Build-a-Dude

Action Selection Networks for Computational Autonomous Agents

Submitted to the Media Arts and Sciences Section in the School of Architecture and Planning in partial fulfillment of the requirements for the degree of Master of Science in Visual Studies at the Massachusetts Institute of Technology, February 1991.

by Michael Boyle Johnson
Bachelor of Science, Department of Computer Science
University of Illinois at Urbana-Champaign, Illinois, 1988

Author
Media Arts and Sciences Section
January 11, 1991

Certified by
David L. Zeltzer
Associate Professor of Computer Graphics
Thesis Supervisor

Accepted by
Stephen A. Benton
Departmental Committee on Graduate Students

Build-a-Dude

Action Selection Networks for Computational Autonomous Agents

by Michael Boyle Johnson

Submitted to the Media Arts and Sciences Section in the School of Architecture and Planning for the degree of Master of Science in Visual Studies on January 11, 1991.

Abstract

I present the ubiquitous problem of selecting the next appropriate action in a given situation, with an emphasis on its application for a computational autonomous agent in an animation/simulation system. I review the relevant literature. I describe an algorithm for action selection, derived from one originally presented by Maes, describe extensions to this algorithm, with an emphasis on efficient distributed implementations of the algorithm. I present a parallel distributed implementation which encompasses both the original algorithm and many of my proposed extensions. I informally verify that the implementation satisfies the mathematical model, and give several detailed examples of the implementation in use with an emphasis on showing the extensions I have made. I discuss some of the limitations of the current theory, the implementation, and current and future directions of this work toward alleviating some of these problems.

Thesis Supervisor: David L. Zeltzer
Title: Associate Professor of Computer Graphics

This work was supported in part by the National Science Foundation (Grant IRI-8712772), and equipment grants from Stardent Computer, Hewlett-Packard Co., and Apple Computer, Inc.

Acknowledgements

- To my parents, Genevieve and Robert, and my sisters, Helen, Marirose, and Jean. Their love and support have gotten me here, and their individual examples of strength have inspired me.

- To Elizabeth; my friend and my love. For late night pizza, for music and movies, for desktop-design-on-demand, for understanding, and for love. I know, I know, *next* time I can take as long as I want, as long as I get the degree in 3 years...

- To my advisor David Zeltzer, who gave me the chance to study and learn here at the Media Lab and the freedom to pursue this work.

- To Pattie Maes, who has been so generous with her ideas, her time, and her source code.

- To my fellow late-night Labbers Straz and Bob, for rides and repartee.

- To Nicholas Negroponte, for the gift and challenge of the Media Lab, and for allowing it to become the wonderfully strange place that it is today.

- To Dave Chen and Steve Pieper, whose implementation of 3d and bolio, respectively, informed the implementation of the work in this thesis immensely.

- To all the Labbers who have listened to me over the years and taught me all manner of things: David Atherton (the *original* GED), Steve Benton, Brent, Joe Chung, das, Glorianna Davenport, Marc Davis, Didier, djs, Judith, Steven Drucker, dsmall, dyoung, foof, gmo, Grace, halazar, Mike Hawley, Irfan, Jan Marie, jh, Kevin, Rap-Master Lasky, Laureen, Ben Lowengard, Mark Lucente, Elaine McCarthy, Martin, Tod Machover, michelle, mikey, Margaret Minsky, Marvin Minsky, Muriel, Mary Ann Norris, Janette Noss, Laura Robin, Sandy, Peter Schröder, Sylvaine, teo, Tinsley, Trevor, Wendy.

- This document was laid out in Aldus Pagemaker 4.0, and the illustrations were done in Aldus Freehand 2.0. Page design and illustrations were done with the amazing assistance of Elizabeth Glenewinkel.

Contents

1 Introduction
The Problem of Action Selection

If you cannot—in the long run—tell everyone what you have been doing, your doing has been worthless.

Erwin Schrödinger

The question of selecting the next appropriate action to take in a given situation is a ubiquitous one. In the field of computer graphics, with the advent of graphical simulation techniques and physically-based modeling systems, the problem manifests itself as the question of how to intelligently control the degrees of freedom (DOF) problem such systems present the user. Users, from animators to video game enthusiasts, are increasingly being given more degrees of freedom to control, and, in the process, being overwhelmed. If we as media technologists are to intelligently put the power of today's computing systems into the hands of non-computer graphicists, we need to allow users to interact with such systems at a much higher level than is currently possible.

DOF problem

The innumberable ways in which any action may be performed. (**Turvey 1977**)

(Turvey 1977)

Turvey, M.T. *Preliminaries to a Theory of Action With Reference to Vision*, in *Perceiving, Acting, and Knowing*. Lawrence Erlbaum Associates (1977).

This thesis represents my exploration into this problem from the perspective of a computational graphicist, interested in the problem of controlling simulations which present the user with a bewildering number of parameters to manipulate. I am interested in designing and implementing animation and simulation systems that are inhabited by intelligent, computational autonomous agents. Such agents would exist within the context of the animation/simulation system, and be able to autonomously interact with other simulations in the system, as well as with the users of such systems.

animation == simulation

Many researchers, myself included, consider animation as a form of simulation. This is sometimes referred to as *graphical simulation*. see (**Zeltzer 1985**)

(Zeltzer 1985)

Zeltzer, D. *Towards an Integrated View of 3-D Computer Animation*. Visual Computer 1,4 (December 1985).

