-content :tag (tag ap)
:command 'instance
:action (action ap)
:value (value ap)
:timestamp tim))
  ) ; defmethod new-instance sol-generic



;; I don't use this, but you might...
(defmethod cancel ((sg sol-generic))
  (declare (special schedule))
  (let ((tag (tag sg)))
    (do ((oldsched (cdr schedule) (cdr oldsched))
 (item (car schedule) (car oldsched))
 (newsched '()))
((null oldsched) (setf schedule newsched))
      (unless (eq tag (tag item)) (push item newsched))
      )))

;;; Sol-Edsol Methods

; notice an action-pattern's "children" have the same tag as the AP, so they
; can easily all be identified.   See comments in "sol-schedule".
; notice also that its important that this makes a new instance, so that
; modifications to prereqs and postreqs aren't permanent!
; Also modified June '00, see sol-schedule above.
; This returns the modified schedule.
; This only ever seems to be called on functions or methods, thus
; new-instance sol-generic never gets called --- July '00
(defmethod schedule-element ((ap sol-edsol) element prereqs postreqs tim)
  (declare (special schedule))
  (cond
   ((or (typep element 'function)      ; If function or method call then...
(typep element 'method))
    (push (make-instance 'sol-generic
 :name (tag ap)
 :content (make-content :command 'request
;;tag (list (tag ap) 'el);dbug
:tag (tag ap)
:action element
:value (value ap)
:timestamp tim)
 :drive-name (drive-name ap)
 :preconditions prereqs  :postactions postreqs
 :timeout (+ tim (timeoutinterval ap))
 :timeoutinterval (timeoutinterval ap))
  schedule))
   ((typep element 'competence-element)           ; make a new one!
    (push (new-instance (action element) (drive-name ap)
tim :preconditions prereqs
:postactions postreqs)   schedule))
   ((typep element 'sol-generic) ; If a subclass of edsol, then...
    (push (new-instance element (drive-name ap) tim
:preconditions prereqs
:postactions postreqs)    schedule))
   (t                                  ; Else complain
    (error (format nil "Unknown object type in schedule-element: ~s"
   element)))
   )) ; defmethod schedule-element


;;; Sol-Sense Methods
(defmethod fire ((obj sol-sense) tim &optional (viewtime DEFAULT-VIEWTIME))
  (declare (special bb))
  (let ((won (eval (sensep obj))))
    (if won
(progn (push (make-bbitem
      :command  'done :tag (tag obj) :action 'sensed
      :drive-name (drive-name obj)
      :value won :timestamp tim :timeout (+ tim viewtime))
     bb)
       (fire-postactions obj tim viewtime 'done))
      (progn (push (make-bbitem
    :command  'fail :tag (tag obj) :action 'sensed
    :drive-name (drive-name obj)
    :value won :timestamp tim :timeout (+ tim viewtime))
   bb)
     (fire-postactions obj tim viewtime 'fail)))
    won)) ; fire sol-sense



;;; Sol-Action-Pattern Methods

; Action pattern objects are only on the schedule once, then they leave
; their kids orphans.  Consequently, all this has to do is activate the
; object.
(defmethod fire ((ap sol-action-pattern) tim &optional (viewtime DEFAULT-VIEWTIME))
  (activate ap tim))

; each element of a sequence has the previous element(s) as preconditions, and
; declares itself finished as a postcondition.  A list within an element list
; indicates a number of elements to go parallel: these share their prereq, and
; each contributes to the next sequence step's prereq.
(defmethod activate ((ap sol-action-pattern) tim)
  (send-sol ap (make-content :command 'active :tag (tag ap)
     :action (action ap)
     :value (value ap)) tim (timeoutinterval ap))
  (do ((el (car (elements ap)) (car els))
       (els (cdr (elements ap)) (cdr els))
       (sym)
       (next-prereqs nil nil)
       (prereqs nil next-prereqs))
      ((null el) t)
    (cond
     ((listp el)      ; for action-pattern, list indicates parallel chunk
      (do ((ell (car els) (car ells))
           (ells (cdr els) (cdr ells)))
          ((null ell))
        (setf sym (list (tag ap) (gensym)))
        (schedule-element ap ell prereqs ; and postreq (below)
                          (list (make-prereq :command 'result :tag sym)) tim)
        (push (make-prereq :command 'done :tag sym) next-prereqs)
        ))
     (t
      (setf sym (list (tag ap) (gensym)))
      (schedule-element ap el prereqs ; and postacts (below)
                        (list (make-prereq :command 'result :tag sym)) tim)
      (push (make-prereq :command 'done :tag sym) next-prereqs)
      )))
  ); defmethod activate sol-action-pattern

; this creates a new (and temporary) instance of a named action-pattern,
; for putting on the scheduler.  Notice it has its own tag and timeout.
(defmethod new-instance ((ap sol-action-pattern) drive-name tim
 &key (postactions nil) (preconditions nil))
  (make-instance 'sol-action-pattern
 :blackboardname (blackboardname ap)
 :drive-name drive-name
 :elements (copy-elements ap drive-name)
 :name (name ap)
 :postactions (append (postactions ap) postactions)
 :preconditions (append (preconditions ap) preconditions)
 :timeoutinterval (timeoutinterval ap)
 :timeout (+ tim (timeoutinterval ap))
 ; add a plus for debugging...
 ; :content (make-content :tag (list (tag ap) '+)
 :content (make-content :tag (tag ap)
:command 'active
:action (action ap)
:value (value ap)
:timestamp tim))
  ) ; defmethod new-instance sol-action-pattern


;; ARGH -- fix me!  The spec allows for there to be parallel elements
;; here, sort of like the priority levels in competences, so I try to
;; do the same here, but posh-script doesn't actually support this, so
;; now I'm in a rush and haven't really checked this...
;; Returns the copy.
(defmethod copy-elements ((ap sol-action-pattern) drive-name)
  (let ((elist (copy-list (elements ap))))
    (do* ((els elist (cdr els))
  (first-el (car els)    ; individual priority list
    (car els)))
         ((null els) elist)
       (if (listp first-el)
 (progn
   (setf first-el (copy-list first-el))
   (setf (car els)
     (do ((pls first-el (cdr pls)))
 ((null pls) first-el)
```

```
       (if (typep (car pls) 'sol-sense)
   (setf (car pls)
     (make-instance 'sol-sense
      :sensep (sensep (car pls))
      :drive-name drive-name)))
       ;else action/function done in copy-list
       ))) ; set the copied list for one priority
 (if (typep (car els) 'sol-sense)                  ; else not listp
   (setf (car els)
 (make-instance 'sol-sense
      :sensep (sensep (car els))
      :drive-name drive-name))))
       ))
  ); defmethod copy-elements sol-competence




;;; Competence Methods

; First, check a competence-element is ready.  This is a lot like
; previous readys, but also checks triggers if prereqs met...
(defmethod ready ((compel competence-element))
  (if
      (do* ((reqs (preconditions (action compel))
                  (cdr reqs))
    (thisreq (car reqs) (car reqs)))
           ((null thisreq) t)  ;; if done, get on to trigger...
 (declare (special bb))
         (unless
    (find-prereq-bb (prereq-command thisreq) (prereq-tag thisreq) bb)
          (return nil)))
      (if (trigger compel)
          (trigger (trigger compel))
        t)
    nil)
  ) ; defmethod ready competence-element

; triggers should either be regular sol-generics, in which case you just
; fire them, or else simplifed action-patterns, which get fired rapidly,
; without going through the scheduler like regular action patterns.  Maybe
; they should be another class, but I can't figure out how to do the type
; checking anyway, so I think I'll just leave that for the GUI.

(defmethod trigger ((trig sol-generic))
 (fire trig (get-internal-real-time)))

(defmethod trigger ((trig sol-action-pattern))
  (do ((el (car (elements trig)) (car els))
       (els (cdr (elements trig)) (cdr els)))
      ((null el) t)
    (unless ; should be a            sense   or an    act      (function)
(if (typep el 'sol-generic) (trigger el) (apply el nil))
      (return nil))))

; the inverse of sol-action-pattern -- here activate does next to nothing,
; though notice the copy of preconditions (since that gets hacked in "fire")
(defmethod activate ((comp sol-competence) tim)
  (send-sol comp (make-content :command 'active :tag (tag comp)
       :timestamp tim
       :action (action comp) :value (value comp))
    tim (timeoutinterval comp))
  (setf (active comp) t)
  (setf (temp-preconditions comp) (preconditions comp))
  ; (push (make-competence-log :command 'active :tag (tag comp) :timestamp tim)
; (logbook comp)) ; JJB Jan 01
  )

(defmethod deactivate ((comp sol-competence) command tim)
  (send-sol comp (make-content :command command :tag (tag comp)
       :timestamp tim
       :action (action comp) :value (value comp))
   tim (timeoutinterval comp))
  (setf (active comp) nil)
  ; (push (make-competence-log :command command :tag (tag comp) :timestamp tim)
; (logbook comp)) ; JJB Jan 01
  )

; because scheduler will have already fired this, we have to add it
; back on to the schedule until it's done... so add both your chosen
; element and yourself, with post and preconditions set up so the
; element runs first.  "temp-preconditions" is the regular
; preconditions for the competence (see "activate" above)
; Notice this requires that each element 1) has it's name in its
; core element and 2) stands as an identity with that same name in
; the global environment.  This is messy, but the best thing I can
; think of to match between Edmund's need to keep track of the number
; of times an individual element has been triggered, and Ymir's use of
; the environment.
; Modified July '00 to deal with priority levels --- but not cleverly!
; I could randomize within priority, but normally only one should be ready
; anyway.
(defmethod fire ((comp sol-competence) tim &optional viewtime)
  (unless (active comp) (activate comp tim)) ; Activate if not already active.
  (if (ready (goal comp)) (deactivate comp 'done tim) ;Done if goal met,...
    (do ((pl (car (elements comp)) (car pls))        ; otherwise, start an el.
 (pls (cdr (elements comp)) (cdr pls)))      ; this checks priorities
((null pl) (deactivate comp 'fail tim))
      (let ((result
     (do ((el (car pl) (car els))    ; this checks individual elements
  (els (cdr pl) (cdr els)))
 ((null el) nil)
       (if (and (ready el) (/= 0 (retries el)))
   (return (fire-cel el comp tim viewtime)))
       )))
(if result (return result))))
    )
  ); defmethod fire competence

; this puts the element on the schedule, and returns 'preserve to keep
; the parent on the schedule too.  It also sets a key for the child to
; fire as a postreq, so that at least this copy of the parent won't be
; reactivated 'til the child is done.
(defmethod fire-cel ((el competence-element) (comp sol-competence)
     tim &optional viewtime)
  (let ((symb (list (tag comp) (gensym))))
    (setf (retries el)
  (- (retries el) 1))
    (schedule-element
     comp (action (action el)) nil  ; just pass element?
     (list (make-prereq :command 'done :tag symb)) tim)
    (setf (preconditions comp)
  (cons (make-prereq :command 'done :tag symb)
(temp-preconditions comp)))
    'preserve
     )) ; defmethod fire-cel

; this creates a new (and temporary) instance of a named competence
; for putting on the scheduler.  Notice it has its own tag and timeout.
(defmethod new-instance ((comp sol-competence) drive-name tim
 &key (postactions nil) (preconditions nil))
  (make-instance 'sol-competence
 :blackboardname (blackboardname comp)
 :drive-name drive-name
 ; WARNING! if elements become structures, copy-tree won't do
```

```
                 ; a deep enough copy!  Write another method...
 :elements (copy-elements comp drive-name)
 :name (name comp)
 :postactions (append (postactions comp) postactions)
 :preconditions (append (preconditions comp) preconditions)
 :timeoutinterval (timeoutinterval comp)
 :timeout (+ tim (timeoutinterval comp))
 ; add a plus for debugging...
 ; :content (make-content :tag (list (tag comp) '+)
 :content (make-content :tag (tag comp)
:command 'active
:action (action comp)
:value (value comp)
:timestamp tim)
 :goal (goal comp)
 ; :logbook (logbook comp) ; JJB Jan 01
 :temp-preconditions (temp-preconditions comp))
  ) ; defmethod new-instance sol-competence

; notice, it's important to make new elements, so that the retries
; stand on their own!  the copy-list creates the basic structure,
; preserving order (and therefor priority!)  Changed July '00 to
; reflect extra nesting for shared priority.  Also to carry drive
; label through from motivating drive, not from exemplar.
(defmethod copy-elements ((comp sol-competence) drive-name)
  (let ((elist (copy-list (elements comp))))
    (do* ((els elist (cdr els))
  (plist (copy-list (car els))    ; individual priority list
 (copy-list (car els))))
         ((null els) elist)
       (setf (car els)
     (do ((pls plist (cdr pls)))
 ((null pls) plist)
       (setf (car pls)
     (make-instance 'competence-element
      :trigger (trigger (car pls))
      :drive-name drive-name
      :action (action (car pls))
      :retries (retries (car pls))))
       )) ; set the copied list for one priority
       )) ; set the copied list of elements
  ); defmethod copy-elements sol-competence

;;; Sol-Drive Methods
```

```
; this needs a serious re-think --- In the context of SoL, are the
; drives really like a competence, or not?

; [July 2000]
;Current answer --- NOT!  (But we're not really in the SoL context)
;See posh-run.lisp The only one of the above competence methods which
;is needed by the drive is "ready".

(defmethod ready ((compel drive-element))
  (if (trigger compel)
      (trigger (trigger compel))
    t)
  ) ; defmethod ready competence-element
```

A.2 Running POSH

Unlike the previous version of POSH, here drive collections are managed seperately from the rest of the execution. This file has the main control loops for the POSH action selection cycle.

```
;; posh-run.lisp -- the main control loop for the posh system,
;; including a lot of the stuff that makes drives work.


#|
  We expect the particular behavior library to define init-world,
  debug-world (which gets called when *debug* is true), and
  handle-bod-gui-requests (which gets called if *gui* is true).

  (defun init-world () ...
  (defun debug-world (output-stream time) ...
  (defun handle-bod-gui-requests (input-stream) ...
|#


; still a true global -- although set by

;;; variables about how things should run.  These relate to the GUI,
;;; and are only used in the code below when shielded with a parameter
;;; to check if this is run is connected to the GUI.
(defvar *debug* t)
(defvar *debug-world* t "print output from (debug-world)?")
(defvar *debug-drive*  nil "show which drive has been selected?")
(defvar *debug-failed-action* nil "shows things that didn't get fired.")
(defvar *debug-action* nil  "show which element of the drive is fired.")

#|
  There are two kinds of time -- 1) simulated time (like for Tyrrell's
  AS or nethack) where each cycle advances the clock one tick 2)
  real-time where time follows the internal clock.

  Thought about checking time from center for MAS, but this seems
  over-complex  JB Nov 2000
|#



; although real-time should be the same across MAS, it's purpose as a
; global is just to communicate to the GUI
(defvar *bb-stream* nil)
(defvar *as-stream* nil)
(defvar *posh-output* t)

; (require "comm")  ; for socket communication -- doesn't seem to work when
; compiling though, so have to do in advance... see astart.lisp

;; for calling posh from the GUI -- need to keep the stream open...
(defun posh-from-gui (fast-p to-gui-stream from-gui-stream)
  (unwind-protect
      (if fast-p (real-posh :fast-p t :drive-collection *drive-collection* :init-worl
        (real-posh :gui-p t :posh-output to-gui-stream
    :from-gui from-gui-stream :drive-collection *drive-collection*
    :bb-stream *bb-stream* :as-stream *as-stream*
    :debug-world (if (and (fboundp 'debug-world) *debug-world*)
     #'debug-world nil)
    :init-world (if (fboundp 'init-world) #'init-world nil)))
    ; to-gui-stream is attached to GUI, don't close it!
    (close from-gui-stream)))

; for backwards compatability with my memory
(defun posh (planname)
  (real-posh :plan-name planname :posh-output t :bb-stream t :as-stream t))

#|
;;; Believe it or not, these are the essence of real-posh

   We might also want a fast version of driver && driven-action-scheduler
|#
```

```
(defun fast-posh (drive-collection)
  (do ((result nil) (time 0))
      ((or (eq result 'drive-lost) (eq result 'drive-won)) result)
    (setf time (+ time 1))
    ;;; move requests onto schedule, and trim bb ;;;
    (check-bb)
    ;;; execute scheduled items ;;;
    (setf result (driver drive-collection time nil))
    )) ; fast-posh

(defun real-fast-posh (drive-collection)
  (do ((result nil) (time 0))
      ((or (eq result 'drive-lost) (eq result 'drive-won)) result)
    (setf time (get-internal-real-time))
    ;;; move requests onto schedule, and trim bb ;;;
    (check-bb)
    ;;; execute scheduled items ;;;
    (setf result (driver drive-collection time nil))
    )) ; real-fast-posh

;; If posh-output is t then debug prints to monitor, to-gui it goes to gui.
(defun real-posh (&optional &key (plan-name nil) (gui-p nil) (posh-output nil)
       (from-gui nil) (fast-p nil)
       (as-stream nil) (bb-stream nil)
       (drive-collection nil) ; comes from reading script
       (debug-world nil) (init-world nil) ; these are funcs
       (hush nil) ; only matters when no gui
       )
  ; may have to initialize something
  (when init-world (apply init-world nil))
  ; sanity checking
  (unless (xor drive-collection plan-name)
    (error "real-posh: must have drive-collection XOR plan-name. dc:~s pn:~s"
   drive-collection plan-name))
  (unless drive-collection        ; if not, haven't read brain yet!
    (setf drive-collection (posh-script plan-name)))
  ; may only want to fire off a single competence for some reason
  (unless drive-collection
    (defun driver (&rest no-reference) (action-scheduler)))
  ; may have to initialize something
  (when init-world (apply init-world nil))
  ; this works even if drive-collection is nil
  (let ((real-time (typep drive-collection 'real-time-drive-collection))
(bb nil) (schedule nil))
    (declare (special bb schedule))
    ; print to the GUI Output window if desired...
    (if fast-p
      (progn
        (when (or gui-p bb-stream as-stream posh-output)
            (error
     (format nil "posh:  Can't be fast if running GUI or debugging.")))
        (if real-time
    (real-fast-posh drive-collection) (fast-posh drive-collection))
)
    ; else not fast-p
     (do ((result nil) (time 0) (last-bb-head nil))
 ((or (eq result 'drive-lost) (eq result 'drive-won)) result)
(setf time                                ; see comment at top of file
      (if real-time
  (get-internal-real-time)
(+ time 1)))
        ;;; move requests onto schedule, and trim bb ;;;
(check-bb)
(when as-stream
  (show-schedule schedule as-stream time))
        ;;; execute scheduled items ;;;
(setf result (driver drive-collection time posh-output))

;;; the rest of this function is for debugging ;;;
(when bb-stream
  (show-bb bb bb-stream time last-bb-head)
  (setf last-bb-head (car bb)))
(when debug-world
  (apply debug-world (list posh-output time)))
(if gui-p   ; do stepping and such-like
    (handle-bod-gui-requests from-gui)
  ; else no gui or not hushed
  (when (not hush)
    (format t "~%Type a letter (b to break, q to quit, n no debug) and return to proc
    (let ((res (read nil)))
      (cond
       ((eq res 'b) (break))
       ((eq res 'n) (setf *debug* nil))
       ((or (eq res 'q) (eq res 'quit)) (return))
       )))) ; if gui-p
)))) ; do, if-fast, let, defun real-posh

; "el" should be a drive element
(defmacro time-for (el time)
  `(let ((d-el ,el))
     (or
      (<= (frequency d-el) 0)
      (> (frequency d-el) (- ,time (last-fired d-el))))))

(defun driver (drive-collection time debug-stream)
  (let ((dc drive-collection))
    (if (ready (goal dc)) 'drive-won
      (do ((pl (car (elements dc)) (car pls))   ; otherwise, start an el.
   (pls (cdr (elements dc)) (cdr pls))) ; this checks drive priorities
  ((null pl) 'drive-lost)
(let ((result
      (do ((el (car pl) (car els))   ; this checks individual elements
   (els (cdr pl) (cdr els)))
  ((null el) nil)
(if (and (time-for el time) (ready el))
    (progn
      (if (and debug-stream *debug-drive*)
  (format debug-stream "~%~s: Fired drive ~s"
  time (drive-name el)))
      (if (driven-action-scheduler  ; if an active element...
    (drive-name el) debug-stream
    *debug-action* *debug-failed-action*)
  (setf (last-fired el) time) ; ...reset fire time
(send-sol                     ; else start at root
 (drive-root el) (content (drive-root el)) time))
      (return t))    ; if ready
  (if (and debug-stream *debug-drive*)  ; else not firing
      (if (time-for el time)
  (format debug-stream "~%~s: Drive not ready ~s"
  time (drive-name el))
(format debug-stream "~%~s: Not time for drive ~s"
time (drive-name el))))
  )   ; end if fire
)))   ; end assignment to result
  (if result (return result))) ; body of "let result"
     )))) ; defun driver

(defun show-schedule (schedule stream time)
  (format stream "~%Schedule at ~d:" time)
  (do ((item (car schedule) (car sched))
       (sched (cdr schedule) (cdr sched))
       (iii 0 (+ iii 1)))
      ((null item) t)
    (format stream "~%~@4d ~@8s ~@40s ~8s ~s"
    iii (class-name (class-of item)) (content item)
    (drive-name item) item)
    )) ; show-schedule

(defun show-bb (bb stream time last-bb-head)
  (format stream "~%BB ~d adds:" time)
  (loop for item in
(do ((item (car bb) (car sched))
     (sched (cdr bb) (cdr sched))
     (result nil (cons item result)))
    ((or (eq item last-bb-head) (null item)) result))
    do (cond
      ((typep (bbitem-action item) 'sol-generic)
       (format stream "~%~@8s ~16s ~12s ~12s ~@8s"
         (bbitem-command item) (bbitem-tag item)
       (if (eq (bbitem-drive-name item) 'RAW-PROTOTYPE-NO-DRIVE)
   'none-known (bbitem-drive-name item))
       (name (bbitem-action item)) (bbitem-timeout item)))
      (t
       (format stream "~%~@8s ~16s ~12s ~12s ~@8s"
         (bbitem-command item) (bbitem-tag item)
       (if (eq (bbitem-drive-name item) 'RAW-PROTOTYPE-NO-DRIVE)
   'none-known (bbitem-drive-name item))
       (if (typep (bbitem-action item) 'function)
   'function (bbitem-action item)) ; then else
       (bbitem-timeout item)))
       )
    )) ; defun show-bb



#|

NOTE (Dec 2000): the fast-posh's have been replaced, without the lack
of check of result.  Drivers haven't been replaced yet, and maybe
action-scheduling should too.

There was no reason for these to be macros! (duh)

;; this doesn't check return value from drive, so won't notice termination
;; (not that fast-drive catches termination anyway!)
(defmacro fast-posh ()
  (do ((time 0 (+ time 1))) (nil)
      (fast-driver time)))

(defmacro real-fast-posh ()
  (do ((time (get-internal-real-time) (get-internal-real-time))) (nil)
      (fast-driver time)))

; this assumes that you've written the POSH code so that there's no
; goal, and there will always be some action to execute.  If you want
; safety, run your code in regular mode until you're sure it's
; debugged!
(defmacro fast-driver (time)
  `(let ((dc *drive-collection*))
     (do ((pl (car (elements dc)) (car pls))      ; otherwise, start an el.
  (pls (cdr (elements dc)) (cdr pls)))      ; this checks priorities
 ((null pl) 'drive-lost)
       (let ((result
      (do ((el (car pl) (car els))   ; this checks individual elements
   (els (cdr pl) (cdr els)))
  (nil)
(if (and (> (frequency el) (- ,time (last-fired el)))
 (ready el))
    (progn
      (setf (last-fired el) ,time)
      (return (driven-action-scheduler (drive-name el)))))
)))
 (if result (return result))))
     )
  ) ; fast-driver

|#
```

A.3 Reading POSH scripts

You can probably tell from this I actually have never bothered to learn much about parsing.

```
;; posh-script.lisp  -- JJB June 2000

;; This file should load after "posh" itself.  Then load the behaviors,
;; then the primitives, then read the scripts in.  Right now, it only allows
;; for one drive collection (DC or RDC).

#|
TODO --

Make the script global hashes into hashes on process or filename, so
that more than one script can be used in a MAS.  For now, all the
agents use the same scripts, and conditionals check things like gender
```

```
and dominance to change behaviors.  Alternatively, make them special
variables and declare / control them in the posh launchers.

put line number, object name in errors -- make them global variables too.

2640    FEMALE SIGN
2642    MALE SIGN
|#

;;;  GLOBALS AND ACCESSORS --- for the scripts, and for the primitives.
;;;

; you have to load the primitives before you can load the scripts!
; but you have to load this file before you do either....

(defvar *sense-primitives* (make-hash-table))
(defvar *act-primitives* (make-hash-table))
(defvar *action-patterns* (make-hash-table))
(defvar *competences* (make-hash-table))
;; for now, only onle drive-collection per agent
(defvar *drive-collection* nil)
(defvar *drive-name* nil)
(defvar *script-comment* nil)

; actually, these aren't necessarily always predicates...
(defmacro add-sense (name predicate)
  `(setf (gethash ,name *sense-primitives*) ,predicate))
(defmacro get-sense (name)
  `(gethash ,name *sense-primitives*))

(defmacro add-act (name function)
  `(setf (gethash ,name *act-primitives*) ,function))
(defmacro get-act (name)
  `(gethash ,name *act-primitives*))

; make sure this returns the drive-collection
(defmacro add-drive (name function)
  `(progn
     (setf *drive-name* ,name)
     (setf *drive-collection* ,function)
     ))

(defmacro get-drive (name)
  `(if (or (nullp ,name) (eq *drive-name* ,name))
       *drive-collection*
     nil))

;; These following are the structures actually used in the rest of this file
;; NB: validate-aggregate also knows the names of these hash tables


; actually, these aren't necessarily always predicates...
(defmacro add-action-pattern (name ap)
  `(setf (gethash ,name *action-patterns*) ,ap))
(defmacro get-action-pattern (name)
  `(gethash ,name *action-patterns*))

(defmacro add-competence (name comp)
  `(setf (gethash ,name *competences*) ,comp))
(defmacro get-competence (name)
  `(gethash ,name *competences*))


;these are only for editing / debugging GUI scripts
(defvar *competence-elements* (make-hash-table))
(defmacro add-competence-element (name comp)
  `(setf (gethash ,name *competence-elements*) ,comp))
(defmacro get-competence-element (name)
  `(gethash ,name *competence-elements*))

(defvar *drive-elements* (make-hash-table))
(defmacro add-drive-element (name comp)
  `(setf (gethash ,name *drive-elements*) ,comp))
(defmacro get-drive-element (name)
  `(gethash ,name *drive-elements*))


; useful for debugging...
(defun flush-brains ()
  (clrhash *sense-primitives*)
  (clrhash *act-primitives*)
  (clrhash *action-patterns*)
  (clrhash *competences*)
  (setf *drive-collection* nil)
  (setf *drive-name* nil)
  (setf *script-comment* nil)
  (format t "now re-load sense and act primitives")
  )

(defun partial-brain-flush ()
  (clrhash *action-patterns*)
  (clrhash *competences*)
  (setf *drive-collection* nil)
  (setf *drive-name* nil)
  (setf *script-comment* nil)
  )


;;;  BASIC FUNCTIONS --- for controlling things.  posh-script is the main
;;;     (READ)           interface to this file, symbol-cat might be
;;;                      generally useful.
;;;  (write functions below)


; Posh-script returns the drive collection, if there is one.
; It assumes the behavior library is already loaded.  'filename'
; should be a string, not a symbol.
; N.B. Unfortunately, read doesn't work as documented when you compile,
; so we make the script one big object...
(defun posh-script (filename)
  (unless (string-equal filename ".lap"
:start1 (- (array-total-size filename) 4))
    (setf filename (concatenate 'string filename ".lap")))
  (let* ((file (open filename :direction :input))
 (drive-collection (read-aggregates (read file nil 'eof))))
    (close file)
    (validate-aggregates)
    drive-collection))

; This doesn't actually check that there's only one drive-collection,
; it just overwrites and takes the last one read.
(defun read-aggregates (aggs)
  (do ((agg (car aggs) (car aggs))
       (aggs (cdr aggs) (cdr aggs))
       (agg-num 1 (+ 1 agg-num))
       (drive-collection nil))
      ((null agg) drive-collection)
    (let ((agtype (car agg)) (agg (cdr agg)))
      (if (find-element (cadr agg))           ; check name
  (error (format nil
   "read-aggregates: name (~s) already used, line ~s of file"
   (cadr agg) agg-num)))
      (cond
       ((or (eq agtype 'DOCUMENTATION) (eq agtype 'COMMENT))
(setf *script-comment* (cons agtype agg)))
```

```
       ((eq agtype 'C) (read-competence agg))
       ((eq agtype 'DC) (setf drive-collection
      (read-drive-collection agg 'drive-collection)))
       ((eq agtype 'RDC) (setf drive-collection
       (read-drive-collection agg
'real-time-drive-collection)))
       ((eq agtype 'AP) (read-action-pattern agg))
       (t (error
   (format nil "Unknown aggregate type passed to read-aggregate: ~s"
   agg)))))
    )) ; defun read-aggregates

;; this checks that everything referred to in the hierarchy has been defined.
;; this assumes that primitives (senses and acts) are already declared, as
;; well as the rest of the hierarchy having been read in.
(defun validate-aggregates ()
  (maphash #'validate-action-pattern *action-patterns*)
  (maphash #'validate-competence *competences*)
  ; drives only differ from comps at the element level...
  (if *drive-collection* (validate-competence 'bogus *drive-collection*))
  )

; this checks everyplace an "action" might be
(defun find-action-element (name)
  (cond
   ((get-act name))
   ((get-action-pattern name))
   ((get-competence name))
   ((and *drive-collection* (eq (name *drive-collection*) name))
    *drive-collection*)
   (t nil)))

; This checks *everyplace*.  Drive and competence element names are
; only for clarity in editing scripts and debugging logs -- they
; aren't actually used by the posh system for anything.
(defun find-element (name)
  (cond
   ((get-act name))
   ((get-sense name))  ; here's the difference from find-action-element
   ((get-action-pattern name))
   ((get-competence name))
   ((get-competence-element name))
   ((and *drive-collection* (eq (name *drive-collection*) name))
    *drive-collection*)
   ((get-drive-element name))
   (t nil)
   ))

(defmacro unused-name (name)
  `(not (find-element ,name)))

;;;  BASIC FUNCTION --- for controlling things.  posh-script is the main
;;;     (WRITE)         interface to this file
;;;

; This assumes you've already checked whether you really want to
; overwrite an existing file --- do that in the GUI.
; See comment above on reading for why we pretend this is all one object
(defun write-posh-script (filename)
  (let ((file (open filename :direction :output
    :if-exists :overwrite :if-does-not-exist :create)))
    (write-char #\( file)

    (if *script-comment* (pprint *script-comment* file))
    (if *drive-collection*
(pprint (write-drive-collection *drive-collection*) file))
    (loop for comp being each hash-value of *competences*
  do (pprint (write-competence comp) file))
    (loop for ap being each hash-value of *action-patterns*
  do (pprint (write-action-pattern ap) file))

    (write-char #\Newline file)
    (write-char #\) file)
    (close file)
    ))

#|
Notice that this constrains the actual POSH engine.  For example,  POSH
goal objects may be any sol-generic; POSH action-pattern elements may be
parallel.
The reason for this constraint is just that I can't be bothered to write
the full parser. I'll worry about it if someone needs it (or they can!)
--- JJB June 2000

            name :: [a token]
        sol-time :: (<time-unit> #)
        time-unit:: minutes, seconds, hz(1/sec), pm(1/min), none(not realtime)
                     should maybe add hours, days, microseconds?
            goal :: (goal <ap>)
      Competence :: (C <name> <sol-time> <goal> <other-elements>)
```

```
                       where time is for timeout, Ymir-like
Drive-collection :: (DC|RDC <name> <goal> <drive-elements>)
                       RDC for real-time systems
  other-elements :: (elements (<comp-element>+)+)
                       inner paren is priority level
  drive-elements :: (drives (<drive-element>+)+)
                       inner () is priority
    comp-element :: (<name> (trigger <ap>) <name> [#])
                       2nd name is an action, followed by optional # of retries
   drive-element :: (<name> (trigger <ap>) <name> [<sol-time>])
                       drive name, trigger, POSH root, [freq. for scheduling]
              ap :: (<act|(sense [value [predicate]])>*)
                       default value is t, default predicate is eq
                       "act" can be an action primitive or a composite
  Action-pattern :: (AP <name> <sol-time> <ap>)
                       where time is for timeout, Ymir-like
|#

;;;  READING --- from the file; script spec is directly above.
;;;

;;;  Drives

; this returns the drive collection, because add-drive does.
(defun read-drive-collection (agg drive-type)
  (let ((agg-name (car agg))
(agg-goal (cadr agg))
(agg-elements (caddr agg)))
    (add-drive
     agg-name
     (make-instance
      drive-type
      :name agg-name
      :content (make-content :command 'request
     :tag agg-name :action 'edsol)
      :goal (make-instance 'competence-element
     :competence 'see-action-tag
     :action (make-instance 'sol-generic
  :name (symbol-cat agg-name '-goal)
                          :content (make-content
                                    :command 'request
    :tag (symbol-cat agg-name '-goal)
                                    :action nil ))
     :trigger (read-ap-from-guts (cadr agg-goal)
    (symbol-cat agg-name '-goal-trigger)
    0 nil)) ; FIXME s/b -1?
      :elements (read-drive-elements (cdr agg-elements) agg-name)))
    )) ; defun read-drive-collection

; recursively build element list and return it
(defun read-drive-elements (agg-elements drive-collection)
  (cond
   ((null agg-elements) agg-elements)
   ((not (listp (car agg-elements)))
    (error
     (format nil
     "read-drive-elements: format error <~s> s/b
        (drives (<drive-element>+)+) where inner ()s indicate priority level."
     (car agg-elements))))
   ((listp (caar agg-elements))
    (cons (read-drive-element-sublist (car agg-elements) drive-collection)
  (read-drive-elements (cdr agg-elements) drive-collection)))
   )) ; read-drive-elements

; recursively build element sub-list and return it -- no sub-sub-list allowed
(defun read-drive-element-sublist (agg-elements drive-collection)
  (cond
   ((null agg-elements) agg-elements)
   ((or (not (listp (car agg-elements))) (listp (caar agg-elements)))
    (error
     (format nil
     "read-drive-element-sublist: format error <~s> s/b
              drive-element :: (<name> (trigger <ap>) [#])
                                optional # of retries"
     (car agg-elements))))
   (t
    (cons (read-drive-element (car agg-elements) drive-collection)
  (read-drive-elements (cdr agg-elements) drive-collection)))
   )) ; read-drive-element-sublist

;; drive-element :: (<name> (trigger <ap>) <name> [<sol-time>])
;;                    drive name, trigger, POSH root, [freq. for scheduling]

(defun read-drive-element (el drive-collection)
  (let ((drive-name (car el))
(trigger-ap (cadr (second el)))
(drive-root (third el))
(frequency-rep (fourth el))
(frequency nil))
    (if frequency
 (setf frequency (read-sol-time frequency))
      (setf frequency -1))
    (add-drive-element drive-name
      (make-instance 'drive-element
       :drive-collection drive-collection
       :drive-root-rep drive-root
       :drive-root (make-instance
'sol-generic
:name drive-name
:drive-name drive-name
:content (make-content
  :command 'request :tag drive-name
  :action drive-root))
       :frequency frequency
       :frequency-rep frequency-rep
       :drive-name drive-name
       :trigger (read-ap-from-guts
 trigger-ap (symbol-cat drive-name '-trigger) -1 trigger-ap))
  ))) ; defun read-drive-element


;;; Competences ---

; this could do more error checking on the key words above...
(defun read-competence (agg)
  (let ((agg-name (car agg))
(agg-timeout-rep (cadr agg))
(agg-goal (caddr agg))
(agg-elements (cadddr agg))
(agg-timeout (read-sol-time (cadr agg))))
    (add-competence
     agg-name
     (make-instance
      'sol-competence
      :name agg-name
      :content (make-content :command 'request
     :tag agg-name :action 'edsol)
      :timeout-rep agg-timeout-rep
      :timeout agg-timeout
      :goal (make-instance 'competence-element
     :competence 'see-action-tag
     :action (make-instance 'sol-generic
  :name (symbol-cat agg-name '-goal)
                          :content (make-content
                                   :command 'request
    :tag (symbol-cat agg-name '-goal)
                                :action nil ))
     :trigger (read-ap-from-guts (cadr agg-goal)
    (symbol-cat agg-name '-goal-trigger)
    agg-timeout agg-timeout-rep))
      :elements (read-competence-elements (cdr agg-elements)
  agg-name agg-timeout)))
    )) ; defun read-competence

; recursively build element list and return it
(defun read-competence-elements (agg-elements competence timeout)
  (cond
   ((null agg-elements) agg-elements)
   ((not (listp (car agg-elements)))
    (error
     (format nil
     "read-competence-elements: format error <~s>
        (elements (<comp-element>+)+) where inner ()s indicate priority level."
     (car agg-elements))))
   ((listp (caar agg-elements))
    (cons (read-competence-element-sublist (car agg-elements) competence
   timeout)
  (read-competence-elements (cdr agg-elements) competence timeout)))
   )) ; read-competence-elements

; recursively build element sub-list and return it -- no sub-sub-list allowed
(defun read-competence-element-sublist (agg-elements competence timeout)
  (cond
   ((null agg-elements) agg-elements)
   ((or (not (listp (car agg-elements))) (listp (caar agg-elements)))
    (error
     (format nil
     "read-competence-element-sublist: format error <~s> s/b
              comp-element :: (<name> (trigger <ap>) [#])
                              optional # of retries"
     (car agg-elements))))
   (t
    (cons (read-competence-element (car agg-elements) competence timeout)
  (read-competence-elements (cdr agg-elements) competence timeout)))
   )) ; read-competence-element-sublist

; this does return the competence element, but also saves it to make
; editing a bit faster.
(defun read-competence-element (el competence timeout)
  (let ((ce-label (car el))
(trigger-ap (cadr (second el)))
(action (third el))
(retries (fourth el)))
    (unless retries (setf retries -1))
    (unless (numberp retries)
      (error
       (format nil
     "read-competence-element: format error <~s> s/b
              comp-element :: (<name> (trigger <ap>) [#])
                              optional # of retries" el)))
    (add-competence-element ce-label
       (make-instance 'competence-element
 :competence competence
 :ce-label ce-label
 :action (make-instance
  'sol-generic
  :name action
  :content (make-content
    :command 'request :tag ce-label :action action))
 :retries retries
 :trigger (read-ap-from-guts
   trigger-ap (symbol-cat ce-label '-trigger)
   timeout trigger-ap)))
  )) ; defun read-competence-element

; internal-time-units-per-second is provided by ANSI standard lisp
(defun read-sol-time (sol-time)
  (unless (and (listp sol-time) (cdr sol-time) (numberp (cadr sol-time)))
    (error "poorly formed sol-time ~s, see posh-script comments" sol-time))
  (let ((unit (car sol-time))
(quant (cadr sol-time)))
    (cond
     ((or (eq unit 'none)
  (eq unit 'absolute)
  (eq unit 'not-real-time)) quant)
     ((or (eq unit 'minute)
  (eq unit 'min)
  (eq unit 'minutes)) (* quant internal-time-units-per-second 60))
     ((or (eq unit 'second)
  (eq unit 'sec)
  (eq unit 'seconds)) (* quant internal-time-units-per-second))
     ((or (eq unit 'hertz)
  (eq unit 'hz)) (* (/ 1 quant) internal-time-units-per-second))
     ((or (eq unit 'per-minute)
  (eq unit 'pm)) (* (/ 1 quant) internal-time-units-per-second 60))
```

```
     (t (error
 (format nil
 "Unknown time type passed to read-sol-time: ~s"
   unit))))
    ))  ; defun read-sol-time

;;; Action Patterns ---

(defun read-ap-from-guts (agg name timeout timeout-rep)
  (make-instance
   'sol-action-pattern
   :name name
   :ap-guts agg
   :timeout timeout
   :timeout-rep timeout-rep
   :content (make-content :command 'element
  :tag name :action 'edsol)
   :elements (mapcar #'read-ap-element agg)
   ))

(defun read-action-pattern (agg)
  (unless (and (symbolp (car agg)) (numberp (cadadr agg)))
     (error
      (format nil
      "read-action-pattern: missing a name and/or timeout ~s" agg)))
  (let ((agg-name (car agg))
(agg-timeout-rep (cadr agg))
(agg-timeout (read-sol-time (cadr agg)))
(ap (caddr agg)))
    (add-action-pattern
     agg-name
     (read-ap-from-guts ap agg-name agg-timeout agg-timeout-rep))
    )) ; read-action-pattern

; has to return something for the ap list (collected by mapcar)
; if you forget to put parens around a sense and nil, nil will trigger read-ap-sense
; which will then barf
(defun read-ap-element (item)
  (cond
   ((listp item) (read-ap-sense item))
   (t item)))

;; ap :: (<act|(sense [value [predicate]])>*)
;;    act or sense are names.  if no value, test is not null.
;;    if value, default predicate is eq.
;; returns a sol-generic
(defun read-ap-sense (item)
  (let* ((name (car item))
 (sensor (get-sense name))
 (value 'sol-not-valid-sense-value)
 (predicate nil)
 (sensep nil))
    (unless sensor
      (error
       (format nil "read-ap-sense: ~s is not a known sense." name)))
    (if (not (consp (cdr item)))        ; there's no value (value can be nil!)
(setf sensep `(not (null (apply ,sensor nil))))
      (progn                            ; else a target value
(setf value (cadr item))
(if (caddr item) ; there's a predicate
  (setf predicate (caddr item))
  (setf predicate 'eq)
          )
(setf sensep `(,predicate (apply ,sensor nil) ,value))
) ; if there's a sense
      )
    (make-instance 'sol-sense :name name
   :sensep sensep
   :content (make-content :tag name :command 'sense
  :action predicate :value value)
   )
    )) ; defun read-ap-sense


;;;  VALIDATING --- make sure all the elements of aggregates were
;;;                 eventually read in; replace symbols with functions

; since validate-{action-pattern,competence} are run from maphash, they
; get "no-reference" (key) as an argument whether they need it or not.

; this also returns the action-pattern, in case you need it
(defun validate-action-pattern (no-reference ap)
    (setf (elements ap) (mapcar #'validate-ap-element (elements ap)))
    ap)

; this just returns an appropriate element, mapcar builds the list
(defun validate-ap-element (ap-el)
  (cond
   ((symbolp ap-el)
    (let ((real-ap (find-element ap-el)))
      (if real-ap real-ap
(error       ; argh, this should know the line number,... globals?
 (format nil "validate-ap-element: unknown element ~s." ap-el)))
      ))
   ((typep ap-el 'sol-sense)       ; senses are sorted when read
    ap-el)
   (t (error
       (format nil
       "validate-ap-element: ~s is of unknown type." ap-el)))
   )) ; defun validate-ap-element

; see comment about "no-reference" above
(defun validate-competence (no-reference comp)
  (setf (trigger (goal comp))
(validate-action-pattern
 'bogus (trigger (goal comp))))
  (setf (elements comp) (validate-competence-elements (elements comp)))
  ; (setf (gethash name *competences*) comp) ; shouldn't be necessary
  )

; returns the corrected list of elements, or dies trying
; note: this works for drive elements too.
(defun validate-competence-elements (elements)
  (cond
   ((null elements) elements)
   ((listp (car elements))
    (cons (validate-competence-element-sublist (car elements))
  (validate-competence-elements (cdr elements))))
   ((typep (car elements) 'competence-element)
    (cons (validate-competence-element (car elements))
  (validate-competence-elements (cdr elements))))
   ((typep (car elements) 'drive-element)
    (cons (validate-drive-element (car elements))
  (validate-competence-elements (cdr elements))))
   (t (error
       (format nil
       "validate-competence-elements: ~s is of unknown type."
       (car elements))))
   )) ; defun validate-competence-elements

; returns the corrected sub-list (same priority) of elements, or dies trying
; competence element sublists can't contain lists
; note: this works for drive elements too.
(defun validate-competence-element-sublist (elements)
  (cond
   ((null elements) elements)
   ((typep (car elements) 'competence-element)
    (cons (validate-competence-element (car elements))
  (validate-competence-element-sublist (cdr elements))))
   ((typep (car elements) 'drive-element)
    (cons (validate-drive-element (car elements))
  (validate-competence-element-sublist (cdr elements))))
   (t (error
       (format nil
       "validate-competence-element-sublist: ~s is of unknown type."
       (car elements))))
   )) ; defun validate-competence-element-sublist

; returns the fixed element.  largely replicated below for drives
; Note: if you try to validate something already validated, this craps out.
;       eventually I should fix this, but right now it's an unsubtle
;       indication you forgot to flush-brains.  FIXME
(defun validate-competence-element (el)
  (setf (trigger el) (validate-action-pattern 'bogus (trigger el)))
  (let ((el-action (action el)))
    (unless (typep el-action 'sol-generic)
      (error                 ; argh, this should know the line number,...
       (format nil
       "validate-competence-element: element ~s should be an s-g.")))
    (let ((real-act (action el-action)))
      (cond
       ((and (symbolp real-act) (find-action-element real-act))
(fix-content-action el-action (find-element real-act))
el)   ; this is the return value.
       (t  (error       ; argh, this should know the line number,...
    (format nil
    "validate-competence-element: unknown action ~s."
    real-act)))
       ))
    )) ; defun validate-competence-element


; returns the fixed element.  largely replicates code above.
; Notes: a drive-element-(drive-root) should be a sol-generic... (argh!)
(defun validate-drive-element (el)
  (setf (trigger el) (validate-action-pattern 'bogus (trigger el)))
  (let ((el-action (drive-root el)))
    (unless (typep el-action 'sol-generic)
      (error                 ; argh, this should know the line number,...
       (format nil
       "validate-competence-element: element ~s should be an s-g.")))
    (let ((real-act (action el-action)))
      (cond
       ((and (symbolp real-act) (find-action-element real-act))
(fix-content-action el-action (find-element real-act))
el)   ; this is the return value.
       (t  (error       ; argh, this should know the line number,...
    (format nil
    "validate-drive-element: unknown action ~s."
    real-act)))
       ))
    )) ; defun validate-drive-element


;;;  WRITING --- notice these functions don't write to the file, but
;;;              build up the list that should be written.
;;;

;;;  Drives
(defun write-drive-collection (dc)
  (let ((drive-type-spec
 (cond
  ((typep dc 'real-time-drive-collection) 'RDC)
  ((typep dc 'drive-collection) 'DC)
  (t (error "~s not really a drive-collection!" dc)))))
    (list drive-type-spec (name dc)
  (list 'goal (ap-guts (trigger (goal dc))))
  (cons 'drives (mapcar #'write-drive-element-list (elements dc)))
  )
    ))

(defun write-drive-element-list (elements)
  (mapcar #'write-drive-element elements)
   ) ; write-drive-element-list

(defun write-drive-element (el)
  (if (frequency-rep el)
      (list (drive-name el)
    (list 'trigger (ap-guts (trigger el)))
    (drive-root-rep el)
    (frequency-rep el))
    (list (drive-name el)
  (list 'trigger (ap-guts (trigger el)))
  (drive-root-rep el))))
```

```
;;; Competences ---

; this could do more error checking on the key words above...
(defun write-competence (comp)
  (list 'C (name comp)
(timeout-rep comp)
(list 'goal (ap-guts (trigger (goal comp))))
(cons 'elements (mapcar #'write-competence-element-list (elements comp)))
)) ; defun write-competence

(defun write-competence-element-list (elements)
  (mapcar #'write-competence-element elements)
   )

(defun write-competence-element (el)
  (if (> (retries el) -1)
      (list (ce-label el)
    (list 'trigger (ap-guts (trigger el)))
    (name (action el))
    (retries el))
    (list (ce-label el)
  (list 'trigger (ap-guts (trigger el)))
  (name (action el))
  )))

;;; Action Patterns ---

(defun write-action-pattern (ap)
  (list 'AP (name ap)
(timeout-rep ap)
(ap-guts ap)
))
```

A.4 Utility Functions

For the POSH GUI, it became useful to define all the files that need to be read in for a particular behavior library in one place. That place is here:

```
;;; posh-bl.lisp --- this is a file that has to be edited as libraries
;;; are added.  It's used by bod-interface, but can be used by
;;; anything.  --- JJB Dec 2000
```

```
(defun load-behavior-library (libname)
  (cond
   ((eq libname 'monkey)
    (compile-file "monkey-bl" :load t)
    (compile-file "monkey-prims" :load t)
    (compile-file "monkey-world" :load t))
   ((eq libname 'primate)          ;; edit colony.lisp (intitialize pm) too ;;
    (compile-file "posh-utils" :load t) ;; utils, useful for real time stuff
    (compile-file "primate-interface" :load t) ;; launcher & world
    (compile-file "colony" :load t)   ; makes the world, but also basic bl
    (compile-file "primate-groom" :load t)
    (compile-file "primate-novelty" :load t)
    (compile-file "primate-prims" :load t))
   (t (error "load-behavior-library:  unknown library passed ~s" libname))
   ))
```

Here are a couple functions I think ought to be in CLOS (everything else is!)

```
;; jjb-utils.lisp -- stuff lisp ought to have

(defmacro xor (a b)
  `(let ((aa ,a) (bb ,b))
     (and (or aa bb) (not (and aa bb)))))

;this returns a symbol, honest... (and also a length)
(defmacro symbol-cat (s1 s2)
 `(read-from-string (concatenate 'string (string ,s1) (string ,s2))))
```

Here are a couple more utility functions specific to POSH. Notice one is the drive-level code for the primate behavior library — I suspect that's also an idiom, so I've put it here.

```
;; posh-utils.lisp -- JJB Jan 2001.  Stuff you might want in more than
;; one behavior library.   Maybe each "thing" should be posh-u-thing.lisp

;; Right now, just one thing: drive-mem --- Amorphous recent memory
;; for maintaining drive equilibrium.  Oh, and Time, since that's
;; needed for drive-mem.

;ITUPS is defined by lisp
(defconstant INTERNAL-TIME-UNITS-PER-MINUTE (* 60 INTERNAL-TIME-UNITS-PER-SECOND))

;; All times should be relative to this, so you can speed up or slow
;; down the simulation! (esp. with GUI turned off...)
(defvar *time-increment* 0.1 "how often you show updates in World GUI")
(defvar *minute-time-increment* 'bogus "how often per minute you update...")
(defvar *incs-per-second* 'bogus "mult. factor for getting seconds")
(defvar *incs-per-minute* 'bogus "mult. factor for getting seconds")
(defvar *incs-per-time* 'bogus "mult. factor for get-internal-real-time")

;; RERUN THE BELOW if you change *time-increment*
(defun update-increments ()
  (setf *incs-per-second* (/ 1 *time-increment*))
  (setf *incs-per-minute* (* 60 *incs-per-second*))
  (setf *minute-time-increment* (/ 1 *incs-per-minute*))
  (setf *incs-per-time*   ; (incs / sec) / (itu / sec) => incs / itu
(/ *incs-per-second* INTERNAL-TIME-UNITS-PER-SECOND))
  )

(update-increments)


;; Most recent stuff is stored as it comes in, gets chunked by minutes
;; and stored, old chunks get dropped off.  The use of minutes as a
;; time unit is somewhat arbitrarily, but convenient for models of
;; non-human primates.

;; "level" tells you the current level of the drive, which you can
;; compare to your ideal.  "increment" means you spent a unit doing
;; that thing.  These are all that should be called by another
;; program.  "update-memory" quietly does the book-keeping.
;; except "latched" which helps you use a drive for persistance

;; you want to start every mem with a time and probably a false memory
;; e.g. (make-instance 'drive-memory :cc-start-time (get-internal-real-time)
;;                     :recent-chunks `(0.2 0.2 0.2 0.2 0.2))

(defclass drive-memory ()
  (
   (current-chunk :accessor current-chunk :initarg :current-chunk :initform 0)
   (cc-start-time :accessor cc-start-time :initarg :cc-start-time :initform 0)
   ; most recent is first
   (recent-chunks :accessor recent-chunks :initarg :recent-chunks
  :initform `(0.2 0.2 0.2 0.2 0.2))
   (rc-total :accessor rc-total :initarg :rc-total :initform 1.0)
   ; not including the current chunk!
   (memory-length :accessor memory-length :initarg :memory-length :initform 5)
   ; 3 below are for latching
   (engage-latch :accessor engage-latch :initarg :engage-latch :initform 0.1)
   (disengage-latch :accessor disengage-latch :initarg :disengage-latch
    :initform 0.3)
   (latch :accessor latch :initarg :latch :initform nil)
   )
)

(defmethod show ((dm drive-memory))
  (format t "current ~s; recent ~s; level ~s" (current-chunk dm) (recent-chunks dm) (

; engage when below min, disengage when above max, otherwise just
; report current state
(defmethod latched ((dm drive-memory))
  (if (latch dm)
      (when (> (level dm) (disengage-latch dm)) (setf (latch dm) nil))
    (when (< (level dm) (engage-latch dm)) (setf (latch dm) t)))
  (latch dm))

; level will be between 0 and 1
(defmethod level ((dm drive-memory))
  (let ((now (get-internal-real-time)))
    (update-memory dm now)
    (let ((current-mem-length (/ (- now (cc-start-time dm))
       INTERNAL-TIME-UNITS-PER-MINUTE)))
      (/ (+ (current-chunk dm) (rc-total dm))
 (+ current-mem-length (memory-length dm)))
      )))

(defmethod increment ((dm drive-memory) &optional (incrs 1))
  (update-memory dm (get-internal-real-time))
  (setf (current-chunk dm) (+ (current-chunk dm)
      (* incrs *minute-time-increment*)))
  )

; The do loop updates the memory evenly, but leaves none for the
; current time slot.  This is a hack but it's probably good enough.
(defmethod update-memory ((dm drive-memory) now)
  (let ((current-mem-length (/ (- now (cc-start-time dm))
     INTERNAL-TIME-UNITS-PER-MINUTE)))
    (when (< 1.0 current-mem-length)
      (setf current-mem-length
    (do ((ixi current-mem-length (- ixi 1.0))
 (weight (/ (current-chunk dm)
    (floor current-mem-length))))
((> 1.0 ixi) ixi)
```

```
      (setf (recent-chunks dm)
    (cons weight (butlast (recent-chunks dm)))))
    ) ;setf current-mem-length
      (setf (rc-total dm) (apply #'+ (recent-chunks dm)))
      (setf (cc-start-time dm)
    (- now (floor
    (* current-mem-length INTERNAL-TIME-UNITS-PER-MINUTE))))
      (setf (current-chunk dm) 0.0))
    ) t)
```

A.5 The GUI

Here's how I load all these files. In Lispworks, you have to type (require “comm”) before you type (load “posh-gui”).