Of equal importance to me is understanding how such systems can be efficiently implemented on current architectures of computing systems, as well as how to scale such implementations to take advantage of parallel computing by distributing computation over high speed networks. If we are to build such systems and test them, this is of vital interest.

The particular domain I have chosen for this work is understanding how to organize the behavior of a computational autonomous agent. This dovetails nicely with other work being done here at the MIT Media Lab in the Computer Graphics and Animation Group. It also paves the way for building generic simulator systems, which have potentially widespread application.

current CG&A work

Other researchers in CG&A are building graphical simulation systems and physically-based models which can simulate the motor skills of a virtual actor; what is needed is some high level form of controlling such systems.

Such systems could be used to test robots before they were ever physically built; such systems could become the virtual laboratory of the computational scientist of the near future. On a potentially widespread level, such simulator systems could be a platform for new forms of home entertainment. In such systems, users could construct their own simulacra—virtual actors and actresses which wandered about the network, interacting with other virtual constructions and real users. Conversely, it could be the entrance for humans to move beyond the physical—to touch something which has never existed physically, to interact with some visual, aural, or tactile phenomena which has never existed outside of simulation.

I have built a set of tools for defining and controlling the behavior of a graphically simulated autonomous agent (a **dude**), based on a theory presented in this thesis. Rather than focus on the design and implementation of complicated jointed-figure motions, I have concentrated on the problems of organizing and coordinating currently implementable motor skills to enable meaningful goal-driven behaviors.

dude

The use of the term *dude*, seemingly facetious, is intended to represent the way I want to see interaction occur in simulation systems. In the future, I want to say "Dude! What happened?" and have the computational autonomous agents I'm dealing with understand what I mean and do the right thing.

The theory for organizing motor skills into a behavior repertoire has taken its inspiration from the work of ethologists and other researchers who study the behavior of animals; human and otherwise. More pragmatically, the work done by Pattie Maes, David Zeltzer, and Marvin Minsky has directly influenced the development, details and implementation of this theory.

I believe that this thesis and the resulting set of tools, constitutes a working platform for learning about paradigms for the representation and control of the behavior of autonomous, graphically simulated agents, and will provide the basis for future exploration of these issues by myself and other researchers.

Map of the Thesis Document

This chapter is intended to inform the reader of the general scope and intent of this research and explain a bit about my particular perspective. The next chapter discusses relevant work done in many different disciplines—from previous work done by fellow computer graphicists, through research done by researchers in AI and robotics, to seminal work done by those who study animal behavior. The area to be covered is vast, and my treatment makes no attempt at being exhaustive. My intention is to point out some major themes and show where I have received my guidance and inspiration.

Chapter 3 presents an algorithm for the problem of action selection, embodied as a network of choices. This algorithm borrows liberally from the work of both David Zeltzer and Pattie Maes. Zeltzer broadly sketched such a sys-

tem in (**Zeltzer 1983**) Later, Maes independently elaborated a similar algorithm in (**Maes 1989**). She also presented a mathematical model of the algorithm, which is reproduced in Chapter 3 of this document. I include it as both an elegant and complete statement of her currently published algorithm, and because it was the starting point of my implementation. I then present extensions to this algorithm which (in conjunction with my advisor David Zeltzer) we have made to increase its usefulness both in general and for my chosen domain.

(Zeltzer 1983)

Zeltzer, D. *Knowledge-Based Animation.* Proceedings of ACM SIGGRAPH/SIGART Workshop on Motion (April 1983).

(Maes 1989)

Maes, P. *How to Do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology. (December 1989).

Chapter 4 presents my implementation of the algorithm outlined in Chapter 3. I describe a *skill network*, which consists of a set of motor skills corresponding to the simulated skills of a virtual actor. I describe some of the factors I considered in its design and implementation, and go on to describe in some detail exactly how the skill network was implemented as a set of distributed processes running on a network of workstations.

Chapter 5 discusses the current results of using the Build-a-Dude system to generate behavior. It first discusses a benchmark I used to calibrate one case with respect to Maes' original implementation. I then go on to discuss two other examples which have been simulated by the system. The first is a rather simple one, showing how the algorithm can be used to simulate some of the low-level workings mediating the execution of an animal's reflexes during the walk cycle. The second example has three parts, and concerns a one-armed virtual actor sitting in a chair with a drink in its hand who is then told to open a door. The first part shows how the algorithm causes the actor to select a series of actions to call the appropriate routines to get the door opened. The second part adds the complication of trying to get the dude to close a window on its way to opening the door. This is intended to show how the algorithm (and its current implementation) can handle parallel execution of tasks. Finally, the third example demonstrates how the algorithm deals with failure and run-time arbitration of new skills. When our virtual actor suddenly finds itself without the ability to walk, it discovers and uses its new found ability of crawling across the floor. Each example includes a synopsis of what happened, gleaned from the log files of the running implementation, and a discussion of what occurred.

Chapter 6 closes the thesis document proper with a discussion of some of the limitations of the current algorithm and its implementation. I talk about some of the work I am doing now, as well as work under consideration, to extend and improve the Build-a-Dude system. At the end of this chapter, I wrap up with some conclusions of what I have accomplished to date with this work.