```
;;; to startup --- (require "comm") (load "posh-gui")

#| There are 3 streams involved in the communication between the GUI
   and the posh process.  One is a direct one from posh to the output
   window on the GUI.  The other 2, for sending signals to posh from
   the GUI, should prefereably be 1, but get connected through a server.
   I tried really hard just to pass one that direction too, but I couldn't
   get it working.  I think it's because lispworks doesn't give you
   the pieces you need for a server (e.g. "listen"), it just gives you
   a server function.
|#

; (require "comm")  ; for socket communication -- doesn't seem to work when
; compiling though, so have to do in advance...

; will this work??? If not, bug report again...
(SCM:REQUIRE-PRIVATE-PATCH "~/etc/update-scroll-bar.ufsl" :capi)

(in-package capi)  ; this should really be fixed!  FIX ME

(compile-file "jjb-utils" :load t)
(compile-file "posh" :load t)
(compile-file "posh-script" :load t)
(compile-file "posh-run" :load t)
(compile-file "posh-bl" :load t)
```

```
(compile-file "bod-interface" :load t)
(compile-file "lap-interface" :load t)

; get the GUI set up properly

(setf *gui* t)
(setf *debug* nil) ;; default is t in posh-run, but this screws up the GUI
(defvar *posh-port* 117674)  ;; 11POSH
(setf *posh-port* 117666) ;; emergency backup port
(defvar *posh-proc* nil)

(defvar *bod-interface* (make-instance 'bod-interface))
(display *bod-interface*)
(defvar *debug-interface* nil)

#| Making the GUI be able to talk to posh is non-trivial, esp. given the
   limitations lispworks sets on how you make servers (lots is hidden.)
   This relies on an independent server to start up the posh process --
   it has to get any arguments from global variables.  Making a posh
   process is triggered by trying to set up a pipe to talk to it.  See also
   bod-interface.lisp and posh-run.lisp .
|#

(defun maybe-read-line (stream b c)
               (let ((ch (read-char-no-hang stream)))
                 (cond
                  (ch (unread-char ch stream)
                      (read-line stream b c))
                  (t "nothing"))))

(defun make-stream-and-posh (handle)
  (let ((from-gui-stream (make-instance 'comm:socket-stream
                               :socket handle
                               :direction :io
                               :element-type
                                  'base-char)))
  (setf *posh-proc*
(mp:process-run-function (format nil "~S ~D"
 (if *run-fast* 'fastposh 'posh)
 handle)
 '()
 'posh-from-gui *run-fast*
 (collector-pane-stream
  (get-output-cp *bod-interface*))
 from-gui-stream))))
```

```
(comm:start-up-server :function 'make-stream-and-posh
                      :service *posh-port*)
```

Here's the real GUI code:

```
;; bod-interface.lisp -- making the posh process actually run also
;; relys on code in posh-gui.lisp, which sets up the server

(defvar *color* ':thistle3)
(defvar *color2* ':lemonchiffon)
(defvar *posh-input-stream* nil)
(defvar *run-fast* nil)

; these variables can be different in the different processes --- see
; handle-bod-gui-requests comment below! (and bod-env-from-message)
(defvar *stepping* nil)         ; N n   capital is setting to t
(defvar *step-please* nil)      ; S s   small is setting to nil
(defvar *continue-please* nil)  ; C c
(defvar *break-please* nil)     ; B b

; remember --- this is called from posh, which is a seperate process
; working with it's own copies of the above variables (?).  So anytime you
; set them and something is running, you'd better send it notice!
; read all waiting characters, then process them.  If waiting for a step
; signal, wait a tenth of a second then tail-recurse
(defun handle-bod-gui-requests (from-gui)
; (error (progn
;  (format (collector-pane-stream (get-output-cp *bod-interface*))
;  "break ~s; step ~s; continue ~s; stepping ~s" *break-please*
;  *step-please* *continue-please* *stepping*)
;   "hi"))
  (do ((new-char (read-char-no-hang from-gui nil)
 (read-char-no-hang from-gui nil)))
      ((or (not new-char) (eq new-char :eof) t)
    (bod-env-from-message new-char)))
  (cond
   (*break-please* (break "Click Debug.") (setf *break-please* nil))
   (*step-please* (setf *stepping* t) (setf *step-please* nil))
   (*continue-please* (setf *stepping* nil) (setf *continue-please* nil))
   (*stepping* ; otherwise (not stepping and no requests) fall through
    (sleep 0.1)
    (handle-bod-gui-requests from-gui)))
  ) ; defun handle-bod-gui-requests

; see comments above.  Most of these don't actually get sent.
(defun bod-env-from-message (msg-char)
  (cond
   ((eq msg-char #\N) (setf *stepping* T))
   ((eq msg-char #\n) (setf *stepping* nil))
   ((eq msg-char #\S) (setf *step-please* T))
   ((eq msg-char #\s) (setf *step-please* nil))
   ((eq msg-char #\C) (setf *continue-please* T))
   ((eq msg-char #\c) (setf *continue-please* nil))
   ((eq msg-char #\B) (setf *break-please* T))
   ((eq msg-char #\b) (setf *break-please* nil))))

;; Trying to talk to posh-port automatically starts a posh process.
;; See posh-gui.lisp and posh-run.lisp .
(defun send-run (interface)
  (reset) ; erase prior stuff on bb and schedule
  (setf *run-fast* nil)
  (setf *posh-input-stream* (comm:open-tcp-stream "localhost" *posh-port*))
  (force-output *posh-input-stream*)    ; in case pending char e.g."step"
  (setf (capi:button-enabled (get-run-pb interface)) nil)
  (setf (capi:button-enabled (get-fast-run-pb interface)) nil)
  (setf (capi:button-enabled (get-quit-pb interface)) t)
  )

(defun send-fast-run (interface)
  (reset) ; erase prior stuff on bb and schedule
  (setf *run-fast* t)
  (setf *posh-input-stream* (comm:open-tcp-stream "localhost" *posh-port*))
  (force-output *posh-input-stream*)    ; in case pending char e.g."step"
  (setf (capi:button-enabled (get-run-pb interface)) nil)
  (setf (capi:button-enabled (get-fast-run-pb interface)) nil)
  (setf (capi:button-enabled (get-bod-debug-cb interface)) nil)
  (setf (capi:button-enabled (get-quit-pb interface)) t)
  )

(defun send-quit (interface)
  (mp:process-kill *posh-proc*) (reset-bod-run-vars)
  (setf (capi:button-enabled (get-run-pb interface)) t)
  (setf (capi:button-enabled (get-fast-run-pb interface)) t)
  (setf (capi:button-enabled (get-bod-debug-cb interface)) t)
  (setf (capi:button-enabled (get-quit-pb interface)) nil)
  )
```

```
(defun send-exec (item)
  (cond
   ((eq item 'step) (write-char #\S *posh-input-stream*))
   ((eq item 'continue) (write-char #\C *posh-input-stream*))
   ((eq item 'break) (write-char #\B *posh-input-stream*))
   (t (error "I'm a -~s- dammit" item)))
  (if *posh-input-stream*
      (force-output *posh-input-stream*))
  )

(defun reset-bod-run-vars ()
  (setf *quit-please* nil) (setf *break-please* nil)
  (setf *step-please* nil)
  (setf *continue-please* nil) (setf *posh-proc* nil)
  (if *debug* (setf *stepping* t)(setf *stepping* nil))
)

;needs to be :callback-type :interface
(defun set-bod-debug-on (interface)
  (setf *debug* t)
  (set-button-panel-enabled-items (get-execution-pbp interface) :set t)
  (setf *debug-interface* (make-instance 'bod-debug-viewer))
  (setf (capi:button-enabled (get-db-world-cb interface)) t
(capi:button-enabled (get-db-drive-cb interface)) t
(capi:button-enabled (get-db-action-cb interface)) t
(capi:button-enabled (get-db-non-act-cb interface)) t)
  (display *debug-interface*)
  (setf *bb-stream*
(capi:collector-pane-stream (get-bb-cp *debug-interface*)))
  (setf *as-stream*
(capi:collector-pane-stream (get-as-cp *debug-interface*)))
  (if *posh-proc*
      (set-button-panel-enabled-items (get-execution-pbp interface) :set t))
  (setf *stepping* t) ; default, otherwise no way to get in and step!
  )

(defun set-bod-debug-off (interface)
  (setf *debug* nil)
  (when *debug-interface*
    (destroy *debug-interface*) (setf *debug-interface* nil))
  (setf (capi:button-enabled (get-db-world-cb interface)) nil
(capi:button-enabled (get-db-drive-cb interface)) nil
(capi:button-enabled (get-db-action-cb interface)) nil
(capi:button-enabled (get-db-non-act-cb interface)) nil)
  (set-button-panel-enabled-items (get-execution-pbp interface) :set nil)
  )

;needs to be :callback-type :interface
(defun choose-bod-lap (interface)
  (let ((file (prompt-for-file "Which Script File?" :filter "*.lap")))
    (partial-brain-flush)
    (posh-script (file-namestring file))
    (setf (display-pane-text (get-choose-fap-dp interface))
  (file-namestring file))
    (setf (capi:button-enabled (get-run-pb interface)) t)
    (setf (capi:button-enabled (get-fast-run-pb interface)) t)
    (setf (capi:button-enabled (get-graph-fap-cb interface)) t)
    (setf (capi:display-pane-text (get-real-time-dp interface))
  (if *drive-collection*
      (string (typep *drive-collection* 'real-time-drive-collection))
    "no d/c"))  ; this shouldn't happen much --- it won't run!
  ))

(defun set-debug-length (choice)
  (let ((new-time
 (cond
  ((eq choice '.5sec) (* 0.5 INTERNAL-TIME-UNITS-PER-SECOND))
  ((eq choice '2sec) (* 2 INTERNAL-TIME-UNITS-PER-SECOND))
  ((eq choice '30sec) (* 30 INTERNAL-TIME-UNITS-PER-SECOND))
  ((eq choice '2min) (* 120 INTERNAL-TIME-UNITS-PER-SECOND))
  ((eq choice '10min) (* 600 INTERNAL-TIME-UNITS-PER-SECOND))
  )))
    (setf DEFAULT-VIEWTIME new-time DEBUG-SCHEDULE-TIME new-time)
  )) ; defun set-time-inc


;needs to be :callback-type :item-interface
(defun load-bod-library (choice interface)
  (flush-brains)
  (kill-fap-windows)
  ; don't allow new script file read until loading is finished
  (setf (capi:button-enabled (get-choose-fap-pb interface)) nil)
  (setf (capi:button-enabled (get-fast-run-pb interface)) nil)
  (setf (capi:button-enabled (get-run-pb interface)) nil)
  (setf (display-pane-text (get-choose-fap-dp interface)) nil)
  (setf (capi:button-enabled (get-graph-fap-cb interface)) nil)
  (setf (capi:display-pane-text (get-real-time-dp interface)) "n/a")
  (unless (null choice)
    (load-behavior-library choice)
    (setf (capi:button-enabled (get-choose-fap-pb interface)) t))
  ) ; defun load-bod-library

;; More graph & edit functions are in lap-interface.lisp ;;
(defun graph-bod-lap (interface)
  (setf (capi:button-selected (get-graph-fap-cb interface)) t)
  (setf *posh-graph-tool* (make-instance 'bod-fap-graph))
  (setf (capi:title-pane-text (get-graph-script-name *posh-graph-tool*))
(display-pane-text (get-choose-fap-dp interface)))
  (display *posh-graph-tool*)
  )

(defun kill-graph-bod-lap (unreferenced-interface)
  (if *posh-graph-tool* (destroy *posh-graph-tool*))
  (no-more-fap-graph)
  )

(defun edit-bod-lap (interface)
  (setf (capi:button-selected (get-edit-fap-cb interface)) t)
  (setf *posh-edit-tool* (make-instance 'bod-fap-edit))
  (setf (capi:display-pane-text (get-lap-file-dp *posh-edit-tool*))
(display-pane-text (get-choose-fap-dp interface)))
  (display *posh-edit-tool*)
  )

(defun kill-edit-bod-lap (unreferenced-interface)
  (if *posh-edit-tool* (destroy *posh-edit-tool*))
  (no-more-fap-edit)
  )

(defun kill-fap-windows ()
  (if *posh-graph-tool* (kill-graph-bod-lap *posh-graph-tool*))
  (if *posh-edit-tool* (kill-edit-bod-lap *posh-edit-tool*))
  )
;; More graph & edit functions are in lap-interface.lisp ;;

(defun kill-all-the-bod-kids (&rest unreferenced-args)
  (setf *bod-interface* nil)
  (kill-fap-windows)
  (if *debug-interface* (destroy *debug-interface*))
  )

(defun nuke-debug-window-ref (&rest unreferenced-args)
  (setf *debug-interface* nil)
  (setf *bb-stream* nil)
  (setf *as-stream* nil)
  )

(defun db-world-on (interface)
  (setf (capi:button-selected (get-db-world-cb interface)) t
*debug-world* t)
  )
(defun db-world-off (interface)
  (setf (capi:button-selected (get-db-world-cb interface)) nil
*debug-world* nil)
  )
(defun db-drive-on (interface)
  (setf (capi:button-selected (get-db-drive-cb interface)) t
*debug-drive* t)
  )
(defun db-drive-off (interface)
  (setf (capi:button-selected (get-db-drive-cb interface)) nil
*debug-drive* nil)
  )
(defun db-action-on (interface)
  (setf (capi:button-selected (get-db-action-cb interface)) t
*debug-action* t)
  )
(defun db-action-off (interface)
  (setf (capi:button-selected (get-db-action-cb interface)) nil
*debug-action* nil)
  )
(defun db-non-act-on (interface)
  (setf (capi:button-selected (get-db-non-act-cb interface)) t
*debug-failed-action* t)
  )
(defun db-non-act-off (interface)
  (setf (capi:button-selected (get-db-non-act-cb interface)) nil
*debug-failed-action* nil)
  )

(define-interface bod-interface ()
  ()
  (:panes
   (title-pane-1
    title-pane
    :text "POSH Control Panel"
    :background :white)
   (debug-cb check-button
    :text "Debug Mode"
    :selection-callback 'set-bod-debug-on
```

```
 :retract-callback 'set-bod-debug-off
 :reader get-bod-debug-cb
 :callback-type :interface
 :background *color*
 ; :foreground *color*  ; this makes the text orange, not the button
 )
(real-time-dp
 display-pane
 :text "n/a"
 :title "                                     Real Time:"
 :title-position :left
 :reader get-real-time-dp
 :background :azure)
(debug-tp
 title-pane
 :title "     Debug Options:"
 :title-position :left)
(time-dp
 display-pane
 :text " --- "
 :title "   Time:"
 :title-position :left
 :background :azure)
(choose-fap-dp
 display-pane
 :title " Script:"
 :title-position :left
 :text "none"
 :visible-border t
 :min-width 200
 :background :white
 :reader get-choose-fap-dp)
(choose-fap-pb
 push-button
 :text "Choose Script"
 :selection-callback 'choose-bod-lap
 :callback-type :interface
 :reader get-choose-fap-pb
 :enabled nil
 :background *color*)
(graph-fap-cb check-button
 :text "Graph Script"
 :selection-callback 'graph-bod-lap
 :retract-callback 'kill-graph-bod-lap
 :callback-type :interface
 :reader get-graph-fap-cb
 :enabled nil
 :background *color*
 ; :foreground *color*  ; this makes the text orange, not the button
 )
(edit-fap-cb check-button
 :text "Edit Script"
 :selection-callback 'edit-bod-lap
 :retract-callback 'kill-edit-bod-lap
 :callback-type :interface
 :reader get-edit-fap-cb
 :enabled t
 :background *color*
 ; :foreground *color*  ; this makes the text orange, not the button
 )
(db-world-cb check-button
 :text "world"
 :selection-callback 'db-world-on
 :retract-callback 'db-world-off
 :callback-type :interface
 :reader get-db-world-cb
 :enabled nil
 :selected *debug-world*
 :background *color*
 )
(db-drive-cb check-button
 :text "drive"
 :selection-callback 'db-drive-on
 :retract-callback 'db-drive-off
 :callback-type :interface
 :reader get-db-drive-cb
 :enabled  nil
 :selected *debug-drive*
 :background *color*
 )
(db-action-cb check-button
 :text "action"
 :selection-callback 'db-action-on
 :retract-callback 'db-action-off
 :callback-type :interface
 :reader get-db-action-cb
 :enabled  nil
 :selected *debug-action*
 :background *color*
 )
```

```
(db-non-act-cb check-button
 :text "non-acts"
 :selection-callback 'db-non-act-on
 :retract-callback 'db-non-act-off
 :callback-type :interface
 :reader get-db-non-act-cb
 :enabled  nil
 :selected *debug-failed-action*
 :background *color*
 )
(library-op
 option-pane
 :items '(nil monkey primate)
 :selected-item 0
 :title "Library:"
 :selection-callback 'load-bod-library
 :callback-type :item-interface
 :reader get-library-op
 :background :white)
(run-pb
 push-button
 :text "Run"
 :selection-callback 'send-run
 :callback-type :interface
 :enabled nil
 :reader get-run-pb
 :background *color*)
(fast-run-pb
 push-button
 :text "Run Fast"
 :selection-callback 'send-fast-run
 :callback-type :interface
 :enabled nil
 :reader get-fast-run-pb
 :background :hotpink)
(quit-pb
 push-button
 :text "Quit"
 :selection-callback 'send-quit
 :callback-type :interface
 :enabled nil
 :reader get-quit-pb
 :background *color*)
(execution-pbp
 push-button-panel
```

```
   :items '(step continue break)
   :selection-callback 'send-exec
   :callback-type :data
   :print-function 'string-capitalize
   :enabled nil
   :reader get-execution-pbp
   :x-adjust :center  ; no impact...
   :background *color*)
  (debug-length-op ; this actually *can* affect operations as well
   option-pane
   :items '(default .5sec 2sec 30sec 2min 10min)
   :print-function 'string-downcase
   :selected-item 0
   :title "Debug Trail Length:"
   :selection-callback 'set-debug-length
   :callback-type :item
   :min-width 100
   :max-width 100
   :reader get-debug-length-op
   :background :white)
  (output-cp
   collector-pane
   :title "Debugging Output:"
   ; :stream *posh-output-stream*
   :reader get-output-cp
   :visible-min-width '(:character 80)
   )
  (output-lp
   listener-pane
   :title "Listener:"
   ; :stream *posh-output-stream*
   :reader get-output-lp
   :visible-min-width '(:character 80)
   )
  (column-layout-divider-1
   column-layout-divider
   :background :black
   :foreground *color*)
  (column-layout-divider-2
   column-layout-divider
   :background :black
   :foreground *color*)
  )
 (:layouts
  (column-layout-1
     column-layout
     '(row-layout-1 library-op row-layout-2 column-layout-divider-1
    row-layout-3 row-layout-4 output-lp
    column-layout-divider-2 row-layout-5 output-cp))
    (simple-layout-1  ; this didn't help right justify
     simple-layout
     '(debug-cb)
     :x-adjust :right
     )
    (row-layout-1
     row-layout
     '(title-pane-1))
    (row-layout-2
     row-layout
     '(choose-fap-dp choose-fap-pb graph-fap-cb edit-fap-cb))
    (row-layout-blank  ; this is a bad idea: it expands vertically on stretch!
     row-layout
     '())
    (row-layout-3
     row-layout
     '(run-pb quit-pb fast-run-pb real-time-dp time-dp))
    (row-layout-4
     row-layout
     '(debug-cb execution-pbp debug-length-op)
     :x-adjust :centre
     )
    (row-layout-5
     row-layout
     '(db-world-cb db-drive-cb db-action-cb db-non-act-cb)
     :x-adjust :centre
     ))
  (:default-initargs
   :best-height 339
   :best-width 458
   :destroy-callback 'kill-all-the-bod-kids ; destroy stuff hangs things
   :title "BOD Support"
   :background *color*)
)

(define-interface bod-debug-viewer ()
  ()
  (:panes
   (bb-cp
    collector-pane
    :title "Bulletin Board"
    ; :stream *bb-stream*
    :reader get-bb-cp
    :visible-min-width '(:character 80))
   (column-layout-divider-1
    column-layout-divider
    :background :black
    :foreground *color*)
   (as-cp
    collector-pane
    :title "Action Scheduler"
    ; :stream *as-stream*
    :reader get-as-cp
    :visible-min-width '(:character 80)))
  (:layouts
   (column-layout-1
    column-layout
    '(bb-cp column-layout-divider-1 as-cp))
   )
  (:default-initargs
   :best-height 339
   :best-width 458
   :layout 'column-layout-1
   :destroy-callback 'nuke-debug-window-ref
   :title "BOD Support - Debug Window"
   :background *color*))
```

Here's some more!

```
;; lap-interface.lisp -- edit POSH plans.  A couple of functions
;; involving these interfaces rae in bod-interface.lisp, since that
;; triggers this.

(defvar *posh-graph-tool* nil)
(defvar *posh-edit-tool* nil)
(defvar *edit-length* 9 "default value for how many slots in the editor")

; Argh!  Why isn't there "scan-hash" or "keys" in lispworks?
(defun hash-keys (hash)
  (let ((result nil))
    (loop for x being each hash-key of hash
  do (setf result (cons x result)))
    result))

(defun symbol< (thing2 thing1)
  (string< (string thing2) (string thing1)))

(defun edit-posh-element (data unreferenced-interface)
  (unless *posh-edit-tool*
    (edit-bod-lap *bod-interface*))
;  (setf (capi:display-pane-text (get-fap-bit-dp *posh-edit-tool*))
; (string data))
  (palette-selector data *posh-edit-tool*)
  )

(defun no-more-fap-graph (&rest unreferenced-args)
  (setf *posh-graph-tool* nil)
  (if *bod-interface*
      (setf (capi:button-selected (get-graph-fap-cb *bod-interface*)) nil))
  )

(defun no-more-fap-edit (&rest unreferenced-args)
  (setf *posh-edit-tool* nil)
  (if *bod-interface*
      (setf (capi:button-selected (get-edit-fap-cb *bod-interface*)) nil))
  )

;; maybe this ought to discriminate the different levels of priority,
;; but right now they get erradicated with "append"

;; Note -- the graph shows all the things actually connected to drives
;; and potentially used by the agent's brain.  The lap file itself may
;; contain spare elements that are not currently connected.

(define-interface bod-fap-graph ()
  ()
  (:panes
   (script-name-pane
    title-pane
    :text "No one told me the script name"
    :reader get-graph-script-name
    :background :white)
   (fap-graph
    graph-pane
;;  :roots (mapcar #'drive-name (apply #'append (elements *drive-collection*)))
    :roots (list (name *drive-collection*))
    :node-pinboard-class 'push-button
;     :best-width 1000
;     :best-height 350
    :selection-callback 'edit-posh-element
    :children-function
    #'(lambda (x)
(let ((xx (find-element x)))
  (cond
   ((typep xx 'drive-collection)
    (mapcar #'drive-name (apply #'append (elements xx))))
   ((typep xx 'drive-element)
    (if (typep (action (drive-root xx)) 'sol-generic)
(list (name (action (drive-root xx)))))) ; else terminal
   ((typep xx 'sol-competence)
    (mapcar #'ce-label (apply #'append (elements xx))))
   ((typep xx 'competence-element)
    (if (typep (action  xx) 'sol-generic)
(list (name (action xx))))) ; else terminal
   ((typep xx 'sol-action-pattern)
    (ap-guts xx))
   ))))
   )
  (:layouts
   (column-layout-1
    column-layout
    '(script-name-pane fap-graph))
   )
  (:default-initargs
   :best-height 500
   :best-width 750
   :destroy-callback 'no-more-fap-graph ; destroy stuff hangs things
   :layout 'column-layout-1
   :title "POSH Script Graph"
   :background *color*))


;;; ------------------ LAP Editor ------------------- ;;;

(defvar edit-primitive-cl nil
  "set by (build-edit-primitive) below")
(defvar edit-action-pattern-cl nil
  "set by (build-edit-action-pattern) below")
(defvar edit-competence-cl nil
  "set by (build-edit-competence) below")
(defvar edit-drive-collection-cl nil
  "set by (build-edit-drive-collection) below")


; the first task here is to make the different palette items mutually
; exclusive. Note: this is getting used not only from
; palette-selector, so some of these types can’t actually be accessed
; that way.
(defun palette-selector (item interface)
  (let ((object nil) (type nil) (name item))
    ;select only the proper thing
    (if (get-sense item)
(setf object 'primitive type 'sense)
      (setf (choice-selected-item (get-sense-primitive-lp interface)) nil))
    (if (get-act item)
(setf object 'primitive type 'act)
      (setf (choice-selected-item (get-act-primitive-lp interface)) nil))
    (if (get-action-pattern item)
(setf object (get-action-pattern item) type 'action-pattern)
      (setf (choice-selected-item (get-action-pattern-lp interface)) nil))
    (if (get-competence item)
(setf object (get-competence item) type 'competence)
      (setf (choice-selected-item (get-competence-lp interface)) nil))
    ; don’t worry about the selection stuff if came from elsewhere...
    (if (not object)
(progn
  (cond
   ((get-competence-element item)
    (setf object (get-competence
  (competence (get-competence-element item)))
  type 'competence))
   ((and (get-drive-element item)
 (eq (drive-collection (get-drive-element item))
     (name *drive-collection*)))
    (setf object *drive-collection* type 'drive-collection))
   ((eq (name *drive-collection*) item)
    (setf object *drive-collection* type 'drive-collection))
   (t (error "palette-selector: <~s> is of unknown type" item))
   )
  (setf name (name object))
  ))
    ; now show the stuff
    (if (eq object 'primitive) ; primitive should maybe throw up an editor?
        (display-object-in-lap-editor name type interface)
      (display-object-in-lap-editor object type interface))
  ))

(defun display-object-in-lap-editor (object type interface)
  ;fix the name in editor...
  (setf (capi:display-pane-text (get-fap-bit-dp interface))
        (string-downcase (string (if (symbolp object)
                                     object (name object)))))
  ; fix the editor pane, and remember it
  (let ((sub-interface (lap-edit-type type interface)))
    ;now set the type, and fill in the slots...
    (cond
     ((or (eq type 'sense) (eq type 'act))
      (setf (capi:choice-selected-item (get-type-op interface)) 'none)
      ; this ought to pop up an editor window! TODO
      )
     ((eq type 'action-pattern)
      (setf (capi:choice-selected-item (get-type-op interface))
            'action-pattern)
      ; (fill-action-pattern object sub-interface) ;TODO + other two below
      )
     ((or (eq type 'competence) (eq type 'competence-element))
      (setf (capi:choice-selected-item (get-type-op interface))
            'competence))
     ((or (eq type 'drive-collection) (eq type 'drive-element))
      (setf (capi:choice-selected-item (get-type-op interface))
            'drive-collection))
     )
    )) ; defun display-object-in-lap-editor

; this doesn't work --- apparently the reader names aren't getting set
; by the building code below (grrr....)
(defun fill-action-pattern (ap pane)
  (do ((count 1)
       (priority 1 (+ priority 1))
       (prs (ap-guts ap) (cdr prs)))
      ((null prs) t)
    (do ((els (car prs) (cdr els))
 (p-slot (symbol-cat 'ap-priority (format nil "~d" count))
 (symbol-cat 'ap-priority (format nil "~d" count)))
 (prim-slot (symbol-cat 'ap-prim (format nil "~d" count))
    (symbol-cat 'ap-prim (format nil "~d" count)))
 )
((null els) t)
      (eval `(setf ,(apply p-slot (list pane))
   ,priority))
      (eval `(setf ,(apply prim-slot (list pane))
   ,(car els)))
      (setf count (+ count 1))
      )))
```

```
;needs to be :callback-type :item-interface
(defun lap-edit-type (choice interface)
  (setf (switchable-layout-visible-child (get-editing-sp interface))
(cond
 ((eq choice 'action-pattern) edit-action-pattern-cl)
 ((or (eq choice 'competence)
      (eq choice 'competence-element)) edit-competence-cl)
 ((or (eq choice 'drive-collection)
      (eq choice 'drive-element)) edit-drive-collection-cl)
 (t edit-primitive-cl)))
  ) ;lap-edit-type

(define-interface bod-fap-edit ()
  ()
  (:panes
   (palette-tp
    title-pane
    :text "Palette"
    :background *color*)
   (competence-lp
    list-panel
    :title "Competences"
    :background :black ; :brick-red
    :foreground :white
    :selection-callback 'palette-selector
    :callback-type :item-interface
    :selection nil
    :print-function 'string-downcase
    :reader get-competence-lp
    :items (sort (hash-keys *competences*) #'symbol<)
    :interaction :single-selection)
   (action-pattern-lp
    list-panel
    :title "Action Patterns"
    :background :blue ; :slate
    :foreground :white
    :selection-callback 'palette-selector
    :callback-type :item-interface
    :selection nil
    :print-function 'string-downcase
    :reader get-action-pattern-lp
    :items (sort (hash-keys *action-patterns*) #'symbol<)
    :interaction :single-selection)
   (act-primitive-lp
    list-panel
    :title "Acts"
    :background :brown ; :forest-green
    :foreground :white
    :selection-callback 'palette-selector
    :callback-type :item-interface
    :selection nil
    :print-function 'string-downcase
    :reader get-act-primitive-lp
    :items (sort (hash-keys *act-primitives*) #'symbol<)
    :interaction :single-selection)
   (sense-primitive-lp
    list-panel
    :title "Senses"
    :background :red; :indigo
    :foreground :white
    :selection-callback 'palette-selector
    :callback-type :item-interface
    :selection nil
    :print-function 'string-downcase
    :reader get-sense-primitive-lp
    :items (sort (hash-keys *sense-primitives*) #'symbol<)
    :interaction :single-selection)
   (lap-file-dp
    display-pane
    :title "        File:"
    :title-position :left
    :text (if *bod-interface*
      (display-pane-text (get-choose-fap-dp *bod-interface*)) "none")
    :visible-border t
    :min-width 200
    :background :white
    :reader get-lap-file-dp)
   (lap-lib-dp
    display-pane
    :title " Library:"
    :title-position :left
    :text (if (not *bod-interface*) "none"
      (string (choice-selected-item (get-library-op *bod-interface*))))
    :visible-border t
    :min-width 100
    :background :white
    :reader get-lap-lib-dp)
   (save-script-pb
    push-button
    :text "Save Changes"
```

```
    :selection-callback 'save-lap-script
    :callback-type :interface
    :reader get-save-script-pb
    :enabled nil
    :background *color*)
;;;
;;; Editing Pane <<<
;;;
   (edit-tp
    title-pane
    :text "Edit/View"
    :background *color*)
   (edit-view-cbp
    check-button-panel
    :items '("View" "Edit")
    :layout-class 'capi:row-layout
    :title-position :left
    :title "     "
    :background *color*
    :interaction :single-selection
    :selection 0
    :enabled nil)
   (new-fap-bit-pb
    push-button
    :title-position :left
    :title "               "
    :text "Create"
    :selection-callback 'new-fap-bit
    :callback-type :interface
    :reader get-new-fap-bit-pb
    :background *color*
    :enabled t)
   (fap-bit-dp
    display-pane
    :title "Name:"
    :text "None Chosen"
    :title-position :left
    :visible-border t
    :min-width 180
    :background :white
    :reader get-fap-bit-dp)
   (type-op
    option-pane
    :items '(none action-pattern competence drive-collection)
    :selected-item 0
```

```
     :title "Type:"
     :max-width 170
     :selection-callback 'lap-edit-type
     :callback-type :item-interface
     :print-function 'string-capitalize
     :reader get-type-op
     :background :white)
    (resort-elements-pb
     push-button
     :text "Resort"
     :title-position :left
     :title "  "
     :selection-callback 'resort-elements
     :callback-type :interface
     :reader get-resort-elements-pb
     :enabled t
     :background *color*)
    (editing-sp
     switchable-layout
     :reader get-editing-sp
     :description (list edit-primitive-cl edit-action-pattern-cl edit-competence-cl ed
    )
   (:layouts
    (edit-cl
     column-layout
     '(edit-full-title-cl editing-sp))
    (edit-full-title-cl
     column-layout
     '(edit-title-rl  edit-name-rl)
     :visible-border t
     :background *color*)
    (edit-title-rl
     row-layout
     '(edit-tp edit-view-cbp new-fap-bit-pb resort-elements-pb)
     :visible-border nil
     :background *color*)
    (edit-name-rl
     row-layout
     '(fap-bit-dp type-op)
     :visible-border nil
     :background *color*)
    (palette-cl
     column-layout
     '(palette-title-rl palette-rl))
    (palette-title-rl
     row-layout
     '(palette-tp lap-file-dp lap-lib-dp save-script-pb)
     :visible-border t
     :background *color*)
    (palette-rl
     row-layout
     '(competence-lp action-pattern-lp act-primitive-lp sense-primitive-lp)
     :visible-border t
     :background *color*)
    (top-rl
     row-layout
     '(edit-cl palette-cl))
    )
   (:default-initargs
    :layout 'top-rl
    :best-width 700
    :best-height 260
    :title "POSH Element Editor"
    :destroy-callback 'no-more-fap-edit  ; destroy stuff hangs things
    :background *color*))

 (defun write-to-me (item)
   (setf *write-here* item))

 ; this one doesn't let you do anything right now --- maybe it ought to
 ; let you edit the primitives or behavior files!
 (defun build-edit-primitive () ; (&optional (rows *edit-length*))
   (setf edit-primitive-cl
     (make-instance 'column-layout :description nil)))

 (defun build-edit-action-pattern (&optional (rows *edit-length*))
   (setf edit-action-pattern-cl
     (make-instance 'column-layout :description
       (cons
 (make-instance 'row-layout :description (list
           (make-instance 'text-input-pane :max-characters 2
  :max-width 26 :min-width 26
        :reader 'ap-priority1 :title "P" :title-position :top)
           (make-instance 'text-input-pane :max-width 180 :min-width 180
        :reader 'ap-prim1 :title "Sense or Action"
        :title-position :top
        :selection-callback 'write-to-me :callback-type :item)))
       (loop for ix from 2 to rows collect
 (make-instance 'row-layout :description (list
           (make-instance 'text-input-pane :max-characters 2
  :max-width 26 :min-width 26
        :reader (symbol-cat 'ap-priority (format nil "~d" ix)))
           (make-instance 'text-input-pane :max-width 180 :min-width 180
        :reader (symbol-cat 'ap-prim (format nil "~d" ix))
        :selection-callback 'write-to-me :callback-type :item)
           )))))))


 (defun build-edit-competence (&optional (rows *edit-length*))
   (setf edit-competence-cl
     (make-instance 'column-layout :description
       (cons
 (make-instance 'row-layout :description (list
           (make-instance 'text-input-pane :max-characters 2
  :max-width 26 :min-width 26
        :reader 'c-priority1 :title "P" :title-position :top)
           (make-instance 'text-input-pane :max-width 140 :min-width 140
        :reader 'c-label1 :title "Label" :title-position :top
        :selection-callback 'write-to-me :callback-type :item)
           (make-instance 'text-input-pane :max-characters 4
  :max-width 45 :min-width 45
        :reader 'c-retries1 :title "Retries" :title-position :top
        :selection-callback 'write-to-me :callback-type :item)
           (make-instance 'text-input-pane :max-width 220 :min-width 220
        :reader 'c-trigger1 :title "Trigger" :title-position :top
        :selection-callback 'write-to-me :callback-type :item)
   (make-instance 'text-input-pane :max-width 140 :min-width 140
        :reader 'c-element1 :title "Element" :title-position :top
        :selection-callback 'write-to-me :callback-type :item)))
       (loop for ix from 2 to rows collect
 (make-instance 'row-layout :description (list
           (make-instance 'text-input-pane :max-characters 2
  :max-width 26 :min-width 26
        :reader (symbol-cat 'c-priority (format nil "~d" ix)))
           (make-instance 'text-input-pane :max-width 140 :min-width 140
        :reader (symbol-cat 'c-label (format nil "~d" ix)))
           (make-instance 'text-input-pane :max-characters 4
  :max-width 45 :min-width 45
        :reader (symbol-cat 'c-retries (format nil "~d" ix)))
           (make-instance 'text-input-pane :max-width 220 :min-width 220
        :reader (symbol-cat 'c-trigger (format nil "~d" ix))
        :selection-callback 'write-to-me :callback-type :item)
   (make-instance 'text-input-pane :max-width 140 :min-width 140
        :reader (symbol-cat 'c-element (format nil "~d" ix))
        :selection-callback 'write-to-me :callback-type :item)
```

```
          )))))))

(defun build-edit-drive-collection (&optional (rows *edit-length*))
  (setf edit-drive-collection-cl
    (make-instance 'column-layout :description
      (cons
(make-instance 'row-layout :description (list
          (make-instance 'text-input-pane :max-characters 2
 :max-width 26 :min-width 26
       :reader 'dc-priority1 :title "P" :title-position :top)
          (make-instance 'text-input-pane :max-width 140 :min-width 140
       :reader 'dc-label1 :title "Label" :title-position :top
       :selection-callback 'write-to-me :callback-type :item)
          (make-instance 'text-input-pane :max-characters 4
 :max-width 45 :min-width 45
       :reader 'dc-frequency1 :title "Freq." :title-position :top
       :selection-callback 'write-to-me :callback-type :item)
          (make-instance 'text-input-pane :max-width 220 :min-width 220
       :reader 'dc-trigger1 :title "Trigger" :title-position :top
       :selection-callback 'write-to-me :callback-type :item)
  (make-instance 'text-input-pane :max-width 140 :min-width 140
       :reader 'dc-drive1 :title "Drive Root" :title-position :top
       :selection-callback 'write-to-me :callback-type :item)))
      (loop for ix from 2 to rows collect
(make-instance 'row-layout :description (list
          (make-instance 'text-input-pane :max-characters 2
 :max-width 26 :min-width 26
       :reader (symbol-cat 'dc-priority (format nil "~d" ix)))
          (make-instance 'text-input-pane :max-width 140 :min-width 140
       :reader (symbol-cat 'dc-label (format nil "~d" ix)))
          (make-instance 'text-input-pane :max-characters 4
 :max-width 45 :min-width 45
       :reader (symbol-cat 'dc-frequency (format nil "~d" ix)))
          (make-instance 'text-input-pane :max-width 220 :min-width 220
       :reader (symbol-cat 'dc-trigger (format nil "~d" ix))
       :selection-callback 'write-to-me :callback-type :item)
  (make-instance 'text-input-pane :max-width 140 :min-width 140
       :reader (symbol-cat 'dc-drive (format nil "~d" ix))
       :selection-callback 'write-to-me :callback-type :item)
          )))))))


(build-edit-primitive)
(build-edit-action-pattern)
(build-edit-competence)
```

```
(build-edit-drive-collection)
```

Appendix B

The Transitive-Inference Behavior Library

B.1 Interface

As I say in Chapter 8, the primitive interface file is the main documentation of what's going on in a library. Here's the interface for the transitive-inference task.

```
;; the primitives for the monkey experiments --- July 2000 JJB,
;; revised for learning Dec 2000 JJB

(add-act 'fail
  (lambda () nil))

(add-sense 'see-red
  (lambda () (see 'red)))

(add-sense 'see-white
  (lambda () (see 'white)))

(add-sense 'see-green
  (lambda () (see 'green)))

(add-sense 'see-yellow
  (lambda () (see 'yellow)))

(add-sense 'see-blue
  (lambda () (see 'blue)))

(add-sense 'grasping
  (lambda () (grasping)))  ; could use &optional here to make this general (?)
```

```
(add-act 'grasp-seen
  (lambda () (grasp)))

(add-act 'grasp-other
  (lambda () (grasp 'not)))

(add-act 'pick-other
  (lambda () (pick 'other)))

(add-act 'pick-this
  (lambda () (pick 'this)))

(add-act 'screech
  (lambda () (squeek)))

(add-act 'hoot
  (lambda () (call)))



;; adaptive behaviors in the monkey


(add-sense 'rewarded
  (lambda () *reward*))

(add-act 'adaptive-choice
  (lambda () (educated-choice)))

(add-act 'consider-reward
  (lambda () (learn-from-reward)))

(add-act 'fairly-consider-reward
  (lambda () (learn-from-reward-fairly)))

; dropped as not really interesting -- see local-prior-learn.lap
; (add-act 'locally-consider-reward
;  (lambda () (learn-locally-from-reward)))

(add-sense 'focus-rule
  (lambda () (get-focus-rule *color-rule-trick*)))

(add-act 'priority-focus
  (lambda () (focus-on-something *color-rule-trick*)))
```

```
(add-sense 'target-chosen
  (lambda () (not (null *attention*))))

(add-act 'rules-from-reward
  (lambda () (learn-rules-from-reward)))



;; these are really for the testing environment, not the monkey


(add-sense 'pending-test
  (lambda () *pending-tests*))

(add-act 'new-bt-test
  (lambda () (setf *test-board* (random-copy (pop *pending-tests*)))))

(add-act 'save-result
  (lambda () (print-test-result)))

(add-act 'save-mcg-result
  (lambda () (print-brendan-like-result)))

(add-act 'save-learning-results
  (lambda () (save-monkey-results *color-trick*)))

(add-act 'save-rule-learning-results
  (lambda () (save-monkey-results *color-rule-trick*)))

(add-sense 'no-test
  (lambda () (not *test-board*)))

(add-act 'finish-test
  (lambda () (end-test)))

(add-act 'new-test
  (lambda () (start-test)))

(add-act 'new-training-set
  (lambda () (give-training-set)))

(add-sense 'find-red
  (lambda () (find-in-test 'red)))
```

```
(add-sense 'find-white
  (lambda () (find-in-test 'white)))

(add-sense 'find-green
  (lambda () (find-in-test 'green)))

(add-sense 'find-yellow
  (lambda () (find-in-test 'yellow)))

(add-sense 'find-blue
  (lambda () (find-in-test 'blue)))

(add-act 'reward-found
  (lambda () (reward-monkey)))

;; regimented test generating behaviors

; if only one thing on board, sense it's color
(add-sense 'board-only
  (lambda () (if (cadr *test-board*) nil (car *test-board*))))

; sense what's in the monkey's hand
(add-sense 'hand
  (lambda () *hand*))

(add-act 'give-peanut
  (lambda () (setf *reward* 'peanut)))

(add-act 'buzzer
  (lambda () (setf *reward* 'no-thanks-for-playing)))

(add-sense 'criteria
  (lambda () (criteria *this-test*)))

(add-sense 'test-done
  (lambda () (test-over *this-test*)))

(add-act 'clean-up
  (lambda () (cleanup-time)))

(add-sense 'bigram-done
  (lambda () (binary-done *this-test*)))

(add-act 'check-criteria
  (lambda () (check-test-criteria *this-test*)))
```

```
(add-act 'pick-ngram
  (lambda () (set-test *this-test*)))

(add-act 'pick-pair
  (lambda () (set-pair *this-test*)))

; Stuff for me!

(add-sense 'accounted
  (lambda () *right*))

(add-act 'monkey-right
  (lambda () (setf *right* 1)))

(add-act 'monkey-wrong
  (lambda () (setf *right* 0)))
```

B.2 Scripts

Here are the script files, roughly in the order they were written.

```
; binary-test.lap   From June/July 2000, debugging lisp version  JJB
; test w/ white blue and green are the same level, just to test
;   (elements ((get-red (trigger ((see-red))) noisy-grasp))

((C elvis (minutes 10) (goal ((grasping)))
   (elements ((get-red (trigger ((see-red))) noisy-grasp))
     ((get-white (trigger ((see-white))) grasp-seen))
     ((get-blue (trigger ((see-blue))) grasp-seen))
     ((get-green (trigger ((see-green))) grasp-seen))
     ((get-yellow (trigger ((see-yellow))) grasp-seen))
     ))
(AP noisy-grasp (minutes 10) (screech grasp-seen)))


; driven-btest.lap (from binary-test)  17 July 2000, debug drives
; test w/ white blue and green are the same level, just to test
;   (elements ((get-red (trigger ((see-red))) noisy-grasp))

(
 (C elvis-choice (minutes 10) (goal ((grasping)))
    (elements ((get-red (trigger ((see-red))) noisy-grasp))
      ((get-white (trigger ((see-white))) grasp-seen))
      ((get-blue (trigger ((see-blue))) grasp-seen))
      ((get-green (trigger ((see-green))) grasp-seen))
      ((get-yellow (trigger ((see-yellow))) grasp-seen))
      ))
 (AP noisy-grasp (minutes 10) (screech grasp-seen))
 (DC life (goal (fail))
     (drives
      ((set-puzzle (trigger ((no-test))) new-test))
      ((reward (trigger ((grasping))) finish-test))
      ((choose (trigger ((grasping nil))) elvis-choice))
      ((complain (trigger nil) hoot)))
     )
 )


; prior-learn.lap (from driven-btest from binary-test) 30 Dec 2000,
; debug drives test w/ white blue and green are the same level, just
; to test (elements ((get-red (trigger ((see-red))) noisy-grasp))

(
 (C elvis-reward (minutes 10) (goal ((rewarded)))
    (elements ((get-red (trigger ((find-red))) reward-found))
      ((get-white (trigger ((find-white))) reward-found))
      ((get-blue (trigger ((find-blue))) reward-found))
      ((get-green (trigger ((find-green))) reward-found))
      ((get-yellow (trigger ((find-yellow))) reward-found))
      ))
 (AP educated-grasp (minutes 10) (adaptive-choice grasp-seen))
 (AP end-of-test (minutes 10) (consider-reward save-learning-results finish-test))
 (DC life (goal (fail))
     (drives
      ((set-puzzle (trigger ((no-test))) new-test))
      ((reward (trigger ((grasping)(rewarded nil))) elvis-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger nil) hoot)))
     )
 )


; fair-prior-learn.lap (from prior-learn, driven-btest, binary-test) 8
; Feb 2001 JJB
; Different from prior-learn in that only trained on adjacent pairs,
; and negative reward is more realistic.
```

```
(
 (C elvis-reward (minutes 10) (goal ((rewarded)))
    (elements ((get-red (trigger ((find-red))) reward-found))
      ((get-white (trigger ((find-white))) reward-found))
      ((get-blue (trigger ((find-blue))) reward-found))
      ((get-green (trigger ((find-green))) reward-found))
      ((get-yellow (trigger ((find-yellow))) reward-found))
      ))
 (AP educated-grasp (minutes 10) (adaptive-choice grasp-seen))
 (AP end-of-test (minutes 10) (fairly-consider-reward save-learning-results finish-te
 (DC life (goal (fail))
     (drives
      ((set-puzzle (trigger ((no-test))) new-training-set))
      ((reward (trigger ((grasping)(rewarded nil))) elvis-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger nil) hoot)))
     )
 )


; local-prior-learn.lap (from fair-prior-learn, prior-learn,
; driven-btest, binary-test) 9 Feb 2001 JJB
; Different from fair-prior-learn in that training only affects
; weights of present members.  Not finished - decided not important

(
 (C elvis-reward (minutes 10) (goal ((rewarded)))
    (elements ((get-red (trigger ((find-red))) reward-found))
      ((get-white (trigger ((find-white))) reward-found))
      ((get-blue (trigger ((find-blue))) reward-found))
      ((get-green (trigger ((find-green))) reward-found))
      ((get-yellow (trigger ((find-yellow))) reward-found))
      ))
 (AP educated-grasp (minutes 10) (adaptive-choice grasp-seen))
 (AP end-of-test (minutes 10) (locally-consider-reward save-learning-results finish-t
 (DC life (goal (fail))
     (drives
      ((set-puzzle (trigger ((no-test))) new-training-set))
      ((reward (trigger ((grasping)(rewarded nil))) elvis-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger nil) hoot)))
     )
 )
```

```
; roger-test.lap (from driven-btest, binary-test)  8 Feb 2000
; Make something more like Harris' model

(
 (C roger-choice (minutes 10) (goal ((grasping)))
    (elements ((get-red (trigger ((see-red))) grasp-seen))
      ((get-not-yellow (trigger ((see-yellow))) grasp-other))
      ((get-blue (trigger ((see-white))) grasp-seen))
      ((get-green (trigger ((see-blue))) grasp-seen))
      ))
 (AP record-finish-test (minutes 10) (save-mcg-result finish-test))
 (DC life (goal ((no-test) (pending-test nil) hoot))
     (drives
      ((set-puzzle (trigger ((no-test))) new-bt-test))
      ((reward (trigger ((grasping))) record-finish-test))
      ((choose (trigger ((grasping nil))) roger-choice))
      ((complain (trigger nil) hoot)))
     )
 )


; rule-learn.lap (from prior-learn from driven-btest from binary-test)
; 18 Jan 2001 > made operational 9 Feb, some stuff from fair-prior-learn.lap
; Learn priorities of rules, not just priority of items.

(
 (C elvis-reward (minutes 10) (goal ((rewarded)))
    (elements ((get-red (trigger ((find-red))) reward-found))
      ((get-white (trigger ((find-white))) reward-found))
      ((get-blue (trigger ((find-blue))) reward-found))
      ((get-green (trigger ((find-green))) reward-found))
      ((get-yellow (trigger ((find-yellow))) reward-found))
      ))
 (C educated-grasp (minutes 10) (goal ((grasping)))
   (elements ((grasp-it (trigger ((target-chosen))) grasp-seen))
     ((avoid-it (trigger ((focus-rule 'avoid))) pick-other))
     ((select-it (trigger ((focus-rule 'select))) pick-this))
     ((focus-it (trigger ()) priority-focus))
     ))
 (AP end-of-test (minutes 10) (rules-from-reward save-rule-learning-results finish-te
 (DC life (goal (fail))
     (drives
      ((set-puzzle (trigger ((no-test))) new-training-set))
      ((reward (trigger ((grasping)(rewarded nil))) elvis-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger nil) hoot)))
     )
 )


; BAD!!! child-test.lap (from rule-learn, prior-learn driven-btest
; binary-test) 22 Jan 2001 -- JJB.  Learn rules, not just priority of
; items, give a complicated training regime.  Demos a way to interface
; to a "reasoning system" (in this case, just an algorithm) for
; determining the course of study.  Never actually used --- behavior
; handles more of the picking-test than this.  See test-child.lap

(
 (C elvis-reward (minutes 10) (goal ((rewarded)))
    (elements ((bad-red (trigger ((board-only 'red) (hand 'white))) buzzer))
      ((bad-white (trigger ((board-only 'white)(hand 'blue))) buzzer))
      ((bad-blue (trigger ((board-only 'blue)(hand 'green))) buzzer))
      ((bad-green (trigger ((board-only 'green)(hand 'yellow))) buzzer))
      ((ok-other (trigger ()) give-peanut))
      ))
 (C pick-test (minutes 10) (goal ((no-test nil)))
    (elements ((set-trigram (trigger ((criteria 3) (bigram-done)) new-trigram)))
      ((set-test-bigram (trigger ((criteria 3))) new-bigram))
      ((set-shuffle-pair (trigger ((criteria 2))) new-pair))
      ((set-train-pair (trigger ((criteria 1))) ordered-pair))
      ((set-each-pair (trigger ()) clustered-pair))
      ))
 (C educated-grasp (minutes 10) (goal ((grasping)))
   (elements ((grasp-it (trigger ((target-chosen))) grasp-seen))
     ((avoid-it (trigger ((focus-rule 'avoid))) pick-other))
     ((select-it (trigger ((focus-rule 'select))) pick-this))
     ((focus-it (trigger ()) priority-focus))
     ))
 (AP end-of-test (minutes 10) (rules-from-reward save-rule-learning-results finish-te
 (DC life (goal ((test-done)))
     (drives
      ((set-puzzle (trigger ((no-test))) pick-test))
      ((reward (trigger ((grasping)(rewarded nil))) elvis-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger nil) hoot)))
     )
 )


; test-child.lap (from child-test, rule-learn, prior-learn driven-btest
; binary-test) 22 Jan 2001 -- JJB.  Learn rules, not just priority of
; items, give a complicated training regime.  Demos a way to interface
; to a "reasoning system" (in this case, just an algorithm) for
; determining the course of study.

(
 (C elvis-reward (minutes 10) (goal ((rewarded)))
    (elements ((bad-red (trigger ((board-only 'red) (hand 'white))) buzzer)
       (bad-white (trigger ((board-only 'white)(hand 'blue))) buzzer)
       (bad-blue (trigger ((board-only 'blue)(hand 'green))) buzzer)
       (bad-green (trigger ((board-only 'green)(hand 'yellow))) buzzer))
      ((ok-other (trigger ()) give-peanut))
              ))
 (C pick-test (minutes 10) (goal ((no-test nil)))
    (elements ((set-test (trigger ((criteria 3))) pick-ngram))
              ((set-pair (trigger ()) pick-pair))
              ))
 (C educated-grasp (minutes 10) (goal ((grasping)))
   (elements ((grasp-it (trigger ((target-chosen))) grasp-seen))
     ((avoid-it (trigger ((focus-rule 'avoid))) pick-other)
      (select-it (trigger ((focus-rule 'select))) pick-this))
     ((focus-it (trigger ()) priority-focus))
     ))
 (AP end-of-test (minutes 10) (rules-from-reward check-criteria finish-test))
 (DC life (goal (test-done))
     (drives
      ((set-puzzle (trigger ((no-test))) pick-test))
      ((reward (trigger ((grasping)(rewarded nil))) elvis-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger ()) hoot)))
     )
 )

; educate-monkey.lap (from test-child.lap, child-test, rule-learn,
; prior-learn driven-btest binary-test) 10 Feb 2001 -- JJB.  Learn
; rules, not just priority of items, give a complicated training
; regime.  Demos a way to interface to a "reasoning system" (in this
; case, just an algorithm) for determining the course of study.
; Unfortunately, same priority stuff isn't being read in properly from
; the script, so this is a dumb hack of test-child -- it's based on
; maggie's child-training & testing stuff.  Note that you only
; penalize in the case of training pairs done incorrectly.