Finally, there is a list of sources I used and several appendices, containing information I felt would have interrupted the flow of the thesis document, but that I wished to have available to the interested reader. The first appendix goes into some detail about the registry/dispatcher's inner loop, a key part of the implementation. The final appendix discusses a portable, network transparent, message passing library I designed and implemented which underlies the implementation of the Build-a-Dude system.

2 Related Work

Computer Graphics, AI, Robotics, and Ethology

If I have seen further it is by standing on the shoulders of giants.
Sir Isaac Newton

Task Level Animation

In (**Zeltzer 1985**), Zeltzer discusses a three part taxonomy of animation systems: *guiding*, *animator level*, and *task level*. *Guiding* includes motion recording, key-frame interpolation, and shape interpolation systems. *Animator level* systems allow algorithmic specification of motion. *Task level* animation systems must contain knowledge about the objects and environment being animated; the execution of the motor skills is organized by the animation system. The work undertaken in this thesis is an example of one component of a task level animation system.

(Zeltzer 1985)
Zeltzer, D. *Towards an Integrated View of 3-D Computer Animation.* Visual Computer 1(4) (December 1985).

In a task level animation system, there are several kinds of planning activity that can go on. In this work, I am concerned with only the lowest level of planning—what Zeltzer calls motor planning. Motor planning is similar to the kind of problem solver proposed by Simon & Newell in their GPS; which Minsky calls a *difference engine*. "This reflects current notions of how animal behavior is structured in what we call an expectation lattice, in which motor behavior is generated by traversing the hierarchy of skills selected by rules which map the current action and context onto the next desired action." (**Zeltzer 1987**)

(Zeltzer 1987)
Zeltzer, D. *Motor Problem Solving for Three Dimensional Computer Animation.* Proceedings of Proc. L'Imaginaire Numerique (May 14-16 1987).

The notion of designing motor skills and doing motor planning for animated agents, draws from the established fields of mathematics, physics, psychology, physiology, ethology, and newer, hybrid fields including kinesiology, neuroethology, artificial intelligence, robotics, and, of course, computer graphics. What follows is a brief overview of relevant research done in some of these areas.

Computer Graphics

Animated creatures that move realistically have long been a dream of computer graphicists. Recently, with the advent of *physically-based modeling* techniques, animated creatures exhibiting motion akin to the complexity of real creatures have been demonstrated. Physically-based modeling is a catch-all phrase used in the computer graphics community to denote the sim-

ulation of Newtonian physics to help automate motion. It includes forward and inverse kinematics, forward and inverse dynamics, constraints, finite element, and finite difference techniques.

Using forward kinematic techniques, Zeltzer showed a biped with many degrees of freedom that could walk over uneven terrain (**Zeltzer 1984**). His system was a step towards an animation system that allowed interaction at the task level, although the motor skills of the animated figures were limited to forward locomotion.

Girard's PODA system has creatures that can walk, run, turn, and dance using kinematics and point dynamics (**Girard 1985**). Again the emphasis in this system is on the animation of legged locomotion, and allowing the animator control over its creation. Autonomy of the animated creatures is not the goal, rather intelligent and artistic control by the animator is.

Sims designed a system for making creatures that, using inverse kinematics and simple dynamics, could navigate over uneven terrain (**Sims 1987**). This system was notable in that the notion of "walking" was generalized enough that he could generate many different kinds of creatures that all exhibited different behavior very quickly.

In (**Reynolds 1987**), Reynolds describes a system based on the actors model of distributed computation (**Agha 1985**) for animating the behavior of flocks and herds. The use of the actor model allows for a great amount of flexibility, but the communication overhead between actors imposed for their particular application is non-trivial ($O(n^2)$).

Also of note are Miller's snakes and worms, which use relatively simple notions about the motion of real snakes to generate quite interesting motion. The locomotion is controlled by a behavior function which allows the snake to be steered towards a target (**Miller 1988**).

Badler et al. describes a system in (**Badler 1990**) for translating NASA task protocols into animated sequences that portray astronauts performing specified tasks in a space station work environment. The focus of their research is concerned more with portraying and evaluating human motor performance for specified tasks, or for instructing agents in the performance of tasks, rather than the development of architectures for representing and implementing virtual actors.

One of the most ambitious animated creatures to date is a dynamic hexapod, being developed here in the Computer Graphics & Animation Group at the

(Zeltzer, 1984)

Zeltzer, D. *Representation and Control of Three Dimensional Computer Animated Figures.* Ph.D. Thesis. Ohio State University (August 1984).

(Girard 1985)

Girard, M. and A. A. Maciejewski. *Computational Modeling for the Computer Animation of Legged Figures.* Computer Graphics 19,3 (July 1985).

(Sims 1987)

Sims, K. *Locomotion of Jointed Figures over Complex Terrain,* S.M.V.S. Thesis. Massachusetts Institute of Technology (June 1987).

(Reynolds 1987)

Reynolds, C. W. *Flocks, Herds and Schools: A Distributed Behavioral Model.* Computer Graphics 21,4 (July 1987).

(Agha 1985)

Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems,* MIT Press (1985).

(Miller 1989)

Miller, G. *The Motion Dynamics of Snakes and Worms.* Computer Graphics 22,4 (August 1988).

(Badler 1990)

Badler, N. I. and B. L. Webber. *Animation from Instructions.* Making Them Move: Mechanics, Control and Animation of Articulated Figures. Morgan Kaufmann (1991).