(
 (C selective-reward (minutes 10) (goal ((rewarded)))
    (elements ((bad-red (trigger ((board-only 'red) (hand 'white))) buzzer))
      ((bad-white (trigger ((board-only 'white)(hand 'blue))) buzzer))
      ((bad-blue (trigger ((board-only 'blue)(hand 'green))) buzzer))
      ((bad-green (trigger ((board-only 'green)(hand 'yellow))) buzzer))
      ((ok-other (trigger ()) give-peanut))
              ))
 (C pick-test (minutes 10) (goal ((no-test nil)))
    (elements ((set-test (trigger ((criteria 3))) pick-ngram))
              ((set-pair (trigger ()) pick-pair))
              ))
 (C educated-grasp (minutes 10) (goal ((grasping)))
   (elements ((grasp-it (trigger ((target-chosen))) grasp-seen))
     ((avoid-it (trigger ((focus-rule 'avoid))) pick-other))
     ((select-it (trigger ((focus-rule 'select))) pick-this))
     ((focus-it (trigger ()) priority-focus))
     ))
 (AP end-of-test (minutes 10) (rules-from-reward save-rule-learning-results check-cri
 (DC life (goal (test-done clean-up))
     (drives
      ((set-puzzle (trigger ((no-test))) pick-test))
      ((reward (trigger ((grasping)(rewarded nil))) selective-reward))
      ((get-reward (trigger ((rewarded))) end-of-test))
      ((choose (trigger ((grasping nil))) educated-grasp))
      ((complain (trigger nil) hoot)))
     )
 )

; educate-me+monk.lap (from educate-monkey, test-child, child-test,
; rule-learn, prior-learn driven-btest binary-test) 10 Feb 2001 --
; JJB.  Minor details -- keep track of real right answer for my sake,
; not just whether to reward.  Uses nested comps

(
 (C selective-reward (minutes 10) (goal ((rewarded)))
    (elements ((bad-red (trigger ((board-only 'red) (hand 'white))) buzzer))
      ((bad-white (trigger ((board-only 'white)(hand 'blue))) buzzer))
      ((bad-blue (trigger ((board-only 'blue)(hand 'green))) buzzer))
      ((bad-green (trigger ((board-only 'green)(hand 'yellow))) buzzer))
      ((ok-other (trigger ()) give-peanut))
              ))
```

```
(C record-correctness (minutes 10) (goal ((rewarded)))
   (elements ((do-reward (trigger ((accounted))) selective-reward))
     ((my-bad-red (trigger ((find-red) (hand 'white))) monkey-wrong))
     ((my-bad-white (trigger ((find-white)(hand 'blue))) monkey-wrong))
     ((my-bad-blue (trigger ((find-blue)(hand 'green))) monkey-wrong))
     ((my-bad-green (trigger ((find-green)(hand 'yellow))) monkey-wrong))
     ((my-ok-other (trigger ()) monkey-right))
             ))
(C pick-test (minutes 10) (goal ((no-test nil)))
   (elements ((set-test (trigger ((criteria 3))) pick-ngram))
             ((set-pair (trigger ()) pick-pair))
             ))
(C educated-grasp (minutes 10) (goal ((grasping)))
  (elements ((grasp-it (trigger ((target-chosen))) grasp-seen))
    ((avoid-it (trigger ((focus-rule 'avoid))) pick-other))
    ((select-it (trigger ((focus-rule 'select))) pick-this))
    ((focus-it (trigger ()) priority-focus))
    ))
(AP end-of-test (minutes 10) (rules-from-reward save-rule-learning-results check-cri
(DC life (goal (test-done hoot clean-up))
    (drives
     ((set-puzzle (trigger ((no-test))) pick-test))
     ((reward (trigger ((grasping)(rewarded nil))) record-correctness))
     ((get-reward (trigger ((rewarded))) end-of-test))
     ((choose (trigger ((grasping nil))) educated-grasp))
     ((complain (trigger nil) hoot)))
    )
)
```

B.3 Behavior Library

Here is almost all the code for the behaviors. You can see that I wasn't actually that careful about being object-oriented for this library!

```
;; The monkey "behavior libraries" -- currently actually just functions.
;; Remember -- all actions should normally return true.  If they return
;; nil, they will kill any AP they are in.

; some simple predicates

(defun see (ordinat)
  (do ((tb *test-board* (cdr tb)))
      ((not tb) nil)
    (if (equal (car tb) ordinat)
        (progn
          (setf *attention* (car tb))
          (return t)))
  ))

(defun find-in-test (ordinat)
  (if (do ((tb *test-board* (cdr tb)))
      ((not tb) nil)
      (if (equal (car tb) ordinat)
  (progn
          (setf *attention* *test-board*)
          (return t)))
      )
      t
    (if (equal *hand* ordinat) (setf *attention* *hand*))
    ))

(defvar *reward* nil)

(defun reward-monkey ()
  (setf *reward* (if (eq *attention* *hand*)
     'peanut
   'no-thanks-for-playing)))

;; see if thing is in hand if thing; else just is anything in hand?
(defun grasping (&optional (thing nil))
  (if
      thing
      (eq thing *hand*)
    *hand*))

; some simple acts...

(defun release-grip ()
  (if *hand*
      (progn
        (setf *floor* (cons *hand* *floor*))
        (setf *hand* nil)))
  t)

(defun grasp (&optional (graspable nil))
  (if (not graspable)
      (setf graspable *attention*))
  (cond
   ((grasping graspable) t)  ;; already doing it, return true
   ((grasping) nil)          ;; return fail if already holding something
   ((eq graspable 'not)      ;; grasp something I'm not looking at
    (let ((object (random-one-of
   (loop for ixi in *test-board*
 unless (eq ixi *attention*)
 collect ixi))))
      (setf *hand* object)
      (setf *test-board* (delete object *test-board* :count 1))
      ))
   ((not (see graspable)) ;; return fail if can't see it
    nil)
   (t       ;; otherwise...
    (setf *hand* graspable)
    (setf *test-board* (delete graspable *test-board* :count 1))
    ))
  t)

(defun squeek ()
  (format *posh-output* "~%~%eek! eek!~%~%")
  t)

(defun call ()
  (format *posh-output* "~%~%OOOOooo! ooo ooo oo!~%~%")
  t)

;; the learning behavior

; this obviously learns pretty quickly, but I'm in a hurry.
(defclass sequence-learner ()
  (
   (seq :accessor seq :initarg :seq :initform nil)
   (sig-dif :accessor sig-dif :initarg :sig-dif :initform .08)
   (weight-shift :accessor weight-shift :initarg :weight-shift :initform .02)
   ))

(defvar *color-trick* nil "declared in reset at bottom of file")

(defun reset-color-trick ()
  (setf *color-trick* (make-instance 'sequence-learner)))

(defmethod get-item ((sl sequence-learner) label)
  (let ((item (assoc label (seq sl))))
    (unless item (setf item (add-element sl label)))
    item
    ))
```

```
; give new element average weight, and reduce weight of competitiors
; to scale => sum of weights is always 1.0
(defmethod add-element ((sl sequence-learner) label)
  (let ((n (length (seq sl))))
    (setf (seq sl)
          (cons (cons label (/ 1.0 (+ n 1.0)))
                (mapcar
                 #'(lambda (x)
                     (setf (cdr x) (* (cdr x) (/ n (+ n 1.0)))) x)
                 (seq sl))
                )))
  (get-item sl label)
  )

; sum of weights should always be 1.0
(defmethod renormalize ((sl sequence-learner))
  (let* ((sumweights (loop for ixi in (seq sl)
   sum (cdr ixi)))
 (normalizer (/ 1.0 sumweights)))
    (setf (seq sl)
  (mapcar
   #'(lambda (x)
       (setf (cdr x) (* (cdr x) normalizer)) x)
     (seq sl))
                )))

; due to the invariant mentioned above, this should never be called
; directly rather, something should shift weight from one item to
; another or some other such normalizing scheme.
(defmethod update-weight ((sl sequence-learner) label value)
  (do ((xix (car (seq sl)) (car xsl))
       (xsl (cdr (seq sl)) (cdr xsl)))
      ((null xix) nil)
    (when (eq label (car xix))
      (setf (cdr xix) value)
      (return t))
    ))

; Pick the highest valued available option.  One could make this more
; interesting by making result stochastic if values close to each
; other.  But so long as learning keeps happening then changes happen
; anyway in proportion, and in proportion to uncertainty.
(defmethod make-choice ((sl sequence-learner) &optional (list nil))
  (let ((ls (sort
     (if list    ; subset of possible options
 (loop for ixi in list
       collect (get-item sl ixi))
       (seq sl)) ; else, choose from all options
     #'(lambda (x y) (< (cdr y) (cdr x))))
    ))
    (if (null ls) nil (caar ls))
    )) ; defmethod make-choice

(defmethod second-choice ((sl sequence-learner) &optional (list nil))
  (if (null (cdr (seq sl))) nil
    (cadr (sort
    (if list    ; subset of possible options
(loop for ixi in list
      collect (get-item sl ixi))
      (seq sl)) ; else, choose from all options
    #'(lambda (x y) (< (cdr y) (cdr x))))
   )))

; this ignores some data when testing with triads, but those tests
; weren't done for learning, only for evaluation.
(defun learn-from-reward ()
  (let* ((sl *color-trick*)
         (choice (get-item sl *hand*))
         (reject (get-item sl (car *test-board*))))
    (if (eq *reward* 'peanut)
        (good-monkey sl choice reject)
      (bad-monkey sl choice reject))
    t))


; only learn if something surprising happened
(defmethod good-monkey ((sl sequence-learner) choice reject)
  (unless (> (- (cdr choice) (cdr reject)) (sig-dif sl))
    (setf (cdr choice) (+ (cdr choice) (weight-shift sl)))
    (renormalize sl)
    ))

; you thought good-monkey was a hack...
(defmethod bad-monkey ((sl sequence-learner) choice reject)
  (let ((temp (cdr choice)))
    (setf (cdr choice) (cdr reject))
    (setf (cdr reject) temp)))

; a more general / useful method.  Ignores being certain if wrong,
; which really would probably trigger some big flag / conscious
; context which might accelerate learning of the new situation.
(defmethod reenforce ((sl sequence-learner) choice reject right)
  (unless reject (setf reject (second-choice sl)))
  (if right
      (unless (> (- (cdr choice) (cdr reject)) (sig-dif sl))
; (break)
(setf (cdr choice) (+ (cdr choice) (weight-shift sl)))
(renormalize sl)
)
    (progn ; else wrong.  See note above about certainty.
      (setf (cdr reject) (+ (cdr reject) (weight-shift sl)))
      (renormalize sl)))
  (list choice reject)
  ) ; defmethod reenforce

(defun learn-from-reward-fairly ()
  (let* ((sl *color-trick*)
         (choice (get-item sl *hand*))
         (reject (get-item sl (car *test-board*))))
    (reenforce sl choice reject (eq *reward* 'peanut))
    t))

; should really do something indeterminant if two are too close, but
; this is the first cut
(defun educated-choice ()
  (setf *attention* (make-choice *color-trick* *test-board*))
  )

;; Dealing with Rules

(defclass rule-learner ()
  (
   ; this is for the context -- you need to learn what's most
   ; important to attend to
   (attendants :accessor attendants :initarg :attendants
       :initform (make-instance 'sequence-learner :sig-dif .09))
   ; this is the rules for each element of the context
   (rule-seqs :accessor rule-seqs :initarg :rule-seqs
      :initform (make-hash-table))
   ; this is which rule set we're currently pondering
   (current-focus :accessor current-focus :initarg :current-focus :initform nil)
   (current-rule :accessor current-rule :initarg :current-rule :initform nil)
   ))
```

```
(defvar *color-rule-trick* nil "declared in reset at bottom of file")

(defun reset-color-rule-trick ()
  (if *color-rule-trick* (end-test))
  (setf *color-rule-trick* (make-instance 'rule-learner))
  (setf *this-test* (make-instance 'test-giver))
  )

(defmethod clear-focus ((rl rule-learner))
  (setf (current-focus rl) nil)
  (setf (current-rule rl) nil)
  )

(defmethod focus-on-something ((rl rule-learner))
  (setf (current-focus rl) (make-choice (attendants rl) *test-board*))
  )

(defmethod get-focus-rule-seq ((rl rule-learner))
  (let ((res (gethash (current-focus rl) (rule-seqs rl))))
    (if res res
      (setf (gethash (current-focus rl) (rule-seqs rl))
    (make-instance
     'sequence-learner :sig-dif .3
     :seq (random-copy (list (cons 'avoid 0.5) (cons 'select  0.5)))
     )))
    )) ;defmethod get-focus-rule-seq

(defmethod get-focus-rule ((rl rule-learner))
  (cond
   ((current-rule rl) (current-rule rl))
   ((current-focus rl)
    (setf (current-rule rl)
  (make-choice (get-focus-rule-seq rl))))
   (t nil)
   ))

(defun random-one-of (list)
  (nth (random (length list)) list))

(defun pick (which)
  (let ((rl *color-rule-trick*))
    (setf *attention*
  (cond
   ((eq which 'this) (current-focus rl))
   ((eq which 'other) (random-one-of
       (loop for ixi in *test-board*
     unless (eq ixi (current-focus rl))
       collect ixi)))
   ))))

; We have to learn about both the rule and the item.  We don't do
; anything complicated for backprop, we just blame both!
(defun learn-rules-from-reward ()
  (let* ((rl *color-rule-trick*)
         (chosen (get-item (attendants rl) (current-focus rl)))
 (reject nil)
 (goodness (eq *reward* 'peanut))
 (focus-rule-seq (get-focus-rule-seq rl)))
    (setf reject (get-item (attendants rl)
   (if (eq (current-rule rl) 'avoid)
       (setf reject *hand*)
     (car *test-board*))))
    (reenforce (attendants rl) chosen reject goodness)
    (reenforce (get-focus-rule-seq rl)
       (get-item focus-rule-seq (current-rule rl)) nil goodness)
    t))

; should really do something indeterminant if two are too close, but
; this is the first cut
(defun educated-choice ()
  (setf *attention* (make-choice *color-trick* *test-board*))
  )

;;; This stuff isn't really the monkey, it's the test, but this little
;;; program is basically running both conceptual agents...

;;; Sense-Predicates


;;; Actions

(defvar *elvis-training-pairs*
  '((yellow green) (green blue) (blue white) (white red)))

; standard test suite
(defun reset-training-set ()
  (setf *training-set* (random-copy *elvis-training-pairs*)))

(defun give-training-set ()
  (unless *training-set* (reset-training-set))
  (setf *test-board* (random-copy (pop *training-set*))))

; standard test suite
(defun set-pending-tests ()
  (setf *pending-tests* (make-all-tests)))

(defun make-all-tests ()
  (apply #'append (loop for ixi from 1 to 10
collect (append (random-copy *possible-binaries*)
(random-copy *possible-trinaries*)))))

; this is totally special purpose, (I'm obviously in a hurry)
; results in standard A-high results, which are the opposite of mcg
(defun print-which-test ()
  (let ((test (cons *hand* *test-board*))
(symb nil))
    (if (member 'red test) (setf symb 'a))
    (if (member 'white test) (if symb (setf symb 'ab) (setf symb 'b)))
    (if (member 'blue test) (if symb (setf symb (symbol-cat symb 'c))
      (setf symb 'c)))
    (if (member 'green test) (if symb (setf symb (symbol-cat symb 'd))
      (setf symb 'd)))
    (if (member 'yellow test) (setf symb (symbol-cat symb 'e)))
    symb
    ))

(defun print-which-result ()
  (cond
   ((eq *hand* 'red) 'a)
   ((eq 'white *hand*) 'b)
   ((eq 'blue *hand*) 'c)
   ((eq 'green *hand*) 'd)
   ((eq 'yellow *hand*) 'e)
   ))

(defun brendan-which-test ()
  (let ((test (cons *hand* *test-board*))
(symb nil))
    (if (member 'red test) (setf symb 'e))
    (if (member 'white test) (if symb (setf symb 'de) (setf symb 'd)))
    (if (member 'blue test) (if symb (setf symb (symbol-cat 'c symb))
      (setf symb 'c)))
    (if (member 'green test) (if symb (setf symb (symbol-cat 'b symb))
      (setf symb 'b)))
    (if (member 'yellow test) (setf symb (symbol-cat 'a symb)))
    symb
    ))

(defun brendan-which-result ()
  (cond
   ((eq *hand* 'red) 'e)
   ((eq 'white *hand*) 'd)
   ((eq 'blue *hand*) 'c)
   ((eq 'green *hand*) 'b)
   ((eq 'yellow *hand*) 'a)
   ))

(defun print-test-result ()
  (format *monkey-results-file* "~@4s ->   ~s~%"
  (print-which-test) (print-which-result))
  (force-output *monkey-results-file*)
  t)

(defun print-brendan-like-result ()
  (format *monkey-results-file* "~@4s ->   ~s~%"
  (brendan-which-test) (brendan-which-result))
  (force-output *monkey-results-file*)
  t)


(defvar *num-o-tests* 0)
(defvar *past-n-correct* 0)

(defun reset-monkey-counting ()
  (setf *num-o-tests* 0)
  (setf *past-n-correct* 0))

(defmethod print-monkey ((sl sequence-learner))
  (seq sl))

; takes care of printing seq when there wasn't a hash entry yet (HACK!)
(defmethod seq ((thing symbol))
  thing)

(defmethod print-monkey ((rl rule-learner))
  (do
   ((option-list
     (sort (seq (attendants rl))
   #'(lambda (x y) (< (cdr y) (cdr x)))) (cdr option-list))
    (res nil))
   ((null option-list) res)
   (setf res (cons (list (car option-list)
 (seq (gethash (caar option-list) (rule-seqs rl))))
   res))
   )) ; defmethod print-monkey rl

(defun save-monkey-results (monkey &optional (n 10))
  (setf *num-o-tests* (+ *num-o-tests* 1))
  (format *monkey-results-file* "~%~@4d: ~@20s ~@10s ~d ~d"
  *num-o-tests* *test-board* *hand* (if (eq *reward* 'peanut) 1 0)
  *right*)
  (when (eq *reward* 'peanut) (setf *past-n-correct* (+ 1 *past-n-correct*)))
  (when (eq 0 (mod *num-o-tests* n))
    (format *monkey-results-file* "~%CORRECT: ~d/~d  MONKEY: ~s"
    *past-n-correct* n (print-monkey monkey))
    (force-output *monkey-results-file*)
    (setf *past-n-correct* 0))
  t
  )

; note, some of this would be ascribed to the apparatus, some to the monkey
(defun end-test ()
  (setf *hand* nil)
  (setf *attention* nil)
  (setf *test-board* nil)
  (setf *reward* nil)
  (setf *right* nil) ; mostly  in -world & -prims
  (clear-focus *color-rule-trick*)
  t)

(defun start-test ()
  (setf *test-board* (two-of *colors*))
  )


;; the more complicated testing machine

; phase 1: each pair in order repeated til learned. crit: 9 of 10
; correct.  rej: over 30 trials

;phase 2a: 4 of each pair in order.  crit: 32 trials correct. rej:
;over 200 trials
;phase 2b: 2 of each pair in order.  crit: 16 trials correct. rej:
;over 200 trials
;phase 2c: 1 of each pair in order, 6x  (no crit for rej)
```

```
;phase 3: 1 of each pair in random order. crit: 24 trials correct.
;rej: over 200 trials

;test 1: binary tests: 6 sets of 10 pairs in random order.  reward
;unless messed up training pair.

;test 2a: as in phase3 for 32 trials.  unless 90% correct, redo phase 3

;test 2: 6 sets of 10 trigrams in random order, reward for all.

;test 3: extend that test

(defclass test-giver ()
  (
   (test-phase :accessor test-phase :initarg :test-phase :initform 'p1)
   (criteria :accessor criteria :initarg :criteria :initform 0)
   (test-over :accessor test-over :initarg :test-over :initform nil)
   (binary-done :accessor binary-done :initarg :binary-done :initform nil)
   (desired :accessor desired :initarg :desired :initform
     '(yellow green blue white red))
   (test-set :accessor test-set :initarg :test-set :initform
     '(green blue white red))
   (num-correct :accessor num-correct :initarg :num-correct :initform 0)
   (num-trials :accessor num-trials :initarg :num-trials :initform 0)
   (last-two-bad :accessor last-two-bad :initarg :last-two-bad :initform nil)
   ))

(defvar *this-test* (make-instance 'test-giver))
(defvar *possible-binaries*
  '((yellow green) (yellow blue)(yellow white) (yellow red)
    (green blue) (green white) (green red) (blue white) (blue red)
    (white red)))
(defvar *possible-trinaries*
  '((yellow green blue) (yellow green white) (yellow green red)
    (yellow blue white) (yellow blue red) (yellow white red)
    (green blue white) (green blue red) (green white red)
    (blue white red)))

; note num-trials gets incremented in let binding
(defmethod check-test-criteria ((tt test-giver))
  (let ((good (eq *reward* 'peanut))
(numtrials (setf (num-trials tt) (+ 1 (num-trials tt)))))
    (when good (setf (num-correct tt) (+ 1 (num-correct tt))))
    (cond
     ((eq (test-phase tt) 'p1)
      (if good
  (when (9-of-10? tt)
    (pop (test-set tt))
    (if (null (test-set tt))(test-phaser tt)
      (progn (setf (last-two-bad tt) nil) (reset-nums tt))))
(progn (when (> numtrials 30) (setf (test-over tt) 'fail))
       (when (cadr (last-two-bad tt)) (nbutlast (last-two-bad tt)))
       (push (num-trials tt) (last-two-bad tt)))))
     ((eq (test-phase tt) 'p2a)
      (pop (test-set tt))
      (when (not good) (setf (num-correct tt) 0)) ; this crit. must be consec.
      (when (< 31 (num-correct tt))
(test-phaser tt))
      (when (null (test-set tt))
(set-test-set tt))
      (when (> numtrials 200) (setf (test-over tt) 'fail)))
     ((eq (test-phase tt) 'p2b)
      (pop (test-set tt))
      (when (< 15 (num-correct tt))
(test-phaser tt))
      (when (not good) (setf (num-correct tt) 0)) ; this crit. must be consec.
      (when (null (test-set tt))
(set-test-set tt))
      (when (> numtrials 200) (setf (test-over tt) 'fail)))
     ((eq (test-phase tt) 'p2c)
      (pop (test-set tt))
      (when (< 35 (num-correct tt))
(test-phaser tt))
      (when (not good) (setf (num-correct tt) 0)) ; this crit. must be consec.
      (when (null (test-set tt))
(set-test-set tt))
      (when (> numtrials 200) (setf (test-over tt) 'fail)))
     ((eq (test-phase tt) 'p3)
      (pop (test-set tt))
      (when (< 24 (num-correct tt))
(test-phaser tt))
      (when (not good) (setf (num-correct tt) 0)) ; this crit. must be consec.
      (when (null (test-set tt))
(set-test-set tt))
      (when (> numtrials 200) (setf (test-over tt) 'fail)))
     ((eq (test-phase tt) 't1)
      (pop (test-set tt))
      (when (null (test-set tt))
(test-phaser tt)))
```

```
      ((eq (test-phase tt) 't2a)
       (pop (test-set tt))
       (when (null (test-set tt))
    (test-phaser tt)))
      ((eq (test-phase tt) 't2)
       (pop (test-set tt))
       (when (null (test-set tt))
(test-phaser tt)))
      ((eq (test-phase tt) 't3)
       (pop (test-set tt))
       (when (null (test-set tt))
(test-phaser tt)))
      ) t
    )) ;defmethod test-giver


; this is called whenever a new test set is needed -- it may or may
; not advance the phases and criteria.  Note criteria like "two
; succesful runs" have been collapsed into # of trials.
(defmethod test-phaser ((tt test-giver))
  (cond
   ((eq (test-phase tt) 'p1)
    (record-and-reset-nums tt)
    (setf (test-phase tt) 'p2a (criteria tt) 1)
    (set-test-set tt))
   ((eq (test-phase tt) 'p2a)
    (record-and-reset-nums tt)
    (setf (test-phase tt) 'p2b)
    (set-test-set tt))
   ((eq (test-phase tt) 'p2b)
    (record-and-reset-nums tt)
    (setf (test-phase tt) 'p2c)
    (set-test-set tt))
   ((eq (test-phase tt) 'p2c)
    (record-and-reset-nums tt)
    (setf (test-phase tt) 'p3 (criteria tt) 2)
    (set-test-set tt))
   ((eq (test-phase tt) 'p3)
    (record-and-reset-nums tt)
    (if (binary-done tt)
(setf (test-phase tt) 't2 (criteria tt) 3)
      (setf (test-phase tt) 't1 (criteria tt) 3))
    (set-test-set tt))
   ((eq (test-phase tt) 't1)
    (when (> (num-trials tt) 59)
      (record-and-reset-nums tt) ; crit->2 so get right test
      (setf (test-phase tt) 't2a (binary-done tt) t (criteria tt) 2))
    (set-test-set tt))
   ((eq (test-phase tt) 't2a)
    (when (> (num-trials tt) 31)
      (record-nums tt)
      (if (> 0.9 (/ (num-correct tt) (num-trials tt)))
  (setf (test-phase tt) 'p3 (criteria tt) 2)
(setf (test-phase tt) 't2 (criteria tt) 3))
      (reset-nums tt))
    (set-test-set tt))
   ((eq (test-phase tt) 't2)
    (when (> (num-trials tt) 59)
      (record-and-reset-nums tt)
      (setf (test-phase tt) 't3))
    (set-test-set tt))
   ((eq (test-phase tt) 't3)
    (when (> (num-trials tt) 299)
      (record-and-reset-nums tt)
      (setf (test-over tt) t (criteria tt) 'done))
    (set-test-set tt))
   )) ; defmethod test-phaser

(defmethod set-test-set ((tt test-giver))
  (setf (test-set tt)
(cond
 ((eq (test-phase tt) 'p2a)
  (mapcan #'(lambda (x) (list x x x x)) (cdr (desired tt))))
 ((eq (test-phase tt) 'p2b)
  (mapcan #'(lambda (x) (list x x)) (cdr (desired tt))))
 ((eq (test-phase tt) 'p2c)
  (copy-list (cdr (desired tt))))
 ((eq (test-phase tt) 'p3)
  (random-copy (cdr (desired tt))))
 ((eq (test-phase tt) 't1)
  (random-copy *possible-binaries*))
 ((eq (test-phase tt) 't2a)
  (random-copy (cdr (desired tt))))
 ((or (eq (test-phase tt) 't2) (eq (test-phase tt) 't3))
  (random-copy *possible-trinaries*))
 ))
  ) ; defmethod set-test-set

; OK, so now I wish I'd been using arrays ---
```

```
(defun random-copy (oldlist)
  (do ((midlist (copy-list oldlist) (delete (car newlist) midlist :count 1))
       (len (length oldlist) (- len 1))
       (newlist nil))
      ((null midlist) newlist)
    (setf newlist (cons (nth (random len) midlist) newlist))
    ))


; this also records numbers...
(defmethod record-and-reset-nums ((tt test-giver) &optional (trials))
  (record-nums tt) (reset-nums tt))

(defmethod record-nums ((tt test-giver) &optional (trials))
  (format (if *monkey-results-file* *monkey-results-file* t)
  "~%*** Phase ~s Complete: ~d done in ~d trials ***~%"
    (test-phase tt) (num-correct tt) (num-trials tt))
  (force-output *monkey-results-file*))

(defmethod reset-nums ((tt test-giver) &optional (trials))
  (setf (num-correct tt) 0 (num-trials tt) 0))

(defmethod 9-of-10? ((tt test-giver))
  (cond
   ((null (last-two-bad tt)) (< 8 (num-trials tt)))
   ((null (cdr (last-two-bad tt))) (< 9 (num-trials tt)))
   (t  (< 9 (- (num-trials tt) (cadr (last-two-bad tt)))))
  ))

(defmethod set-test ((tt test-giver))
  (setf *test-board* (copy-list (car (test-set tt)))))

(defmethod set-pair ((tt test-giver))
  (let ((item (car (test-set tt))))
    (setf *test-board*
  (copy-list
   (cond
    ((eq item (second (desired tt)))
     (list (first (desired tt)) item))
    ((eq item (third (desired tt)))
     (list (second (desired tt)) item))
    ((eq item (fourth (desired tt)))
     (list (third (desired tt)) item))
    ((eq item (fifth (desired tt)))
     (list (fourth (desired tt)) item))
    (t (error "set-pair: ~s not in cdr of ~s" item (cdr (desired tt))))
    ))))) ; defmethod set-pair

;;;;;;;;;;

(reset-color-trick) ; when you re-read library, want this set
(reset-color-rule-trick) ; when you re-read library, want this set
```

Here is the rest of the code. This file is for the ‘simulation’, but again, it wasn’t trivial to separate the ‘world’ from the monkey and its apparatus.