MIT Media Lab by McKenna and Zeltzer (**McKenna 1990A**). They have demonstrated an articulated figure with 38 degrees of freedom, that uses the gait mechanism of a cockroach to drive a forward dynamic simulation of the creature moving over even and uneven terrain. It is an example of how successfully biologically-based control schemes can be adapted for computer animation. A virtual actor hexapod that uses the same gait controller and exhibits several simple behaviors has been also been demonstrated (**McKenna 1990B**).

(McKenna 1990A)

McKenna, M. *A Dynamic Model of Locomotion for Computer Animation*, S.M. Thesis. Massachusetts Institute of Technology (January 1990).

(McKenna 1990B)

McKenna, M., S. Pieper and D. Zeltzer. *Control of a Virtual Actor: The Roach.* Proceedings of Proc. 1990 Symposium on Interactive 3D Graphics (March 25-28, 1990).

Autonomous Agents from AI, Robotics and Machine Learning

The artificial intelligence community has long been fascinated by the notion of autonomous agents. Recently, systems containing agents with interesting behavior have been developed.

Minsky describes a theory in which a mind is composed of a society of interacting parts, each of which, considered by itself, is explicable and mindless, that he calls the *Society of Mind* (**Minsky 1987**). The work done by Travers for the Vivarium project here at the Media Lab contains good examples of systems of agents that are autonomous and exhibit interesting behavior (**Travers 1989**). His ideas are loosely based on Minsky's Society of Mind theory and model the behavior of groups of insects using perception sensors of the environment and agent-based representations of the state of each insect's "mind".

(Minsky 1987)

Minsky, M. *The Society of Mind.* Simon and Schuster (1987).

(Travers 1989)

Travrers, M. *D. A Agar: An Animal Construction Kit*, S.M. Thesis. Massachusetts Institute of Technology (February 1989).

Agre and Chapman have developed a theory of general activity. They argue that there are two kinds of planning, which can be referred to as capital-P Planning and small-p planning. They contend that much of AI research is on Planning, while what people actually do a lot more of is planning. This is similar to Zeltzer's discussion of motor planning as a subset of more general problem solving skills. Their work on Pengi (**Agre 1987**) is quite interesting because of their assertion that "we believe that combinatorial networks can form an adequate central system for most activity." It is also interesting because their chosen domain, the 2D game of Pengo, could be extended to 3D and implemented in bolio at some point.

(Agre 1987)

Agre, P. and Chapman, D. *Pengi: An Implementation of a Theory of Situated Action.* Proceedings of AAAI-87 (1987).

Wilson describes the *animat problem* in (**Wilson 1987**) which seems to agree well with the ethological approach Zeltzer has long advocated:

(Wilson 1987)

Wilson, S. *Classifier Systems and the Animat Problem*, Machine Learning, 2(3) (1987).

> *To survive in its environment, an animal must possess associations between environmental signals and actions that will lead to satisfaction of its needs. The animal is born with some associations, but the rest must be learned through experience. A similar*

situation might be said to hold for an autonomous robot (say on Mars or under the sea). One general way to represent the associations is by condition-action rules in which the conditions match aspects of the animal's environment and internal state and the actions modify the internal state or execute motor commands.

He describes a system using a classifier system (a variant of the Genetic Algorithm (**Goldberg 1989**)) to approach the problem of an animat in a 2D environment.

At Case Western, researchers are building a simulated insect, the Periplaneta Computatrix (**Beer 1989**). The design of the insect, and the nervous system that controls it, are inspired by the neuroethological literature on several natural animals.

In work directed toward constructing autonomous robots (**Maes 1989** and **1990A**), Maes has described the details of the connections among skills (competence modules in her terminology) for a "situated" agent. In her action selection network, each motor skill has a set of preconditions - the condition list - that must be true in order for the skill to execute. In addition, there is an add-list of propositions expected to become true once the skill has executed, and a delete-list of propositions that will no longer be true. Skills are interconnected through these preconditions, add- and delete-lists in the following ways: a skill S1, that, when executed, will make true the precondition for another skill S2 is called a predecessor node, and S1 may receive activation energy from S2. A skill S2 that has a precondition that will be made true by some other skill S1 is a successor of S1 and receives activation energy from S1. There are also conflicter relationships that correspond to inhibitory connections among nodes.

Importantly, Maes has introduced the notion of spreading activation, which provides for graded recruitment of motor resources—potentiation is not a binary switch, but a continuous quantity, so that a skill may be potentiated by varying amounts. This is also in agreement with the ethological account. The process of action selection takes into account the global goals of the agent, as well as the state of the world. Activation is spread to the skills from the goals and the state, and activation is taken away by the achieved goals which the system tries to protect. Activation is sent forward along the predecessor links, and backwards along the successor links; activation is decreased through the conflicter links, and each skill's activation is normalized such that the total activation energy in the system remains constant. If all the propositions in the condition list of a skill are satisfied in the current state of the world, and that skill's activation energy is higher than some global

(Goldberg 1989)

Goldberg, D.E. *Genetic Algorithms* Addison-Wesley (1989).

(Beer 1989)

Beer, R.D., Sterling, L.S., Chiel, H.J. *Periplanteta Computatrix: The Artificial Insect Project*, Tech Report TR 89-102, Case Western Reserve University (1989).