```
;; monkey-world.lisp -- set up things for the simulation

(defvar *colors* '(red white blue green yellow))
(defvar *test-board* '())
(defvar *hand* nil)
(defvar *floor* nil)     ;; where things you drop go
(defvar *attention* nil) ;; what you're thinking about / looking at
(defvar *monkey-results-file* nil)
(defvar *training-set* nil)
(defvar *pending-tests* nil)
(defvar *right* nil)


(defun init-world ()
  (reset)
  (reset-color-trick)
  (reset-monkey-counting)
  (set-pending-tests)
  (setf *hand* nil)
  (setf *test-board* nil)
  (setf *training-set* nil)
  (if *monkey-results-file* (close *monkey-results-file*))
  (setf *monkey-results-file*
(open "monkey-results" :direction :output
      :if-exists :overwrite :if-does-not-exist :create))
  )

;er... only for rule-trick results!
(defun cleanup-time ()
  (save-monkey-results *color-rule-trick*)
  (force-output *monkey-results-file*)
  (if *monkey-results-file* (close *monkey-results-file*))
  t)
```

```
(defun debug-world (stream time)
  (format stream "~%At time ~d:" time)
  (format stream "~%   Board is ~s" *test-board*)
  (format stream "~%    hand is ~s~%" *hand*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun two-of (somestuff)
  (let* ((lenss (list-length somestuff))
 (thing2 (random lenss))
 (thing1 (random lenss)))
    (if (= thing2 thing1) (setf thing1 (- thing1 1)))
    (if (< thing1 0) (setf thing1 (- lenss 1)))
    (list (nth thing1 somestuff) (nth thing2 somestuff))))

; in McGonigle's tests, this is just two of one of the things...
(defun three-of (somestuff)
  (let ((twothings (two-of somestuff)))
    (cons (car twothings) twothings)))
```

Appendix C

The Primate-Society Behavior Library

C.1 Interface

As I say in Chapter 8, the primitive interface file is the main documentation of what's going on in a library. Here's the interface for the primate society:

```
;;; A basic act --- for things that are never meant to succeed

(add-act 'fail
  (lambda () nil))

;;; motion / "body" prims from colony.lisp

#|
Because this is a MAS, we need to make sure we're using the right
agent's behaviors.  "my" is a macro for looking up the objects
associated with this process.
|#

(add-sense 'alligned-w-target
  (lambda () (let ((self (my 'primate)))
       (and (touching self (target self))
    (alligned self (target self))))))

(add-sense 'touching-target
  (lambda () (let ((self (my 'primate)))
       (touching self (target self)))))

(add-sense 'target-chosen
  (lambda () (target (my 'primate))))
```

```
; this one should probably be a competence
(add-act 'engage
  (lambda ()
    (let ((self (my 'primate)))
      (if (typep (target self) 'primate) (line-up self (target self))
nil) ;; FIXME why does this happen???
      )))

; this one too
(add-act 'approach
  (lambda () (let ((self (my 'primate)))
       (approach self (target self)))))

(add-act 'lose-target
  (lambda () (setf (target (my 'primate)) nil) t))

(add-act 'choose-isolated-place
  (lambda () (let ((self (my 'primate)))
       (setf (target self) (choose-isolated-place self)))))

; this should move into affiliative behavior, but for now is simple
(add-act 'choose-grooming-partner
  (lambda () (let ((self (my 'primate)))
       (setf (target self) (choose-grooming-partner self)))
    ))

(add-sense 'partner-chosen
  (lambda () (let ((partner (target (my 'primate))))
       (and partner (typep partner 'primate)))))

(add-sense 'partner-overlap
  (lambda () (let ((self (my 'primate)))
       (objects-overlap self (target self)))))

(add-act 'go-to-partner-edge
  (lambda () (let ((self (my 'primate)))
       (move-towards-edge self (target self)))
    ))

(add-sense 'partner-questionable ; *
  (lambda () (trim-unavailable-4grooming (target (my 'primate)))))

(add-act 'choose-available-grooming-partner ; *
  (lambda () (let ((self (my 'primate)))
       (setf (target self) (choose-available-grooming-partner self)))
    ))


;;; novelty prims, currently also in primate-novelty.lisp

(add-sense 'bored
  (lambda () (bored (my 'novelty))))  ; this don't work unless you add
                                      ; boredom back into wait
(add-act 'wait
  (lambda ()
    (do-it (my 'primate) 'wait)))     ; used to make more bored


;;; exploration prims, replace novelty, are also in primate-novelty since
;;; very similar

(add-sense 'want-new-loc
  (lambda () (want-new-location (my 'explore))))

(add-act 'leave
  (lambda ()
    (let ((x (my 'explore)))
      (approach (pm x) (target (pm x)))
      (increment (drive x))
      )))




;;; grooming prims from primate-groom.lisp

(add-sense 'want-to-groom
  (lambda () (want-grooming (my 'grooming))))

(add-act 'groom
  (lambda ()
    (let ((g (my 'grooming)))
      (if (typep (target (pm g)) 'primate) (groom g)
nil) ;; FIXME why does this happen???
      )))

;;; affinity should be a behavior --- include kin list
```

C.2 Scripts

Here are the script files, roughly in the order they were written.

```
(
(documentation "wandergroom.lap" "(first primate script)" "
$Log: wandergroom.lap,v $
Revision 3.0  2001/02/01 19:21:55  joanna
primates finally works!

Revision 2.5  2001/01/04 19:02:18  joanna
virtually working version

Revision 2.4  2000/12/07 22:20:59  joanna
Primate libraries load, testing real-time posh & wandergroom before testing
MAS

Revision 1.2  2000/11/26 01:21:08  joanna
making sure logging works

Revision 1.1  2000/11/26 01:19:06  joanna
Initial revision
")
(RDC
 LIFE
 (GOAL (fail))
 (DRIVES
  ((grooming (TRIGGER ((want-to-groom))) groom-comp))
  ((wandering (TRIGGER ()) wander-comp))))
(C
 groom-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((groom-gp (TRIGGER ((partner-chosen) (touching-target) (alligned-w-target
           ((allign-w-gp (TRIGGER ((partner-chosen) (touching-target))) engage))
           ((touch-gp (TRIGGER ((partner-chosen))) approach))
           ((choose-gp (TRIGGER ()) choose-grooming-partner))))
(C
 wander-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((bored-wander (TRIGGER ((bored))) lose-target))
   ((sit (TRIGGER ((target-chosen) (touching-target))) wait))
   ((wander-to (TRIGGER ((target-chosen))) approach))
           ((choose-wander (TRIGGER ()) choose-isolated-place))))
```

```
)

(
(documentation "zero-move-n-groom.lap" "from move-n-groom.lap -- original version wit
$Log: zero-move-n-groom.lap,v $
Revision 1.1  2001/02/16 16:19:07  joanna
Initial revision

Revision 1.1  2001/02/11 01:06:12  joanna
Initial revision

")
(RDC
 LIFE
 (GOAL (fail))
 (DRIVES
  ((grooming (TRIGGER ((want-to-groom))) groom-comp))
  ((exploring (TRIGGER ()) explore-comp))))
(C
 groom-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((groom-gp (TRIGGER ((partner-chosen) (touching-target) (alligned-w-target
           ((allign-w-gp (TRIGGER ((partner-chosen) (touching-target))) engage))
           ((touch-gp (TRIGGER ((partner-chosen))) approach))
           ((choose-gp (TRIGGER ()) choose-grooming-partner))))
(C
 explore-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((get-moving (TRIGGER ((want-new-loc))) lose-target))
   ((sit (TRIGGER ((target-chosen) (touching-target))) wait))
   ((move-away (TRIGGER ((target-chosen))) leave))
           ((choose-wander (TRIGGER ()) choose-isolated-place))))
)

(
(documentation "move-n-groom.lap" "from wandergroom.lap -- but do more moving! fixed
$Log: move-n-groom.lap,v $
Revision 1.2  2001/02/16 16:19:07  joanna
primate works except when on top of what you want to groom.

Revision 1.1  2001/02/11 01:06:12  joanna
Initial revision
```

```
")
(RDC
 LIFE
 (GOAL (fail))
 (DRIVES
  ((grooming (TRIGGER ((want-to-groom))) groom-comp))
  ((exploring (TRIGGER ()) explore-comp))))
(C
 groom-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((groom-gp (TRIGGER ((partner-chosen) (alligned-w-target))) groom))
   ((not-vert-gp (TRIGGER ((partner-overlap))) go-to-partner-edge))
           ((allign-w-gp (TRIGGER ((partner-chosen) (touching-target))) engage))
           ((touch-gp (TRIGGER ((partner-chosen))) approach))
           ((choose-gp (TRIGGER ()) choose-grooming-partner))))
(C
 explore-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((get-moving (TRIGGER ((want-new-loc)(target-chosen)(touching-target))) lo
   ((move-away (TRIGGER ((want-new-loc)(target-chosen))) leave))
           ((choose-wander (TRIGGER (want-new-loc)) choose-isolated-place))
   ((sit (TRIGGER ()) wait))
   ))
)
(
(documentation "just-wait.lap" "from move-n-groom -- but for debugging" "
$Log: just-wait.lap,v $
Revision 1.1  2001/02/16 16:19:07  joanna
Initial revision

Revision 1.1  2001/02/11 01:06:12  joanna
Initial revision

")
(RDC
 LIFE
 (GOAL (fail))
 (DRIVES
  ((exploring (TRIGGER ()) wait))))

)
(
(documentation "quota-groom.lap" "from move-n-groom -- go for grooming slots, no more
$Log: move-n-groom.lap,v $
Revision 1.2  2001/02/16 16:19:07  joanna
primate works except when on top of what you want to groom.

Revision 1.1  2001/02/11 01:06:12  joanna
Initial revision

")
(RDC
 LIFE
 (GOAL (fail))
 (DRIVES
  ((grooming (TRIGGER ((want-to-groom))) groom-comp))
  ((exploring (TRIGGER ()) explore-comp))))
(C
 groom-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((groom-gp (TRIGGER ((partner-chosen) (touching-target) (alligned-w-target
   ((double-check (TRIGGER ((partner-chosen) (partner-questionable))) reconsider-part
           ((allign-w-gp (TRIGGER ((partner-chosen) (touching-target))) engage))
           ((touch-gp (TRIGGER ((partner-chosen))) approach))
           ((choose-gp (TRIGGER ()) choose-grooming-partner))))
(AP reconsider-partner (minutes 1) (wait wait choose-available-grooming-partner))
(C
 explore-comp
 (MINUTES 10)
 (GOAL (fail))
 (ELEMENTS ((get-moving (TRIGGER ((want-new-loc)(target-chosen)(touching-target))) lo
   ((move-away (TRIGGER ((want-new-loc)(target-chosen))) leave))
           ((choose-wander (TRIGGER (want-new-loc)) choose-isolated-place))
   ((sit (TRIGGER ()) wait))
   ))
)
```

C.3 Behavior Library

For the primates, there really is a simulation — there needs to be a 2-D environment that they run around in. Again, it is difficult to divide some of the aspects of behavior primate from the world it lives in, so that behavior is defined in the same source file as the world, at least for now.

```
;; colony.lisp --- for supporting free-ranging primate behavior JB Nov 2000
```

```
#|

This file supports the physical problem of moving around the world. As
such, it's a bit odd as a behavior --- it includes besides the more
trad behavior 'primate:
  o the GUI world interface 'primate-world
  o the list of agents & objects in the world --- all the observables

One might want to say it contains more than one behavior, but they are
all in one file because they are heavily interrelated.

NB!!  When you add new behaviors, add them to method initialize below!


How to recognize what beasties are doing:

   CH    Charlie (all names should be unique in first 2 letters)
   CH-   Charlie gesturing right
  =CH    Charlie touching left
  ^CH    Charlie gesturing above
  vCHv   Charlie touching below
   CH%   Charlie in heat (females only.)

Grey          - neutral: normal motion or sitting
Pink          - displaying
Red           - angry (fighting)
Orange        - frightened (screeching)
Lavender      - aroused
Purple        - mounting
Blue          - playing
Green         - grooming
Turquoise     - eating (food is yellow too -- has digit for # of elements)

max size on my screen is 950x1250

|#

(defvar *screen-height* 500)
(defvar *screen-width* 500)

; the code is now in primate-interface.lisp

  ;;;                         ;;;
  ;;   Creation of the World   ;;
  ;;;                          ;;;

; This is now done via another interface, the launcher, also in
; primate interface

(defvar *world* nil "The screen / arena where the primates live.")

(defvar *primate-launcher* nil "How to build primates.")
(setf *primate-launcher* (make-instance 'primate-launcher))
(display *primate-launcher*)


;;; A variable affecting motion ---
;;     See posh-utils.lisp for *time-increment* stuff

(defvar *move-increment* 4
  "how far (in pixels) you normally move in *time-increment*")


  ;;;                    ;;;
  ;;   Primate Behavior   ;;
  ;;;                    ;;;


; Actually, primate is a subclass of object which is a subclass of target

; default target is really just a point.
; Sometimes you want to go to points.
(defclass target ()
  (
   (x :accessor x :initarg :x :initform 0)
   (y :accessor y :initarg :y :initform 0)
   (width :accessor width :initarg :width :initform 0)
   (height :accessor height :initarg :height :initform 0)
   )
  )

; typically, a is self and b is other.  Both should inherit from point
(defmacro distance (a b)
  `(let ((diffy (- (y ,b) (y ,a)))
 (diffx (- (x ,b) (x ,a))))
     (sqrt (+ (* diffy diffy) (* diffx diffx)))))

; comment as for distance
(defmacro direction (a b)
  `(let ((diffy (- (y ,b) (y ,a)))
 (diffx (- (x ,b) (x ,a))))
     (atan diffy diffx)))

(defmacro get-relative-point (a dist theta)
  `(let ((delta-y (round (* ,dist (sin ,theta))))
 (delta-x (round (* ,dist (cos ,theta)))))
     (make-instance 'target :x (+ (x ,a) delta-x) :y (+ (y ,a) delta-y))
     ))

; the only other object I anticipate is food, but who knows?
(defclass object (target)
  (
   (height :accessor height :initarg :height :initform 1)
   (width :accessor width :initarg :width :initform 1)
   (color :accessor color :initarg :color :initform ':lightgrey)
   (pinup :accessor pinup :initarg :pinup :initform nil)
   )
  )

; (item-text (pinup pm)) <- how to fix label

(defclass primate (object)
  (
   (name :accessor name :initarg :name :initform "name")
   (logo :accessor logo :initarg :logo :initform 'logo)
   (label :accessor label :initarg :label :initform " ~s ")
   (doing :accessor doing :initarg :doing :initform nil)
   (gender :accessor gender :initarg :gender :initform :f)
   ; the focus of attention
   (target :accessor target :initarg :target :initform nil)
   ))

;;                               ;;
;;;  "Special" Primate Methods  ;;;
;;                               ;;
;;; these do more system-level than agent-level work



;this returns a symbol, honest... (and also a length)
; "symb" is for special gender stuff or state maybe
(defmacro logo-from-name (name &optional (symb #\Space))
 `(read-from-string (make-array 3 :element-type 'base-char :initial-contents
(list (aref ,name 0) (aref ,name 1) ,symb))))
```

```
#|

I can't get special individual process state to work the way it's
supposed to (according to Bovy), so I had to axe this bit...

;faster version.  Where is *my-mind*?  It should be set in primate
;method "initialize", and it should be process-specific.
(defmacro my (what)
  `(gethash ,what *my-mind*))

|#

; This is a global reference for the actual bodies -- it's the only
; hash referenced in this behavior, so it might be considered part of
; the "state" of the perception system
(defvar *primates* (make-hash-table)
  "All active primates, hashed by process ID.")

; these are because I can't get *my-mind* (beh's local to procs) to work
; they are part of their own respective behaviors I suppose.
(defvar *grooming* (make-hash-table)
  "The grooming behaviors of all active primates, hashed by process ID.")
(defvar *novelty* (make-hash-table)
  "The novelty behaviors of all active primates, hashed by process ID.")
(defvar *explore* (make-hash-table)
  "The new novelty behaviors -- old *novelty* just kept as an example.")

(defun reset-primates ()
  (loop for xxx being each hash-key of *primates*
do (mp:process-kill xxx))
  (clrhash *primates*)
  (clrhash *grooming*)
  (clrhash *novelty*)
  (clrhash *explore*)
  )

(defmethod initialize ((pm primate))
  ; characterize yourself
  (setf (logo pm) (logo-from-name (name pm)))
  (cond
   ((eq (gender pm) :f)
     (setf (height pm) 29) (setf (width pm) 62))
   )
  ; show yourself
  (be-born pm)
  ; maintain the global list of primates
  (setf (gethash mp:*current-process* *primates*) pm)
  ; (defvar *my-mind* (make-hash-table)) -- if only!  No local minds...
  (setf (gethash mp:*current-process* *grooming*)
(make-instance 'groom-beh :pm pm))
  (setf (gethash mp:*current-process* *novelty*)
(make-instance 'novelty-beh :pm pm))
  (setf (gethash mp:*current-process* *explore*)
(make-instance 'explore-beh :pm pm))
  )

;here's how the agents access their minds
(defmacro my (what)
  `(let ((process mp:*current-process*))
     (cond
      ((eq ,what 'primate) (gethash process *primates*))
      ((eq ,what 'grooming) (gethash process *grooming*))
      ((eq ,what 'novelty) (gethash process *novelty*))
      ((eq ,what 'explore) (gethash process *explore*))
      (t
       (error "my:  my what? (~s)" ,what))
      )))

;; Seeing Jane Goodall speak at MIT was one of the few experiences
;; I've had as an adult that completely recalled the level of
;; admiration I felt for someone as a child.  This does logging;
;; it's called by "do-it"

(defvar *janefile* nil "this gets opened and closed in primate-interface.lisp")
(defmethod jane ((pm primate) what)
;;  (when (eq (logo pm) 'jo) ; just for now
    (format *janefile* "~@20a ~@10s ~20a ~d~%"
    (name pm) what
    (if (target pm) (if (typep (target pm) 'primate)
(name (target pm))
      (list (x (target pm)) (y (target pm))))
      nil)
    (get-internal-real-time))
    (force-output *janefile*)
;;    )
    )

; this changes what the primate looks like it's doing
; if no GUI, then def this to (defmacro make-look (pm) )
```

```
(defmacro make-look (pm)
  `(setf (capi::simple-pane-background (pinup ,pm))
 (color ,pm)
 (capi::item-text (pinup ,pm))
 (format nil (label ,pm) (logo ,pm))))

; this is more by mood than "what", maybe need another variable
; most of the time, neutral color
(defmacro get-color (what)
  `(cond
    ((eq ,what 'display) :pink)
    ((or ; angry
      (eq ,what 'fight)
      (eq ,what 'chase)
      ) :red)
    ((or ; scared
      (eq ,what 'flee)
      (eq ,what 'screech)
      ) :orange)
    ((eq ,what 'exhibit) :lavender)
    ((eq ,what 'mount) :purple)
    ((eq ,what 'play) :blue)
    ((eq ,what 'groom) :green)
    ((eq ,what 'eat) :yellow)
    (t :lightgrey)))

; so far, only "groom" is implemented, but the others are to give a taste
(defmacro get-pose (what)
  `(cond
    ((or
      (eq ,what 'groom)
      (eq ,what 'mount)
      (eq ,what 'fight)
      ) 'touch)
    ((or
      (eq ,what 'beg)
      (eq ,what 'indicate)
      ) 'gesture)
    (t 'neutral)))

(defmacro bottom-of (o)
  `(+ (y ,o) (height ,o)))
(defmacro top-of (o)
  `(y ,o))
(defmacro right-of (o)
  `(+ (x ,o) (width ,o)))
(defmacro left-of (o)
  `(x ,o))

(defconstant *dirarray* (make-array 5 :initial-contents
    '(left up right down left))
  "find directions from theta (see get-pose-direction)")

; lisp trig is all in polar notation
(defmacro get-pose-direction (pm)
  `(let ((dir (direction ,pm (target ,pm))))
    (aref *dirarray*
  (round (* 2 (/ (+ dir pi) pi))))))

; note sex signs (%, !) should be part of the logo, not of this.
(defmethod get-label ((pm primate) what)
  (let* ((pose (get-pose what))
 (direction (if (eq pose 'neutral) nil (get-pose-direction pm)))
 )
    (cond
     ((eq pose 'gesture)
      (cond
       ((eq direction 'up) '"^~s ")
       ((eq direction 'left) '"-~s ")
       ((eq direction 'right) '" ~s-")
       ((eq direction 'down) '" ~sv")
       ))
     ((eq pose 'touch)
      (cond
       ((eq direction 'up) '"^~s^")
       ((eq direction 'left) '"=~s ")
       ((eq direction 'right) '" ~s=")
       ((eq direction 'down) '"v~sv")
       ))
     ((eq pose 'neutral) '" ~s ")
     )))

(defmethod do-it ((pm primate) what)
  (unless (eq what (doing pm))
    (setf (doing pm) what)
    (setf (label pm) (get-label pm what))
    (setf (color pm) (get-color what))
    (make-look pm)
    (jane pm what) ; jane should be here
    )
```

```
;   (jane pm what) ; she's out here for emergency debugging
  (setf (capi:pinboard-pane-position (pinup pm)) (values (x pm) (y pm)))
  (sleep *time-increment*)
  t ; (sleep) returns "nil" for some reason
  ) ; defmethod do-it

;;; Primate Behavior Methods ;;;

; notice this allows for changing color, format and position
(defmethod be-born ((pm primate))
  (if (pinup pm)
      (progn
(setf (element-parent (pinup pm)) nil)
(destroy (pinup pm))))
  (setf (pinup pm)
(make-instance 'capi:push-button :x (x pm) :y (y pm)
       :text (format nil (label pm) (logo pm))
                       :font (gp:make-font-description
                        :family "courier" :size 15 :weight :bold
                        )
       :background (color pm)))
#| this should work but doesn't...
  (setf (pinup pm) ; hack to make font work on windows boxes too
(make-instance 'capi:push-button :x (x pm) :y (y pm)
       :text (format nil (label pm) (logo pm))
       :font (gp:font-description
                              (gp:find-best-font (pinup pm)
                                (gp:make-font-description
                                 :family "courier" :size 15 :weight :bold
                                 )))
       :background (color pm)))
|#
  (setf (element-parent (pinup pm)) (pane-layout *world*))
  ) ; be-born primate

(defmethod die ((pm primate))
  (if (pinup pm)
      (progn
(setf (element-parent (pinup pm)) nil)
(destroy (pinup pm))))
  )

(defmethod go-up ((pm primate))
  (let ((newy (- (y pm) *move-increment*)))
    (if (< newy 0) nil
      (progn
(setf (y pm) newy)
(setf (capi:pinboard-pane-position (pinup pm)) (values (x pm) (y pm)))
))))

(defmethod move-up ((pm primate))
  (loop for ix from 1 to 60
    do (go-up pm) (sleep .1)))

; give the minimum quant in the direction of the first arg
; (absmin -1 5) => -1;  (absmin -5 2) => -2  (absmin 5 2) => 2
(defmacro absmin (posorneg pos)
  `(if (< (abs ,posorneg) ,pos)
      ,posorneg
    (if (< 0 ,posorneg) ,pos (- 0 ,pos))))

(defmethod go-toward-point ((pm primate) x y)
  (let ((diffy (- y (y pm)))
(diffx (- x (x pm)))
(delta-x 0) (delta-y 0) (dist 0))
    (setf dist (round (sqrt (+ (* diffy diffy) (* diffx diffx)))))
    (setf dist (min *move-increment* dist))

    (cond
     ((= diffx 0)
      (setf delta-y (if (< diffy 0) (- dist) dist)))
     ((= diffy 0)
      (setf delta-x (if (< diffx 0) (- dist) dist)))
     (t (let ((theta (atan diffy diffx))
  (setf delta-y (round (* dist (sin theta))))
  (setf delta-x (round (* dist (cos theta))))))
     )
    (setf (x pm) (+ (x pm) delta-x) (y pm) (+ (y pm) delta-y))
    ))

(defmethod touching ((pm primate) target)
  (and (if (< (x pm) (x target))
   (< (- (x target) (x pm)) (+ (width pm) 2))
        (< (- (x pm) (x target)) (+ (width target) 2)))
       (if (< (y pm) (y target))
   (< (- (y target) (y pm)) (+ (height pm) 2))
        (< (- (y pm) (y target)) (+ (height target) 2)))
   ))

; FIXME for diff sizes! (this may be close enough for grooming and...)
```

```
(defmethod aligned ((pm primate) target)
  (cond
   ((typep target 'primate)
    (let ((ydiff (abs (- (y pm) (y target))))
  (xdiff (abs (- (x pm) (x target)))))
      (or
       (and (<= ydiff 2) (<= (- (width pm) xdiff) 2))
       (and (<= xdiff 2) (<= (- (height pm) ydiff) 2)))))
   (t (error "Target ~s of unknown type." target))
   ))

; pm and target better have methods x, y and height!
(defmacro overlapping-in-y (pm target)
  `(or
    (and (> (+ (y ,pm) (height ,pm)) (y ,target)) ; bottom pm lower than
 (< (y ,pm) (y ,target)))                     ; and top pm above top t
    (and (> (+ (y ,pm) (height ,pm)) (+ (y ,target) (height ,pm)))
 (< (y ,pm) (+ (y ,target) (height ,pm)))); or overlapping bottom t
    ))

; pm and target better have methods x, y and height!  See
; overlapping-in-y comments, I can't begin to work this out, I just
; did subs.
(defmacro overlapping-in-x (pm target)
  `(or
    (and (> (+ (x ,pm) (width ,pm)) (x ,target))
 (< (x ,pm) (x ,target)))
    (and (> (+ (x ,pm) (width ,pm)) (+ (x ,target) (width ,pm)))
 (< (x ,pm) (+ (x ,target) (width ,pm))))
    ))

(defmethod objects-overlap ((o1 target) (o2 target))
  (and (overlapping-in-x o1 o2) (overlapping-in-y o1 o2)))

; only do this if already "touching"!!
(defmethod align ((pm primate) target axis)
  (cond
   ((typep target 'primate)
    (if (eq axis :y)
(go-toward-point pm (x pm)(y target))  ;      move in y
    (go-toward-point pm (x target) (y pm))))   ; else move in x
   (t (error "Target ~s of unknown type for aligning." target))
   ))

; for getting out of under/over another monkey... for now, since I'm
; busy, we assume up-down is the way to go (generally shorter so more likely)
(defmethod move-towards-edge ((pm primate) (target target))
  (if (or (and (< (y pm) (y target)) ; top pm  is above top target
       (< 0 (- (y target) (height pm)))) ; not near top wall
  (> *screen-height* (+ (y target) (height target) (height pm))))
      (go-toward-point pm (x pm)(- (y target) (height pm)))   ;    move over it
    (go-toward-point pm (x pm)(+ (y target)(height target)))) ; else move under
  (do-it pm 'unstack)
  )

; this timed version isn't needed when you are letting the POSH arch
; handle the timing, but kept for simplicity until proven bad
(defmethod chase ((pm primate) other
  &optional (time-increment *time-increment*))
  (do ()
      ((touching pm other) t)
    (go-toward-point pm (x other) (y other))
    (setf (capi:pinboard-pane-position (pinup pm)) (values (x pm) (y pm)))
    (sleep time-increment)
    )
  )

; this timed version isn't needed when you are letting the POSH arch
; handle the timing, but kept for simplicity until proven bad
(defmethod approach ((pm primate) other
  &optional (time-increment *time-increment*))
    (go-toward-point pm (x other) (y other))
    (do-it pm 'approach)
  )

; note this bails either for success or for losing a pre-req --
; touching.  Also see comment above on POSH.
(defmethod line-up ((pm primate) other
  &optional (time-increment *time-increment*))
  (let ((axis (if (overlapping-in-y pm other) :y :x)))
    (align pm other axis)
    (do-it pm 'align)
    )
  )

;;; methods involving other agents ;;;

;;; NOTE: navigation should never choose to go past the world's boundaries!