(Maes 1989)

Maes, P. *How to Do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

(Maes 1990A)

Maes, P. *Situated Agents Can Have Goals.* Journal of Robotics and Autonomous Systems 6,1&2 (1990).

threshold (as well as being higher than all the other modules in the network), that skill is invoked to perform its assigned action (thereby adding the propositions in its add list to the state and removing those on its delete list) and returns. If no skill is selected, the global threshold is reduced by some amount. Either way, the spreading of activation continues, as described above. The interested reader is referred to Chapter 3 where I present Maes' mathematical model of the theory and discuss some of my extensions to it.

Rod Brooks has argued that AI should shift to a process-based model of intelligent systems, with a decomposition based on "task achieving behaviors" as the organizational principle (**Brooks 1986**). He described a *subsumption architecture* based on the notion that later, more advanced layers *subsumed* earlier layers, in a sense simulating the evolutionary process biological organisms have undergone. He argues that AI would be better off "building the whole iguana", i.e. building complete systems, albeit simple ones, rather than some single portion of a more complex artificial creature (**Brooks 1989**). To this end, Brooks has spearheaded the construction of several successful (to varying degrees) mobile robots.

(Brooks 1986)

Brooks, R. A. *A Robust Layered Control System for a Mobile Robot.* IEEE Journal of Robotics and Automation 2,1 (1986).

(Brooks 1989)

Brooks, R. A. *The Whole Iguana.* Robotics Science. MIT Press (1989).

One example of a mobile robot based on the subsumption architecture was programmed by Maes to learn how to walk. The algorithm was similar to the one previously described by Maes (and the one implemented in this thesis) with the addition of simple statistically based learning (**Maes 1990B**). In the chosen domain (hexapod walking), the algorithm proved appropriate and accomplished its goal, although it is unclear how well it scales or transfers to other domains.

(Maes 1990B)

Maes, P. and Brooks, R. A. *Learning to Coordinate Behaviors*, Proceedings of AAAI-90, (1990).

On a more practical note, an example of a robotic insect is that of the robot bee, reported in the German journal Naturwissenschaften in June 1989. An interdisciplinary group of researchers led by a bioacoustician and an entomologist, have built and demonstrated a computerized bee that performs bee-dance steps well enough to convince other hive members to follow its directions. Researchers have successfully programmed the robot to dance in such a way as to tell the other bees it had found food 1,000 meters to the southwest. Upon seeing the robot dance, the other bees flew to that exact location. Reprogramming it to tell of food somewhere else causes the bees to fly to the new location. It is hoped that research in building virtual creatures and their associated behavior producing skills, will lead to useful robots such as the bee.

Ethologically-Based Control Ideas

Gallistel argues that action is organized in a hierarchical fashion, and gives examples drawn from a wide range of biological literature (**Gallistel 1980**). Zeltzer argues that a simple but general problem solving capacity, is innate in the organization of an agent's behavior repertoire when it is coupled with the appropriate representation of an object's functional and geometric attributes (**Zeltzer 1987**). Zeltzer and I propose the idea of a *skill network* (**Zeltzer 1990**), that shares many similarities with Brook's ideas of a process model of robot behavior, as well as Minsky's difference engine and parts of his *Society of Mind* theory. Many of these notions of lattice-like control of low-level behavior in animals were first proposed by Tinbergen in his seminal work in ethology (**Tinbergen 1951**).

Greene puts forth the notion of many "virtual arms" in any one real arm, i.e., the idea that, depending on the task, we treat our arm as a much simpler appendage than it is, finding and using recipes to solve the given task (**Greene 1988**). He states that additional degrees of freedom help, rather than hinder, the control process by "providing a variety of recipes to fake what we want to do" (**Greene 1972**). He also arguesthat object-oriented control might be used to model natural motion.

Turvey discusses a theory of the organization of action which draws heavily on the empirical studies of the Russian mathematician and behavioral biologist Nicholas Bernstein (**Turvey 1977** and **Bernstein 1967**). The major themes of Turvey's theory, as outlined in (**Gallistel 1980**), are as follows: the degrees of freedom problem, the idea that the DOF problem is solved by hierarchical command structure, that each level of the hierarchy is relatively autonomous with respect to other levels, and that higher units exert control over lower units by the parameters of the units themselves and parameters of the pathways by which the units interact.

(Gallistel 1980)

Gallistel, C. R. *The Organization of Action: A New Synthesis.* Lawrence Erlbaum Associates (1980).

(Zeltzer 1987)

Zeltzer, D. *Motor Problem Solving for Three Dimensional Computer Animation.* Proceedings of Proc. L'Imaginaire Numerique (May 14-16 1987).

(Zeltzer 1990)

Zeltzer, D. and Johnson, M.B. *Motor Planning: An Architecture for Specifying and Controlling the Behavior of Virtual Actors.* Journal of Visualization and Computer Animation, 2(2) (to appear).

(Tinbergen 1951)

Tinbergen, N. *The Study of Instinct.* Oxford University Press (1951).

(Greene 1988)

Greene, Peter H., *The Organization of Natural Movement,* Journal of Motor Behavior (June 1988).

(Greene 1972)

Greene, Peter H., *Problems of Organization of Motor Systems,* in Progress in Theoretical Biology (Vol. 2), Academic Press (1972).

(Bernstein 1967)

Bernstein, N. *The Coordination and Regulation of Movements.* Pergammon (1967).

(Turvey 1977)

Turvey, M.T. *Preliminiaries to a Theory of Action With Reference to Vision,* in Perceiving, Acting, and Knowing. Lawrence Erlbaum Associates (1977).

3 An Algorithm for Action Selection

L'embarras des richesses
(the more alternatives, the more difficult the choice)
Abbé D'Allainval

The notion of a using a network of interconnected motor skills to control the behavior of a virtual actor was first described by my advisor David Zeltzer in (**Zeltzer 1983**). This was later independently elaborated by Pattie Maes in (**Maes 1989**). Her algorithm was used as the starting point for the work done in this thesis. In addition to implementing her algorithm, I have extended the original in several ways, with an emphasis on the issues involved in a robust, parallel, distributed implementation that is not machine specific (i.e. portable to many different platforms). The implementation itself was quite challenging and brought to light some interesting issues in **MIMD** process synchronization, and will be covered in detail in the next chapter.

This chapter begins with an algorithm for the problem of action selection for an autonomous agent as presented by Maes in (**Maes 1989**), and then goes into some detail about extensions which have been made during the course of implementing it. The mathematical model presented here differs slightly from Maes' original in that it corrects one error I found while implementing it. The particular change is pointed out in a sidenote.

(Zeltzer 1983)

Zeltzer, D. Knowledge-Based *Animation.* Proceedings of ACM SIGGRAPH/SIGART Workshop on Motion (April 1983).

(Maes 1989)

Maes, P. *How to Do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

MIMD

Multiple Instruction, Multiple Data. Refers to a particular kind of parallel processing in which each process acting in parallel is operating on its own data in its own way.

Maes' view of agents

An agent is viewed as a collection of competence modules. Action selection is modeled as an emergent property of an activation/ inhibition dynamics among these modules. (**Maes 1989**)

Maes' Mathematical Model

This section of the paper presents a mathematical description of the algorithm so as to make reproduction of the results possible. Given:

- a set of competence modules $1..n$
- a set of propositions P
- a function $S(t)$ returning the propositions that are observed to be true at time t (the state of the environment as perceived by the agent); S being implemented by an independent process (or the real world)
- a function $G(t)$ returning the propositions that are a goal of the agent at time G; G being implemented by an independent process

- a function $R(t)$ returning the propositions that are a goal of the agent that has already been achieved at time t; R being implemented by an independent process (e.g. some internal or external goal creator)

- a function *executable(i, t)*, which returns 1 if competence module i is executable at time t (i.e., if all of the preconditions of competence module i are members of $S(t)$), and 0 otherwise

- a function $M(j)$, which returns the set of modules that match proposition j, i.e., the modules x for which $j \in c_x$

- a function $A(j)$, which returns the set of modules that achieve proposition j, i.e., the modules x for which $j \in a_x$

- a function $U(j)$, which returns the set of modules that undo proposition j, i.e., the modules x for which $j \in d_x$

- π, the mean level of activation

- θ, the threshold of activation, where θ is lowered 10% every time no module could be selected, and is reset to its initial value whenever a module becomes active

- ϕ, the amount of activation energy injected by the state per true proposition

- γ, the amount of activation energy injected by the goals per goal

- δ, the amount of activation energy taken away by the protected goals per protected goal

Given competence module $x = (c_x, a_x, d_x, \alpha_x)$, the input of activation to module x from the state at time t is:

$$input\_from\_state(x, t) = \sum_j \phi \frac{1}{\#M(j)} \frac{1}{\#c_x}$$

where $j \in S(t) \cap c_x$ and where # stands for the cardinality of a set.

The input of activation to competence module x from the goals at time t is:

$$input\_from\_goals(x, t) = \sum_j \gamma \frac{1}{\#A(j)} \frac{1}{\#a_x}$$

where $j \in G(t) \cap a_x$.

The removal of activation from competence module x by the goals that are protected at time t is:

$$taken\_away\_by\_protected\_goals(x, t) = \sum_j \delta \frac{1}{\#U(j)} \frac{1}{\#d_x}$$

where $j \in R(t) \cap d_x$.

The following equation specifies what a competence module $x = (c_x, a_x, d_x, \alpha_x)$, spreads backward to a competence module $y = (c_y, a_y, d_y, \alpha_y)$:

$$spreads\_bw(x, y, t) = \begin{cases} \sum_j \alpha_x (t-1) \frac{1}{\#A(j)} \frac{1}{\#a_y} & \text{if } executable(x, t) = 0 \\ 0 & \text{if } executable(x, t) = 1 \end{cases}$$

where $j \notin S(t) \wedge j \in c_x \cap a_y$.

The following equation specifies what module x spreads forward to module y:

$$spreads\_fw(x, y, t) = \begin{cases} \sum_j \alpha_x (t-1) \frac{\phi}{\gamma} \frac{1}{\#M(j)} \frac{1}{\#c_y} & \text{if } executable(x, t) = 1 \\ 0 & \text{if } executable(x, t) = 0 \end{cases}$$

where $j \notin S(t) \wedge j \in a_x \cap c_y$.

The following equation specifies what module x takes away from module y:

$$takes\_away(x, y, t) = \begin{cases} 0 & \text{if } (\alpha_x(t-1) < \alpha_y(t-1)) \wedge (\exists i \in S(t) \cap c_y \cap d_x) \\ \min\left(\sum_j \alpha_x (t-1) \frac{\delta}{\gamma} \frac{1}{\#U(j)} \frac{1}{\#d_y}, \alpha_y(t-1)\right) & \text{otherwise} \end{cases}$$

where $j \in c_x \cap d_y \cap S(t)$.