;; but this is a really dumb fix --- pays no attention to other obstacles.
;; changes direction of motion as well as extent.
(defmacro fix-target-boundaries (pm)
  `(progn
     (if (< (x (target ,pm)) 0) (setf (x (target ,pm)) 0))
     (if (< (y (target ,pm)) 0) (setf (y (target ,pm)) 0))
     (if (> (x (target ,pm)) (- *screen-width* (width ,pm)))
 (setf (x (target ,pm)) (- *screen-width* (width ,pm))))
     (if (> (y (target ,pm)) (- *screen-height* (height ,pm)))
 (setf (y (target ,pm)) (- *screen-height* (height ,pm))))
     t
     ))

; also dumb.
; dist's default should prob. be based on the size of the screen...
; this will happily go *past* the center, I just want mingling.
(defmethod go-towards-center ((pm primate) &optional (dist 150))
  (let ((center (make-instance 'target :x (/ *screen-width* 2)
     :y (/ *screen-height* 2))))
    (setf (target pm) (get-relative-point pm dist (direction pm center)))
    ))


(defclass other ()
  (
   (agent :accessor agent :initarg :agent :initform nil)
   (dist :accessor dist :initarg :dist :initform -1)
   (theta :accessor theta :initarg :theta :initform -20)
   )
  )

; this is a questionable algorithm -- should select from nearby quadrants
; or something, so that walls could be more easily taken into account.
; But this may be more bio-looking.
(defmethod choose-isolated-place ((pm primate) &optional (dist 75))
  (let ((others (distance-truncate-others
 (observe-others-and-walls pm) (* 2 dist))))
    (if others
(setf (target pm)
      (get-relative-point pm dist (find-biggest-gap others)))
      (go-towards-center pm))
    (fix-target-boundaries pm)
    (target pm)
    ))

(defmethod observe-others ((pm primate))
  (loop for x being each hash-value of *primates*
unless (eq x pm)  ;; I am not an other!
  collect (make-instance 'other :agent x :dist (distance pm x)
 :theta (direction pm x))))

; treat nearest walls as others too
(defmethod observe-others-and-walls ((pm primate))
  (let ((xwall (make-instance 'target :x (if (< (x pm) (/ *screen-width* 2))
     -1 (+ 1 *screen-width*))
      :y (y pm)))
(ywall (make-instance 'target :y (if (< (y pm) (/ *screen-height* 2))
     -1 (+ 1 *screen-height*))
      :x (x pm))))
    (let ((near-corner (make-instance 'target :x (x xwall)
              :y (y ywall))))
      (cons (make-instance 'other :agent xwall :dist (distance pm xwall)
   :theta (direction pm xwall))
    (cons (make-instance 'other :agent ywall :dist (distance pm ywall)
 :theta (direction pm ywall))
  (cons (make-instance 'other :agent near-corner
       :dist (distance pm near-corner)
       :theta (direction pm near-corner))
(observe-others pm))))
    )))


(defun sort-others (others how)
  (cond
   ((or (eq how 'distance) (eq how 'dist))
    (sort others #'(lambda (x y) (< (dist x) (dist y)))))
   ((or (eq how 'direction) (eq how 'theta))
    (sort others #'(lambda (x y) (< (theta x) (theta y)))))
   ))

(defun distance-truncate-others (others where)
  (loop for x in others
unless (> (dist x) where) collect x)
  )

; this returns the direction of the center of the biggest gap
; others should be a list.  formats are for debugging
(defun find-biggest-gap (others)
  (cond
   ((null others) (error "find-biggest-gap: others can't be null (but is)"))
   ((null (cdr others))
    (if (> (theta (car others)) pi)
(- (theta (car others)) pi) (+ (theta (car others)) pi)))
   (t
    (sort-others others 'direction)
    (let ((maxgap 0)
  (maxgapctr 0)
  (first-other (car others))
  (prev-other (car others)))
      (loop for x in (cdr others)
    do (let ((thisgap (abs (- (theta x) (theta prev-other)))))
 ; (format t "~%maxgap ~d; maxgapctr ~d; theta prev-other ~d; this theta ~d; this gap
 (if (> thisgap maxgap)
     (progn
       (setf maxgap thisgap)
       (setf maxgapctr (/ (+ (theta x) (theta prev-other)) 2))
       ))
 (setf prev-other x)
 )
    finally ; this one is tricky 'cause it wraps...
    (let ((thisgap (abs (- (- (theta prev-other) (* 2 pi))
   (theta first-other)))))
      ; (format t "~%maxgap ~d; maxgapctr ~d; theta prev-other ~d; this theta ~d; thi
      (if (> thisgap maxgap)
  (progn
    (setf maxgap thisgap)
    (setf maxgapctr
  (+ (theta prev-other) (/ thisgap 2))) ; wrapping OK here
    ))))
      maxgapctr))
   )) ; defun find-biggest-gap

;this will get more sophisticated (and change behavior modules)
(defmethod choose-grooming-partner ((pm primate) &optional (dist 150))
  (let ((others (observe-others pm)))
    (if  (car others)
      (progn
(sort-others others 'distance)
(setf (target pm) (agent (car others))))
      nil)
    ))


#|
;;;;;;;  Testing only
```

```
(setf j (make-instance 'primate :name "Joanna"))
(setf w (make-instance 'primate :name "Will" :x 300 :y 300))
(setf (height j) 29) (setf (width j) 62)
(setf (height w) 29) (setf (width w) 62)

(be-born j)
(be-born w)
|#
```

These are the emotion / chemical drive behaviors. Remember that driv-level is actually defined in a general posh file, shown in Section A.4.

```
; primate-groom --- how and when and (eventually) whom do you groom?

; Modeling assumptions: beasts essentially remember the last 5
; minutes.  They want to have been grooming about a minute of that.
; More than a minute and a half of that is annoying.

; interface:  want-to-groom, groom

; This should include a list of who's currently grooming me (objects +
; timestamps) so that one can "tolerate" grooming w/o doing it (color but no
; gesture).
(defclass groom-beh ()
  (
   (pm :accessor pm :initarg :pm :initform nil)
   ; this remembers how much I've been groomed recently
   (drive :accessor drive :initarg :drive
  ; comment out the below to get it back up to 5, but this
  ; fixes quicker
  :initform (make-instance 'drive-memory
               :memory-length 2 :recent-chunks '(0.2 0.2)
        :cc-start-time (get-internal-real-time)
        :engage-latch 0.1 :disengage-latch 0.25))
   ; these are in terms of the drive -- this is now *in* the drive
   ; (want-thresh :accessor want-thresh :initarg :want-thresh :initform 0.1)
   ; (target :accessor target :initarg :target :initform 0.2)
   ; (tolerate-thresh :accessor tolerate-thresh :initarg :tolerate-thresh :initform 0
   ; how long I groom of my own volition, in *time-increments*
   (groomed-threshold :accessor groomed-threshold :initarg :groomed-threshold
      :initform (* 10 *incs-per-second*))
   ))

; there are many reasons to want grooming, this function checks for
; any of them...  This will eventually include tolerance for being
; groomed, and social desires to groom.
; (defmethod want-grooming ((pg groom-beh))
;   (cond
;    ((> (want-thresh pg) (level (drive pg))) t) ; individual drive
;    ((and (engaged pg) (> (target pg) (level (drive pg)))) t)
;    (t nil)))
; only which version of "engaged" is determined by the behavior -- the
; actual state of the animal is determined from its body!
; (defmethod engaged ((pg groom-beh))
;   (eq (get-color 'groom) (color (pm pg))))


(defmethod want-grooming ((pg groom-beh))
  (latched (drive pg)))

; this should also send a message to target's groom behavior that it's
; being groomed.
(defmethod groom ((pg groom-beh) &optional (other nil))
  (when other (setf (target (pm pg)) other))
  (do-it (pm pg) 'groom)
  (increment (drive pg))
  )



; primate-novelty --- attention wandering and such like

; boredom resets if haven't been bored recently.  Note these may have
; to become hashes --- you may still be bored by one thing and not by
; another (yet).  Or maybe it's backwards --- really, notice when
; something is new, not when something is old.
; Jan 30 -- Boredom done as a drive (sort of an inverse one)
(defclass novelty-beh ()
  (
   (pm :accessor pm :initarg :pm :initform nil)
   (drive :accessor drive :initarg :drive :initform
  (make-instance 'drive-memory
 :memory-length 1 :recent-chunks '(0.0)
 :cc-start-time (get-internal-real-time)))
   (bored-threshold :accessor bored-threshold :initarg :bored-threshold
      :initform 0.8)
   ))

(defmethod bored ((pn novelty-beh))
  (< (bored-threshold pn) (level (drive pn))))

; start over from scratch if you've mostly been interested lately
(defmethod get-bored ((pn novelty-beh) increment)
  (increment (drive pn)))


;;;;;;;; look at it another way

; turn it around -- we want to see somewhere new if we haven't been
; moving lately!  Still very short memory, but the thresholds have reversed
(defclass explore-beh ()
  (
   (pm :accessor pm :initarg :pm :initform nil)
   (drive :accessor drive :initarg :drive :initform
  (make-instance 'drive-memory
 :memory-length 1 :recent-chunks '(0.0) :rc-total 0
 :cc-start-time (get-internal-real-time)
 :engage 0.25 :disengage 0.31))
   ))

(defmethod want-new-location ((pe explore-beh))
  (latched (drive pe)))
```

Finally, here's yet more GUI code.

```
; primate-interface.lisp --- January 2001

(defun start-primate-process (name filename)
  (mp:process-run-function
   (format nil "primate ~D" name)
   '()
   'create-primate name filename))

(defun create-primate (name filename)
  (let ((beastie (make-instance 'primate :name name)))
    (initialize beastie)
    (real-posh :plan-name filename :posh-output t :hush t)))

; First, here's the world.
(define-interface primate-world ()
  ()
  (:panes
   )
  (:layouts
   (world
    pinboard-layout
    '()
    :background :oldlace
    :best-height (+ *screen-height* 45) ; need room for menu bar!
    :best-width *screen-width*
   )
   (top
    column-layout
    '(world)
    )
   )
  (:default-initargs
   :best-height *screen-height*
   :best-width *screen-width*
   :destroy-callback 'end-of-the-world
   :layout 'world
   :title "Primate World"
   :background :darkSLATEBLUE))

; this is pretty useless.  I get some error when I try to do more.
(defun end-of-the-world (&rest unreferenced-args)
  (sleep 1)
  (when *world*
    (format *debug-world* "To show primate colony again: (display *world*)"))
  )

(defun end-of-primate-launcher (&rest unreferenced-args)
  (format *debug-world* "To restart launcher: (display *primate-launcher*)"))

(define-interface primate-launcher ()
  ()
  (:panes
   (new-world-pb
    push-button
    :text "New World"
    :selection-callback 'new-world
    :callback-type :interface
    :reader get-new-world-pb
    :enabled t
    :background :rosybrown1)
   (destroy-world-pb
    push-button
    :text "Destroy World"
```

```
 :selection-callback 'destroy-world
 :callback-type :interface
 :reader get-destroy-world-pb
 :enabled nil
 :background :rosybrown1)
(debugging-primate-pb
 push-button
 :text "Debugging Primate"
 :selection-callback 'debugging-primate
 :callback-type :interface
 :reader get-debugging-primate-pb
 :enabled nil
 :background :indianred1)
(new-primate-pb
 push-button
 :text "New Primate"
 :selection-callback 'new-primate
 :callback-type :interface
 :reader get-new-primate-pb
 :enabled nil
 :background :rosybrown1)
(choose-primate-fap-dp
 display-pane
 :title "New Primate's Script:"
 :title-position :left
 :text "none"
 :visible-border t
 :min-width 200
 :background :white
 :reader get-choose-primate-fap-dp)
(choose-primate-fap-pb
 push-button
 :text "Choose"
 :selection-callback 'choose-primate-lap
 :callback-type :interface
 :reader get-choose-primate-fap-pb
 :enabled t
 :background :rosybrown1)
(choose-primate-number-tip
 text-input-pane
 :title "Number of New Primates:"
 :title-position :left
 :callback-type :interface
 :callback 'new-primate
 :reader get-choose-primate-number-tip
```

```
     :enabled nil
     :text "1"
     :max-characters 4
     :max-width 45 :min-width 45
     :background :white
     :reader primate-number-tip)
    (time-inc-op  ; except this is wrong!  this isn't the time-inc...
     option-pane
     :items '(2sec 30sec 2min 10min)
     :selected-item 3
     :title "  Trail Length:"
     :selection-callback 'set-time-inc
     :callback-type :item
     :reader get-time-inc-op
     :background :white)
    )
   (:layouts
    (worlds
     row-layout
     '(new-world-pb destroy-world-pb)
     :visible-border t
     :background :darkSLATEBLUE
     )
    (world-plus
     row-layout
;     '(worlds time-inc-op)
     '(worlds)
     )
    (beasts
     row-layout
     '(new-primate-pb choose-primate-number-tip debugging-primate-pb)
     )
    (sixes
     row-layout
     '(choose-primate-fap-dp choose-primate-fap-pb)
     )
    (top
     column-layout
     '(world-plus beasts sixes)
     )
    )
   (:default-initargs
    :destroy-callback 'end-of-primate-launcher
    :layout 'top
    :title "Primate Launcher"
    :background :darkSLATEBLUE))

 (defun set-time-inc (choice)
   choice  ; this isn't really done yet
   ) ; defun set-time-inc

 (defun new-world (interface)
   (setf *janefile* (open (format nil "jane.~d" (get-universal-time))
  :direction :output
  :if-does-not-exist :create))
   (setf *world* (make-instance 'primate-world))
   (display *world*)
   (setf (capi:button-enabled (get-new-world-pb interface)) nil)
   (setf (capi:button-enabled (get-destroy-world-pb interface)) t)
   )

 (defun destroy-world (interface)
   (when  (confirm-yes-or-no "This will destroy the primates too.~%Proceed?")
     (close *janefile*)
     (quit-interface *world*)
     (setf *world* nil)
     (setf (capi:button-enabled (get-new-world-pb interface)) t)
     (setf (capi:button-enabled (get-destroy-world-pb interface)) nil)
     (reset-primates)
     ))

 (defun new-primate (interface)
   (let ((script (display-pane-text  (get-choose-primate-fap-dp interface)))
 (num (parse-integer (text-input-pane-text
      (get-choose-primate-number-tip interface)))))
     (when (confirm-yes-or-no "Create ~d more primate~:P with ~s?" num script)
       (loop for ixi from 1 to num
     do (launch-primate ixi script))
       )))

 (defun launch-primate (which script)
   (start-primate-process
    (prompt-for-string (format nil "Name for new primate #~D" which)) script))

 (defun choose-primate-lap (interface)
   (let ((file (prompt-for-file "Which Script File?" :filter "*.lap")))
     (partial-brain-flush) ;; FIX ME -- don't want this here! See TODO
     (unless (null file)
       (setf (display-pane-text  (get-choose-primate-fap-dp interface))
     (file-namestring file))
```

```
      (setf (capi:text-input-pane-enabled
     (get-choose-primate-number-tip interface)) t)
      (setf (capi:button-enabled (get-new-primate-pb interface)) t)
      (setf (capi:button-enabled (get-debugging-primate-pb interface)) t)
      )
  ))

(defun primate-number (data interface)
  (let ((num (parse-integer data :junk-allowed 't)))
    (if (or (null num) (< num 1) (> num 100))
(progn
  (display-message "Number of primates must be between 1 & 100")
  (setf (text-input-pane-text
 (get-choose-primate-number-tip interface)) "1"))
      (setf (text-input-pane-text
     (get-choose-primate-number-tip interface)) (format nil "~d" num))
      )))

(defvar *Buster* nil "make me when you need me so I have up-to-date code")

; much of this is from bod-interface.lisp: send-run.  Important stuff
; is done below in "init-world"
(defun debugging-primate (&optional unreferenced-interface)
  (setf *Buster* (make-instance 'primate :name "Buster de Bug"))
  (setf *posh-input-stream* (comm:open-tcp-stream "localhost" *posh-port*))
  )

; this will get run only from posh-from-gui, because regular primates
; don't bother passing init-world in!
(defun init-world ()
  (initialize *Buster*)
  (posh-script (display-pane-text
(get-choose-primate-fap-dp *primate-launcher*)))
  )