The activation level of a competence module y at time t is defined as:

$$\alpha(y, 0) = 0$$
$$\alpha(y, t) = decay(\alpha(y, t-1)(1 - active(y, t-1)) + input\_from\_state(y, t) + input\_from\_goals(y, t) - taken\_away\_by\_protected\_goals(y, t) + \sum_{xz} (spreads\_bw(x, y, t) + spreads\_fw(x, y, t) - takes\_away(z, y, t)))$$

sidenote

The use of the *min*() function here (as opposed to the *max*() function specified by Maes in (**Maes 1989**) is correct. This was discovered in the course of implementing this model, and verified with Maes.

(Maes 1989)

Maes, P. *How to Do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

where x ranges over the modules of the network, z ranges over the modules of the network minus the module y, $t > 0$, and the decay function is such that the global activation remains constant:

$$\sum_{y} \alpha_y(t) = n\pi$$

The competence module that becomes active at time t is module i such that:

$$active(t, i) = 1 \text{ if } \begin{cases} \alpha(i, t) >= \theta & (1) \\ executable(i, t) = 1 & (2) \\ \forall j \text{ fulfilling}(1) \wedge (2) : \alpha(i, t) >= \alpha(i, t) & (3) \end{cases}$$

$$active(t, i) = 0 \text{ otherwise}$$

Maes' Algorithm: Pros and Cons

The Good News

Maes' algorithm is notable on several accounts. First of all, without reference to any ethological theories, she captured many of the important concepts described in the classical studies of animals behavior. Her view of activation and inhibition, especially as a continuously varying signal, are in step with both classical and current theories of animal behavior (**Sherrington 1929** and **McFarland 1975**). Secondly, the algorithm can lend itself to a very efficient implementation, and allows for a tight interaction loop between the agent and its environment, making it suitable for real robots and virtual ones that could be interacted with in real time.

(Sherington 1906)

Sherrington, C.S. *The Integrative Action of the Nervous System.* Yale University Press (1906).

(McFarland 1975)

McFarland, D.J., Sibly, R.M. *The Behavioral Final Common Path.* Phil. Trans. Roy. Soc. London, 270:265-293 (1975).

Her enumeration of how and in what amount activation flows between modules is refreshingly precise:

> *...the internal spreading of activation should have the same semantics/effects as the input/output by the state and goals. The ratios of input from the state versus input from the goals versus output by the protected goals are the same as the ratios of input from predecessors versus input from successors versus output by modules with which a module conflicts. Intuitively, we want to view preconditions that are not yet true as subgoals, effects that are about to be true as 'predictions', and preconditions that are true as protected subgoals.* (**Maes 1989**)

(Maes 1989)

Maes, P. *How To do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

This correspondence gives her theory an elegance which stands head and

shoulders above the tradition of hacks, heuristics, and kludges that AI is littered with.

The Bad News

As with any new and developing theory, Maes' currently suffers from several drawbacks. I'll first list what I feel to be the problems with the algorithm as stated above, and then discuss each in turn.

- the lack of variables
- the fact that loops can occur in the action selection process
- the selection of the appropriate global parameters (θ, ϕ, γ, δ) to achieve a specific task is an open question
- the contradiction that "no 'bureaucratic' competence modules are necessary (i.e. modules whose only competence is determining which other modules should be activated or inhibited) nor do we need global forms of control" (**Maes 1989**) vs. efficiently implementing it as such
- the lack of a method of parallel skill execution

(Maes 1989)

Maes, P. *How to Do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

Discussion and Some Proposed Solutions

lack of variables

Maes asserts that many of the advantages of her algorithm would disappear if variables were introduced. She uses *indexical-functional aspects* to sidestep this problem, an approach I think is too limiting for anything more than toy networks built by hand, as any implementor would soon tire themselves of denoting every item of interest to a virtual actor in this way. Maes argues that the use of indexical-functional notation makes realistic assumptions about what a given autonomous agent can sense in its environment. This is perhaps true in the physical world of real robots, but in the virtual worlds I am concerned with, this is much less an issue. Either way, the addition of the option of using variables can only enhance the algorithm, although perhaps at the cost of some performance.

Indexical-functional aspects

This term was introduced to the AI community by Agre and Chapman (**Agre 1987**), and refers to the idea that an agent need only refer to things in relation to itself, as in "the-chair-near-me" or "the-cup-I-am-holding'.

(Agre 1987)

Agre, P. and Chapman, D. *Pengi: An Implementation of a Theory of Situated Action.* Proceedings of AAAI-87 (1987).

Variables could be introduced into the algorithm with the addition of a sort of generic competence module, which I'll call a *template agent.* These template agents would be members of the action selection network similar to competence modules, except they do not send or receive activation. When a fully specified proposition is entered in G(t), relating to the template agent, it would instance itself with all of its slots filled in. For example, a generic

competence module `walk-to X` might have on its add-list the proposition `actor-at-X`, where `X` was some location to be specified later. If the proposition `actor-at-red-chair` became a member of G(t), this would cause the template agent `walk-to X` to instance itself as a competence module `walk-to-red-chair` with, among other things, the proposition `actor-at-red-chair` on its add-list. This instanced competence module would then participate in the flow of activation just like any other competence module. When the goal was satisfied, or when it was removed from G(t), the competence module could be deleted, to be reinvoked by the template agent later if needed. If the number of modules in a given network was not an issue, any instanced modules could stay around even after the proposition which invoked them disappeared.