(defun make-target ()
  (let ((beastie (make-instance 'primate :name "It" :x 300)))
    (initialize beastie)))
```

Bibliography

Jack A. Adams. Learning of movement sequences. *Psychological Bulletin*, 96(1):3–28, 1984.

Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 196–201, Seattle, Washington, July 1987. Morgan Kaufmann.

Philip E. Agre and David Chapman. What are plans for? In Pattie Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 3–15. MIT Press, Cambridge, MA, 1990.

J. S. Albus. The NIST real-time control system (RCS): an approach to intelligent systems research. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):147–156, 1997.

James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung Hee Hwang, Tsuneaki Kato, Marc Light, Nathaniel Martin, Bradford Miller, Massimo Poesio, and David R. Traum. The TRAINS project: a case study in building a conversational planning agent. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):7–48, 1995.

J. R. Anderson. *Rules of the Mind*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1993.

J. R. Anderson and M. Matessa. The rational analysis of categorization and the ACT-R architecture. In M. Oaksford and N. Chater, editors, *Rational Models of Cognition*. Oxford University Press, 1998.

Elisabeth André, Thomas Rist, and Jochen Müller. Integrating reactive and scripted behaviors in a life-like presentation agent. In Katia P Sycara and Michael Wooldridge, editors, *Proceedings of the Second International Conference on Autonomous Agents*, pages 261–268. ACM Press, 1998.

Colin M. Angle. Genghis, a six legged walking robot. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1989. Bachelor's thesis.

Stephen Appleby and Simon Steward. Mobile software agents for control in telecommnications networks. *BT Technology Journal*, 12(2):104–113, March 1994.

Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

W. F. Asaad, G. Rainer, and E. K. Miller. Task-specific neural activity in the primate prefrontal cortex. *Journal of Neurophysiology*, 84:451–459, 2000.

Christopher G. Atkeson, Andrwe W. Moore, and Stefan Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, 11:75–113, 1997.

Dana H. Ballard, Mary M. Hayhoe, Polly K. Pook, and Rajesh P. N. Rao. Deictic codes for the embodiment of cognition. *Brain and Behavioral Sciences*, 20(4), december 1997.

D. M. Bannerman, M. A. Good, S. P. Butcher, M. Ramsay, and R. G. M. Morris. Distinct components of spatial learning revealed by prior training and ndma receptor blockade. *Nature*, 378:182–186, 1995.

Arvind K. Bansal, Kotagiri Ramohanarao, and Anand Rao. Distributed storage of replicated beliefs to facilitate recovery of distributed intelligent agents. In Munindar P. Singh, Anand S. Rao, and Michael J. Wooldridge, editors, *Intelligent Agents IV (ATAL97)*, pages 77–92, Providence, RI, 1998. Springer.

K. S. Barber and C. E. Martin. Agent autonomy: Specification, measurement, and dynamic adjustment. In *Proceedings of the Autonomy Control Software Workshop at Autonomous Agents (Agents'99)*, pages 8–15, Seattle, WA, 1999.

Joseph Bates, A. Bryan Loyall, and W. Scott Reilly. An architecture for action, emotion, and social behavior. Technical Report CMU-CS-92-144, CMU School of Computer Science, Pitsburgh, PA, May 1992.

A. Baumberg and D. C. Hogg. Generating spatio-temporal models from examples. *Image and Vision Computing*, 14(8):525–532, 1996.

Antoine Bechara, Hanna Damasio, Daniel Tranel, and Antonio R. Damasio. Deciding advantageously before knowing the advantageous strategy. *Science*, 275(5304):1293–1295, February 28 1995a.

Antoine Bechara, Daniel Tranel, Hanna Damasio, Ralph Adolphs, Charles Rockland, and Antonio R. Damasio. Double dissociation of conditioning and declarative knowledge relative to the amygdala and hippocampus in humans. *Science*, 269(5227):1115–1118, August 25 1995b.

Marc Bekoff and John Alexander Byers, editors. *Animal Play: Evolutionary, Comparative, and Ecological Perspectives*. Cambridge University Press, 1998.

Scott Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Stanford University, December 1996. Department of Computer Science.

Derek Bickerton. *Language & Species*. The University of Chicago Press, Chicago, Illinois, 1987.

Emilio Bizzi, Simon 4. Giszter, Eric Loeb, Ferdinando A. Mussa-Ivaldi, and Philippe Saltiel. Modular organization of motor behavior in the frog's spinal cord. *Trends in Neuroscience*, 18:442–446, 1995.

H. C. Blodgett. The effect of the introduction of reward upon the maze performance of rats. *University of California Publications in Psychology*, 4:113–134, 1929.

Bruce Mitchell Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, MIT, September 1996. Media Laboratory, Learning and Common Sense Section.

Barry W. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):22–32, August 1986.

R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):237–256, 1997.

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

Matthew Brand, Nuria Oliver, and Alex Pentland. Coupled hidden markov models for complex action recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR97)*, 1997.

Cynthia Breazeal. *Sociable Machines: Expressive Social Exchange Between Humans and Robots*. PhD thesis, MIT, Department of EECS, 2000.

Cynthia Breazeal and Brian Scassellati. A context-dependent attention system for a social robot. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI99)*, S.F., July 1999a. Morgan Kaufmann Publishers.

Cynthia Breazeal and Brian Scassellati. How to build robots that make friends and influence people. In *International Conference on Intelligent Robots (IROS-99)*, pages 858–863, Kyongju, Korea, 1999b.

Frederick P. Brooks, Jr. *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Publishing Company, Reading, MA, 20th anniversary edition edition, 1995.

Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.

Rodney A. Brooks. A robot that walks: Emergent behavior from a carefully evolved network. *Neural Computation*, 1(2):253–262, 1989.

Rodney A. Brooks. Intelligence without reason. In *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*, pages 569–595, Sydney, August 1991a.

Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991b.

Rodney A. Brooks and Anita M. Flynn. Fast, cheap and out of control; a robot invasion of the solar system. *Journal of the British Interplanetary Society*, 42(10):478–485, 1989.

Joanna Bryson. The subsumption strategy development of a music modelling system. Master's thesis, University of Edinburgh, 1992. Department of Artificial Intelligence.

Joanna Bryson. The reactive accompanist: Adaptation and behavior decomposition in a music system. In Luc Steels, editor, *The Biology and Technology of Intelligent Autonomous Agents*, pages 365–376. Springer, berlin, 1995.

Joanna Bryson. Hierarchy and sequence vs. full parallelism in reactive action selection architectures. In *From Animals to Animats 6 (SAB00)*, pages 147–156, Cambridge, MA, 2000a. MIT Press.

Joanna Bryson. The study of sequential and hierarchical organisation of behaviour via artificial mechanisms of action selection, 2000b. MPhil Thesis, University of Edinburgh.

Joanna Bryson, Keith Decker, Scott DeLoach, Michael Huhns, and Michael Wooldridge. Agent development tools. In C. Castelfranchi and Y. Lespérance, editors, *The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer, 2001. *in press*.

Joanna Bryson, Will Lowe, and Lynn Andrea Stein. Hypothesis testing for complex agents. In *NIST Workshop on Performance Metrics for Intelligent Systems*, Washington, DC, August 2000. in press.

Joanna Bryson and Brendan McGonigle. Agent architecture as object oriented design. In Munindar P. Singh, Anand S. Rao, and Michael J. Wooldridge, editors, *The Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL97)*, pages 15–30, Providence, RI, 1998. Springer.

Joanna Bryson and Kristinn R. Thórisson. Dragons, bats & evil knights: A three-layer design approach to character based creative play. *Virtual Reality*, 2001. *in press*.

Joanna J. Bryson and Jessica Flack. Emotions and action selection in an artificial life model of social behavior in non-human primates. *under review*, 2001.

John Bullinaria. Exploring the Baldwin effect in evolving adaptable control systems. In Robert French and Jacques Sougné, editors, *The Sixth Neural Computation and Psychology Workshop (NCPW6)*. Springer, 2000.

Richard W. Byrne and Anne E. Russon. Learning by imitation: a hierarchical approach. *Brain and Behavioral Sciences*, 21(5):667–721, 1998.

Dolores Cañamero. A hormonal model of emotions for behavior control. Technical Report 97-06, Vrije Universiteit Brussel, Artificial Intelligence Laboratory, 1997.

William H. Calvin. *The Cerebral Code*. MIT Press, 1996.

Niel R. Carlson. *Physiology of Behavior*. Allyn and Bacon, Boston, 2000.

Margaret Chalmers and Brendan McGonigle. Are children any more logical than monkeys on the five term series problem? *Journal of Experimental Child Psychology*, 37:355–377, 1984.

David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

David Chapman. Penguins can make cake. *AI Magazine*, 10(4):51–60, 1989.

David Chapman. Vision, instruction, and action. Technical Report 1204, Massachusetts Institute of TechnologyArtificial Intelligence Lab, Cambridge, Massachusetts, April 1990.

Paolo Ciancarini and Michael Wooldridge, editors. *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*. Springer, Berlin, 2001.

Peter Coad, David North, and Mark Mayfield. *Object Models: Strategies, Patterns and Applications*. Prentice Hall, 2nd edition, 1997.

Michael H. Coen. Building brains for rooms: Designing distributed software agents. In *Proceedings of the Ninth Conference on Innovative Applications of Artificial Intelligence (IAAI97)*, Providence, RI, 1997.

Jonathan H. Connell. *Minimalist Mobile Robotics: A Colony-style Architecture for a Mobile Robot*. Academic Press, Cambridge, MA, 1990. also MIT TR-1151.

Richard Cooper, Tim Shallice, and Jonathon Farringdon. Symbolic and continuous processes in the automatic selection of actions. In John Hallam, editor, *Hybrid Problems, Hybrid Solutions, Frontiers in Artificial Intelligence and Applications*, pages 27–37. IOS Press, Amsterdam, 1995.

M. G. Core, J. D. Moore, C. Zinn, and P. Wiemer-Hastings. Modeling human teaching tactics in a computer tutor. In *Proceedings of the ITS'00 Workshop on Modelling Human Teaching Tactics and Strategies*, Montreal, 2000.

Luis Correia and A. Steiger-Garção. A useful autonomous vehicle with a hierarchical behavior control. In F. Moran, A. Moreno, J.J. Merelo, and P. Chacon, editors, *Advances in Artificial Life (Third European Conference on Artificial Life)*, pages 625–639, Berlin, 1995. Springer.

Antonio R. Damasio. *The Feeling of What Happens: Body and Emotion in the Making of Consciousness*. Harcourt, 1999.

Kerstin Dautenhahn and Chrystopher Nehaniv, editors. *Proceedings of the AISB'99 Symposium on Imitation in Animals and Artifacts*, Edinburgh, April 1999. AISB.

Richard Dawkins. Hierarchical organisation: 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, Cambridge, 1976.

Scott A. DeLoach and Mark Wood. Developing multiagent systems with agentTool. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII: The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL 2000)*. Springer-Verlag, Berlin, 2001.

John Demiris. *Movement Imitation Mechanisms in Robots and Humans*. PhD thesis, University of Edinburgh, May 1999. Department of Artificial Intelligence.

John Demiris and Gillian Hayes. Active and passive routes to imitation. In *Proceedings of the AISB'99 Symposium on Imitation in Animals and Artifacts*, Edinburgh, April 1999. AISB.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society series B*, 39:1–38, 1977.

Daniel C. Dennett. *The Intentional Stance*. The MIT Press, Massachusetts, 1987.

Daniel C. Dennett. *Darwin's Dangerous Idea*. Penguin, 1995.

Daniel C. Dennett and Marcel Kinsbourne. Time and the observer: The where and when of consciousness in the brain. *Brain and Behavioral Sciences*, 15:183–247, 1992.

Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In Munindar P. Singh, Anand S. Rao, and Michael J. Wooldridge, editors, *Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages*, pages 155–176, Providence, RI, July 1997. Springer.

Jon Doyle, Isaac Kohane, William Long, and Peter Szolovits. The architecture of MAITA: A tool for monitoring, analysis, and interpretation. Technical report, MIT LCS Clinical Decision Making Group, 1999. draft of September 21.

Hubert L. Dreyfus. *What Computers Still Can't Do*. MIT Press, Cambridge, MA, 1992.

Jeffrey L. Elman, Elizabeth A. Bates, Mark H. Johnson, Annette Karmiloff-Smith, Domenico Parisi, and Kim Plunkett. *Rethinking Innateness. A Connectionist Perspective on Development*. MIT Press, Cambridge, MA, 1996.

Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 202–207, 1987.

R. James Firby. *The RAP Language Manual*. AI Laboratory, University of Chicago, ver. 1.0 edition, March 1995.

Jerry A. Fodor. *The Modularity of Mind*. Bradford Books. MIT Press, Cambridge, MA, 1983.

Charles L. Forgy. A fast algorithm for the many patterns many objects match problem. *Artificial Intelligence*, 19(3):17–37, 1982.

Carl Frankel and Rebecca Ray. Emotion, intention and the control architecture of adaptively competent information processing. In Aaron Sloman, editor, *AISB'00 Symposium on Designing a Functioning Mind*, pages 63–72, 2000.

Robert French, Bernard Ans, and Stephane Rousset. Pseudopatterns and dual-network memory models: Advantages and shortcomings. In Robert French and Jacques Sougné, editors, *The Sixth Neural Computation and Psychology Workshop (NCPW6)*. Springer, 2000.

Sigmund Freud. *The Interpretation of Dreams*. Avon, New York, 1900.

Sandra Clara Gadanho. *Reinforcement Learning in Autonomous Robots: An Empirical Investigation of the Role of Emotions*. PhD thesis, University of Edinburgh, 1999.

C.R. Gallistel, Ann L. Brown, Susan Carey, Rochel Gelman, and Frank C. Keil. Lessons from animal learning for the study of cognitive development. In Susan Carey and Rochel Gelman, editors, *The Epigenesis of Mind*, pages 3–36. Lawrence Erlbaum, Hillsdale, NJ, 1991.

J. Garcia and R. A. Koelling. The relation of cue to consequence in avoidance learning. *Psychonomic Science*, 4:123–124, 1966.

Erann Gat. *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots*. PhD thesis, Virginia Polytechnic Institute and State University, 1991.

Erran Gat. Three-layer architectures. In David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter 8, pages 195–210. MIT Press, Cambridge, MA, 1998.

Tim van Gelder. The dynamical hypothesis in cognitive science. *Behavioral and Brain Sciences*, 21(5):616–665, 1998.

M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

Andrew Gillies and Gordon W. Arbuthnott. Computational models of the basal ganglia. *Movement Disorders*, 15(5):762–770, September 2000.

Henry Gleitman. *Psychology*. Norton, 4 edition, 1995.

Stephen Grand, Dave Cliff, and Anil Malhotra. Creatures: Artificial life autonomous software agents for home entertainment. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents*, pages 22–29. ACM press, February 1997.

M. S. A. Graziano and S. Gandhi. Location of the polysensory zone in the precentral gyrus of anesthetized monkeys. *Experimental Brain Research*, 135(2):259–66, November 2000.

M. S. A. Graziano, C. S. R. Taylor, and T. Moore. Brain mechanisms for monitoring and controlling the body. personal communication, February 2001.

Stephen Grossberg. How does the cerebral cortex work? Learning, attention and grouping by the laminar circuits of visual cortex. *Spatial Vision*, 12:163–186, 1999.

Kevin Gurney, Tony J. Prescott, and Peter Redgrave. The basal ganglia viewed as an action selection device. In *The Proceedings of the International Conference on Artificial Neural Networks*, Skovde, Sweden, September 1998.

B. E. Hallam, J. R. P. Halperin, and J. C. T. Hallam. An ethological model for implementation on mobile robots. *Adaptive Behavior*, 3:51–80, 1995.

Kristian J. Hammond. Case-based planning: A framework for planning from experience. *The Journal of Cognitive Science*, 14(3), September 1990.

Mitch R. Harris. *Computational Modelling of Transitive Inference: a Micro Analysis of a Simple Form of Reasoning*. PhD thesis, University of Edinburgh, 1988.

Mitch R. Harris and Brendan O. McGonigle. A modle of transitive choice. *The Quarterly Journal of Experimental Psychology*, 47B(3):319–348, 1994.

George Hartmann and Rüdiger Wehner. The ant's path integration system: A neural architecture. *Bilogical Cybernetics*, 73:483–497, 1995.

Barbara Hayes-Roth and Robert van Gent. Story-making with improvisational puppets. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents*, pages 1–7. ACM press, February 1997.

D.O. Hebb. *The Organization of Behavior*. John Wiley and Sons, New York, New York, 1949.

Horst Hendriks-Jansen. *Catching Ourselves in the Act: Situated Activity, Interactive Emergence, Evolution, and Human Thought*. MIT Press, Cambridge, MA, 1996.

R. N. A. Henson and N. Burgess. Representations of serial order. In J. A. Bullinaria, D. W. Glasspool, and G. Houghton, editors, *Proceedings of the Fourth Neural Computation and Psychology Workshop: Connectionist Representations*, London, 1997. Springer.

Richard N. A. Henson. *Short-term Memory for Serial Order*. PhD thesis, University of Cambridge, November 1996. St. John's College.

Richard N. A. Henson. Short-term memory for serial order: the start-end model. *Cognitive Psychology*, 36:73–137, 1998.

Henry Hexmoor, Ian Horswill, and David Kortenkamp. Special issue: Software architectures for hardware agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):147–156, 1997.

Henry H. Hexmoor. *Representing and Learning Routine Activities*. PhD thesis, State University of New York at Buffalo, December 1995.

K. Hindriks, F. De Boer, W. Van Der Hoek, and J.-J. C. Meyer. Control structures of rule-based agent languages. In J.P. Müller, M.P. Singh, and A.S. Rao, editors, *The Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL98)*, pages 381–396, 1999.

Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII: The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL 2000)*. Springer-Verlag, Berlin, 2001.

P. N. Hineline and H. Rachlin. Escape and avoidance of shock by pigeons pecking a key. *Journal of Experimental Analysis of Behavior*, 12:533–538, 1969.

Geoffrey E. Hinton and Zoubin Ghahramani. Generative models for discovering sparse distributed representations. *Philosophical Transactions of the Royal Society B*, 352: 1177–1190, 1997.

Geoffrey E. Hinton and Steven J. Nowlan. How learning can guide evolution. *Complex Systems*, 1:495–502, 1987.

Berthald K. P. Horn and Patrick H. Winston. A laboratory environment for applications oriented vision and manipulation. Technical Report 365, MIT AI Laboratory, 1976.

Todd S. Horowitz and Jeremy M. Wolfe. Visual search has no memory. *Nature*, 357: 575–577, August 6 1998.

Ian D. Horswill. *Specialization of Perceptual Processes*. PhD thesis, MIT, Department of EECS, Cambridge, MA, May 1993.

Ian D. Horswill. Visual routines and visual search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, August 1995.

Ian D. Horswill. Visual architecture and cognitive architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):277–293, 1997.

George Houghton and Tom Hartley. Parallel models of serial behavior: Lashley revisited. *PSYCHE*, 2(25), February 1995.

Marcus J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents’99)*, pages 236–243, Seattle, May 1999.

David Hume. *Philisophical Essays Concerning Human Understanding*. Andrew Millar, London, 1748.

Mark Humphrys. *Action Selection methods using Reinforcement Learning*. PhD thesis, University of Cambridge, June 1997.

Allan D. Jepson, Whitman Richards, and David Knill. Modal structure and reliable inference. In D. Knill and W. Richards, editors, *Perception as Bayesian Inference*, pages pages 63–92. Cambridge University Press, 1996.

Leslie Pack Kaelbling. Why robbie can't learn: The difficulty of learning in autonomous agents. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, August 1997. Morgan Kaufmann. IJCAI Computers and Thought Award talk.

Annette Karmiloff-Smith. *Beyond Modularity: A Developmental Perspective on Cognitive Change*. MIT Press, Cambridge, MA, 1992.

J. Scott Kelso. *Dynamic Patterns: The Self-Organization of Brain and Behavior*. MIT Press, 1995.

Simon Kirby. *Function, Selection and Innateness: the Emergence of Language Universals*. Oxford University Press, 1999.

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In *Proceedings of The First International Conference on Autonomous Agents*. The ACM Press, 1997.

Christopher Kline and Bruce Blumberg. The art and science of synthetic character design. In *AISB'99 Symposium on AI and Creativity in Entertainment and Visual Art*, pages 16–21, 1999.

Chris Knight, Michael Studdert-Kennedy, and James R. Hurford, editors. *The Evolutionary Emergence of Language: Social function and the origins of linguistic form*. Cambridge University Press, 2000.

K. Knight and V. Hatzivassilogon. Two-level, many-paths generation. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguists (ACL95)*, pages 252–260, 1995.

T. Kobayashi, H. Nishijo, M. Fukuda, J. Bures, and T. Ono. Task-dependent representations in rat hippocampal place neurons. *JOURNAL OF NEUROPHYSIOLOGY*, 78(2):597–613, 1997.

Kurt Konolige and Karen Myers. The saphira architecture for autonomous mobile robots. In David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter 9, pages 211–242. MIT Press, Cambridge, MA, 1998.

David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press, Cambridge, MA, 1998.

John E. Laird and Paul S. Rosenbloom. The evolution of the Soar cognitive architecture. In D.M. Steier and T.M. Mitchell, editors, *Mind Matters*. Erlbaum, 1996.

K. S. Lashley. The problem of serial order in behavior. In L. A. Jeffress, editor, *Cerebral mechanisms in behavior*. John Wiley & Sons, New York, 1951.

Joseph LeDoux. *The Emotional Brain : The Mysterious Underpinnings of Emotional Life*. Simon and Schuster, New York, 1996.

Egbert Giles Leigh, Jr. Levels of selection, potential conflicts, and their resolution: The role of the ”common good”. In Laurent Keller, editor, *Levels of Selection in Evolution*, pages 15–30. Princeton University Press, Princeton, NJ, 1999.

V. J. Leon, D. Kortenkamp, and D. Schreckenghost. A planning, scheduling and control architecture for advanced life support systems. In *NASA Workshop on Planning and Scheduling for Space*, Oxnard, CA, October 1997.

James C. Lester and Brian A. Stone. Increasing believability in animated pedagogical agents. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents*, pages 16–21. ACM press, February 1997.

Joseph S. Lonstein and Judith M. Stern. Role of the midbrain periaqueductal gray in maternal nurturance and aggression: *c-fos* and electrolytic lesion studies in lactating rats. *Journal of Neuroscience*, 17(9):3364–78, May 1 1997.

Konrad Lorenz. *Foundations of Ethology*. Springer, New York, 1973.

Will Lowe. Meaning and the mental lexicon. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, August 1997. Morgan Kaufmann.

Pattie Maes. How to do the right thing. A.I. Memo 1180, MIT, Cambridge, MA, December 1989.

Pattie Maes, editor. *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and back*. MIT Press, Cambridge, MA, 1990a.

Pattie Maes. Situated agents can have goals. In Pattie Maes, editor, *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and back*, pages 49–70. MIT Press, Cambridge, MA, 1990b.

Pattie Maes. The agent network architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991a.

Pattie Maes. A bottom-up mechanism for behavior selection in an artificial creature. In Jean-Arcady Meyer and Stuart Wilson, editors, *From Animals to Animats*, pages 478–485, Cambridge, MA, 1991b. MIT Press.

Chris Malcolm. A hybrid behavioural/knowledge-based approach to robotic assembly. DAI Research Paper 875, University of Edinburgh, Edinburgh, Scotland, Oct 1997.

Matthew Marjanovic, Brian Scassellati, and Matthew Williamson. Self-taught visually-guided pointing for a humanoid robot. In Pattie Maes, Maja J. Matarić, Jean-Arcady Meyer, Jordan Pollack, and Stewart W. Wilson, editors, *From Animals to Animats 4 (SAB96)*, Cambridge, MA, 1996. MIT Press.

Maja J. Matarić. A distributed model for mobile robot environment-learning and navigation. Technical Report 1228, Massachusetts Institute of Technology Artificial Intelligence Lab, Cambridge, Massachusetts, June 1990.

Maja J. Matarić. Designing emergent behaviors: From local interactions to collective intelligence. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 432–441, Cambridge, MA, December 1992. MIT Press.

Maja J. Matarić. Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):323–336, 1997.

T. Matheson. Hindleg targeting during scratching in the locust. *Journal of Experimental Biology*, 200:93–100, 1997.

J. L McClelland and D. E. Rumelhart, editors. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press, 1988. two volumes.

James L. McClelland, Bruce L. McNaughton, and Randall C. O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102(3):419–457, 1995.

Brendan McGonigle. Incrementing intelligent systems by design. In Jean-Arcady Meyer and Stuart Wilson, editors, *From Animals to Animats*, pages 478–485, Cambridge, MA, 1991. MIT Press.

Brendan McGonigle and Margaret Chalmers. Are monkeys logical? *Nature*, 267, 1977.

Brendan McGonigle and Margaret Chalmers. The ontology of order. In Les Smith, editor, *Piaget: A Critical Assessment*. Routledge, 1996.

Brendan McGonigle and Margaret Chalmers. Rationality as optimised cognitive self-regulation. In M. Oaksford and N. Chater, editors, *Rational Models of Cognition*. Oxford University Press, 1998.

H. McGurk and J. MacDonald. Hearing lips and seeing voices. *Nature*, 264:746–748, 1976.

J-A. Meyer, A. Berthoz, D. Floreano, H. Roitblat, and S. W. Wilson, editors. *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior*. MIT Press, Cambridge, MA, 2000.

John-Jules Ch. Meyer. Agent languages and their relationship to other programming paradigms. In J.P. Müller, M.P. Singh, and A.S. Rao, editors, *The Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL98)*, pages 309–316, Paris, 1999. Springer.

Frank A. Middleton and Peter L. Strick. Basal ganglia output and cognition: evidence from anatomical, behavioral, and clinical studies. *Brain and Cognition*, 42(2):183–200, 2000.

Jonathon W. Mink. The basal ganglia: focused selection and inhibition of competing motor programs. *Progress In Neurobiology*, 50(4):381–425, 1996.

Marvin Minsky. *The Society of Mind*. Simon and Schuster Inc., New York, NY, 1985.

Karen L. Myers. A procedural knowledge approach to task-level control. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinburgh, 1996.

Karen L. Myers. *Procedural Reasoning System User's Guide*. Artificial Intelligence Center, SRI International, Menlo Park, CA, USA, 1.96 edition, 1997,1999.

Keiichi Nakata. *A Causal Reasoning Approach to Behaviour-Oriented Design*. PhD thesis, University of Edinburgh, Department of Artificial Intelligence, 1994.

J. H. Neely. Semantic priming effects in visual word recognition: A selective review of current findings and theories. In D. Besner and G. W. Humphreys, editors, *Basic Processes in Reading: Visual Word Recognition*, chapter 9. Lawrence Erlbaum Associates, 1991.

O. Nestares and David J. Heeger. Robust multiresolution alignment of MRI brain volumes. *Magnetic Resonance in Medicine*, 43:705–715, 2000.

Alan Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

Nils Nilsson. Shakey the robot. Technical note 323, SRI International, Menlo Park, California, April 1984.

Nils Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

Donald. A. Norman and Tim Shallice. Attention to action: Willed and automatic control of behavior. In R.Davidson, G. Schwartz, and D. Shapiro, editors, *Consciousness and Self Regulation: Advances in Research and Theory*, volume 4, pages 1–18. Plenum, New York, 1986.

Jon Oberlander and Chris Brew. Stochastic text generation. *Philosophical Transactions of the Royal Society of London, Series A*, 358, April 2000.

Lynne E. Parker. ALLIANCE: An architecture for fault tolerant multi-robot cooperation. *IEEE Transactions on Robotics and Automation*, 14(2), 1998.

David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–7, 1986.

Miles Pebody. Learning and adaptivity: Enhancing reactive behaviour architectures in real-world interaction systems. In F. Moran, A. Moreno, J.J. Merelo, and P. Chacon, editors, *Advances in Artificial Life (Third European Conference on Artificial Life)*, pages 679–690, Berlin, 1995. Springer.

Jean Piaget. *The Construction of Reality in the Child*. Basic Books, New York, 1954.

E. Pöppel. Temporal mechanisms in perception. *International Review of Neurobiology*, 37: 185–202, 1994.

Robert F. Port and Timothy van Gelder, editors. *Mind as Motion: Explorations in the Dynamics of Cognition*. MIT Press, Cambridge, MA, 1995.

Tony J. Prescott, Kevin Gurney, F. Montes Gonzalez, and Peter Redgrave. The evolution of action selection. In David McFarland and O. Holland, editors, *Towards the Whole Iguana*. MIT Press, Cambridge, MA, to appear.

Ashwin Ram and Juan Carlos Santamaria. Continuous case-based reasoning. *Artificial Intelligence*, 90(1–2):25–77, 1997.

V. S. Ramachandran and S. Blakeslee. *Phantoms in the brain: Human nature and the architecture of the mind*. Fourth Estate, London, 1998.

Peter Redgrave, Tony J. Prescott, and Kevin Gurney. The basal ganglia: a vertebrate solution to the selection problem? *Neuroscience*, 89:1009–1023, 1999.

W. Scott Neal Reilly. Believable social and emotional agents. Technical Report CMU-CS-96-138, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1996. Ph.D. thesis.

Ronald A. Rensink. The dynamic representation of scenes. *Visual Cognition*, 7:17–42, 2000.

C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

Bradley Rhodes. Pronemes in behavior nets. Technical Report 95-01, MIT, 1995. Media Lab, Learning and Common Sense.

Bradley Rhodes. PHISH-nets: Planning heuristically in situated hybrid networks. Master's thesis, MIT, 1996. Media Lab, Learning and Common Sense.

Patrick Riley, Peter Stone, and Manuela Veloso. Layered disclosure: Revealing agents' internals. In C. Castelfranchi and Y. Lespérance, editors, *The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer, 2001. *in press*.

T. J. Roper. Learning as a biological phenomena. In T. R. Halliday and P. J. B. Slater, editors, *Genes, Development and Learning*, volume 3 of *Animal Behaviour*, chapter 6, pages 178–212. Blackwell Scientific Publications, Oxford, 1983.

K. Rosenblatt and D. Payton. A fine-grained alternative to the subsumption architecture for mobile robot control. In *Proceedings of the IEEE/INNS International Joint Conference on Neural Networks*, 1989.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence : A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

Tuomas W. Sandholm. Distributed rational decision making. In Gerhard Wei, editor, *Multiagent Systems*, pages 201–259. The MIT Press, Cambridge, Massachusetts, 1999.

Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences*, 3(6):233–242, 1999.

Stefan Schaal and Christopher G. Atkeson. Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, 14(1):57–71, 1994.

J. R. Searle. Minds, brains and programs. *Brain and Behavioral Sciences*, 3:417–424, 1980.

Phoebe Sengers. Do the thing right: An architecture for action expression. In Katia P Sycara and Michael Wooldridge, editors, *Proceedings of the Second International Conference on Autonomous Agents*, pages 24–31. ACM Press, 1998.

Phoebe Sengers. *Anti-Boxology: Agent Design in Cultural Context*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.

R. M. Seyfarth, D. L. Cheney, and P. Marler. Monkey responses to three different alarm calls: Evidence of predator classification and semantic communication. *Science*, 14: 801–803, 1980.

M. Siemann and J. D. Delius. Implicit deductive reasoning in humans. *Naturwissenshaften*, 80:364–366, 1993.

W.E. Skaggs and B.L. McNaughton. Spatial firing properties of hippocampal ca1 populations in an environment containing two visually identical regions. *Journal of Neuroscience*, 18(20):8455–8466, 1998.

Aaron Sloman and Brian Logan. Architectures and tools for human-like agents. In *Proceedings of the Second European Conference on Cognitive Modelling*, pages 58–65, Nottingham, UK, April 1998. University of Nottingham Press.

William D. Smart. Location recognition wih neural networks in a mobile robot. Master's thesis, University of Edinburgh, 1992. Department of Artificial Intelligence.

Luc Steels. Building agents with autonomous behavior systems. In Luc Steels and Rodney Brooks, editors, *The 'artificial life' route to 'artificial intelligence'. Building situated embodied agents*. Lawrence Erlbaum Associates, New Haven, 1994a.

Luc Steels. A case study in the behavior-oriented design of autonomous agents. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats (SAB94)*, pages 445–452, Cambridge, MA, 1994b. MIT Press. ISBN 0-262-53122-4.

Peter Stone and Manuela Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110 (2):241–273, 1999.

V. Sundareswaran and L. M. Vaina. Learning direction in global motion: Two classes of psychophysically-motivated models. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, Denver, CO, 1994.

M. Sur, A. Angelucci, and J. Sharma. Rewiring cortex: The role of patterned activity in development and plasticity of neocortical circuits. *Journal of Neurobiology*, 41:33–43, 1999.

J. Tanji and K. Shima. Role for supplementary motor area cells in planning several movements ahead. *Nature*, 371:413–416, 1994.

T. J. Teyler and P. Discenna. The hippocampal memory indexing theory. *Behavioral Neuroscience*, 100:147–154, 1986.

Kristinn R. Thórisson. *Communicative Humanoids: A Computational Model of Psychosocial Dialogue Skills*. PhD thesis, MIT Media Laboratory, September 1996.

Kristinn R. Thórisson. Layered modular action control for communicative humanoids. In Nadia Magnenat Thalmann, editor, *Computer Animation '97*, pages 134–143, Geneva, June 1997. IEEE Press.

Kristinn R. Thórisson. Real-time decision making in face to face communication. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the Second International Conference on Autonomous Agents (Agents '98)*, pages 16–23, Minneapolis, 1998. ACM Press.

Kristinn R. Thórisson. A mind model for multimodal communicative creatures & humanoids. *International Journal of Applied Artificial Intelligence*, 13(4/5):519–538, 1999.

E. C. Tolman. Cognitive maps in rats and men. *Psychological Review*, 55:189–208, 1948.

E. C. Tolman and C. H. Honzik. Introduction and removal of reward, and the maze performance in rats. *University of California Publications in Psychology*, 4:257–275, 1930.

Xiaoyuan Tu. *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception and Behavior*. Springer, 1999.

Toby Tyrrell. *Computational Mechanisms for Action Selection*. PhD thesis, University of Edinburgh, 1993. Centre for Cognitive Science.

Shimon Ullman. Visual routines. *Cognition*, 18:97–159, 1984.

Frans B. M. de Waal. *Good Natured: The origins of right and wrong in humans and other animals*. Harvard University Press, Cambridge, MA, 1996.

Frans B. M. de Waal. Primates—a natural heritage of conflict resolution. *Science*, 289: 586–590, 2000.

Frans B. M. de Waal. *The Ape and the Sushi Master: Cultural Reflections of a Primatologist*. Basic Books, Boulder, CO, 2001.

Frans B. M. de Waal and Lesleigh Luttrell. Toward a comparative socioecology of the genus *macaca*: Different dominance styles in rhesus and stumptailed macaques. *American Journal of Primatology*, 19:83–109, 1989.

Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

Stefan Wermter, Jim Austion, and David Willshaw, editors. *Emergent Neural Computational Architectures Based on Neuroscience.* Springer, 2001. *in press*.

Steven D. Whitehead. Reinforcement learning for the adaptive control of perception and action. Technical Report 406, University of Rochester Computer Science, Rochester, NY, Feb 1992.

Andrew Whiten. Primate culture and social learning. *Cognitive Science*, 24(3):477–508, 2000.

S. I. Wiener. Spatial, behavioral and sensory correlates of hippocampal CA1 complex spike cell activity: Implications for information processing functions. *Progress in Neurobiology*, 49(4):335, 1996.

D. E. Wilkins and K. L. Myers. A multiagent planning architecture. In *Proceedings of AIPS-98*, pages 154–162, Pittsburgh, PA, June 1998.

D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1): 197–227, 1995.

Matthew Wilson and Bruce McNaughton. Reactivation of hippocampal ensemble memories during sleep. *Science*, 261:1227–1232, 29 July 1994.

Patrick Winston. Learning structural descriptions from examples. In Patrick Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill Book Company, New York, 1975.

David H. Wolpert. The existence of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1391–1420, 1996a.

David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996b.

Michael Wooldridge and Paul E. Dunne. Optimistic and disjunctive agent design problems. In C. Castelfranchi and Y. Lespérance, editors, *Intelligent Agents VII: The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL 2000)*. Springer-Verlag, Berlin, 2001.

Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

Michael J. Wooldridge and Nicholas R. Jennings, editors. *Intelligent Agents: the ECAI-94 workshop on Agent Theories, Architectures and Languages*, Amsterdam, 1994. Springer.

Clive D. L. Wynne. A minimal model of transitive inference. In C. D. L. Wynne and J. E. R. Staddon, editors, *Models of Action*, pages 269–307. Lawrence Erlbaum Associates, Mahwah, N.J., 1998.

Lispworks Professional Edition. Xanalys, Waltham, MA, 4.1.18 edition, 1999. (formerly Harlequin).