Loops

The second major difficulty with the current algorithm is that loops can occur. From my perspective, this isn't necessarily a bad thing, since this sort of behavior is well documented in ethology, and could be used to model such behavior in a simulated animal. From the broader perspective of trying to formulate general theories of action selection, it remains a problem to be addressed. Maes suggests a second network, built using the same algorithm, but composed of modules whose corresponding competence lies in observing the behavior of a given network and manipulating certain global parameters (θ, ϕ, γ, δ) to effect change in that network's behavior. This is an idea much in the spirit of Minsky's B-brains (**Minsky 1987**), in which he outlines the notion of a B-brain that watches an A-brain, that, although it doesn't understand the internal workings of the A-brain, can effect changes to the A-brain. Minsky points out that this can be carried on indefinitely, with the addition of a C-brain, a D-brain, etc. Maes is currently investigating this (**Maes 1990C**). While this idea is interesting, it does seem to suffer from the phenomenon sometimes referred to as the *homunculus problem*, or the *meta-meta problem*. The basic idea is that any such system which has some sort of "watchdog system" constructed in the same fashion as itself, can be logically extended through infinite recursion ad infinitum. Given this, I think the use of some other algorithm, most notably a Genetic Algorithm (**Goldberg 1989**), would be more appropriate as the watchdog for a given action selection network.

(Minsky 1987)

Minsky, M. *The Society of Mind.* Simon and Schuster (1987).

(Maes 1990C)

Maes, M. personal communication, (1990).

(Goldberg 1989)

Goldberg, D.E. *Genetic Algorithms* Addison-Wesley (1989).

How to Select θ, ϕ, γ, δ

The selection of the global parameters of the action selection network is an open issue. To generate a given task achieving behavior, it is not clear how to select the appropriate parameters. From the perspective of a user wishing to direct the actions of a virtual actor, this is a grievous flaw which must be

addressed. A similar solution to the one proposed for the loops problem could be used, namely using another network to select appropriate values. Unfortunately, this doesn't really address the problem of accomplishing a specific task. One idea is to use any of several learning methods to allow the network to decide for itself appropriate parameters. Learning by example could be used to excellent effect here.

Another interesting notion which is applicable is to allow the network to have some memory of past situations it has been in before. If we allow it to somehow recognize a given situation ("I'm going to Aunt Millie's—I've done this before. Let's see: I get up, close all the windows, lock all the doors, get in the car, and walk down the street to her house."), we could allow the network to bias its actions towards what worked in that previous situation. If we allowed additional links between completence modules called *follower links*, we could activation to be sent between modules which naturally follow each others' invocation in a given behavioral context. This idea has similarities to Minsky's *K-lines* (**Minsky 1987**) and Schanks *scripts* and *plans* (**Schank 1977**), but is more flexible because it isn't an exact recipe—it's just one more factor in the network's action selection process. This allows continuous control over how much credence the network gives the follower links, in keeping with the continuous quality of the algorithm.

(Minsky 1987)

Minsky, M. *The Society of Mind.* Simon and Schuster (1987).

(Schank 1977)

Schank, R. and Abelson, R. *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Associates (1977).

Supposedly No Global Forms of Control

Maes considers her algorithm to describe a continuous system, both parallel and distributed, with no global forms of control. One of her stated goals in the development of this algorithm was to explore solutions to the problem of action selection in which:

> *no 'bureaucratic' competence modules are necessary (i.e., modules whose only competence is determining which other modules should be activated or inhibited) not do we need global forms of control.* (**Maes 1989**)

(Maes 1989)

Maes, P. *How to Do the Right Thing* A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

Unfortunately, by the use of coefficients on the activation flow which require global knowledge (i.e. every term which involves the cardinality of any set not completely local to a competence module), there is no way her stated goal can be achieved. Secondly, it seems that any implementation of the algorithm has to impose some form of synchronization of the activation flow through the network. These two problem are inextricably linked, as I'll discuss below.

Maes asserts that the algorithm is not as computationally complex as a tradi-

tional AI search, and that it does not suffer from combinatorial explosion (**Maes 1990A**). She also asserts that the algorithm is robust and exhibits graceful degradation of performance when any of its components fail. Unfortunately, any implementation which attempts to implement the robustness implied in the mathematical model begins to exhibit complexity of at least $O(N^2)$, since each module needs to send information to the process supplying the values for M(j), A(j), and U(j). Also, information concerning the cardinality of c_x, a_x, d_x, c_y, a_y, and d_y must also be available to calculate the activation flow. This implies either a global database or shared memory in which these values are stored, or direct communication among the competence modules and the processes managing G(t), S(t), and R(t). Either method implies some method of synchronizing the reading and writing of data. Unfortunately, Maes asserts that the process of activation flow is continuous, which implies that asynchronous behavior of the component modules of the network is acceptable, which it clearly is not.

(Maes 1990A)

Maes, P. *Situated Agents Can Have Goals.* Journal of Robotics and Autonomous Systems 6,1&2 (1990).

If we are to implement this algorithm in a distributed fashion, which is desirable to take advantage of the current availability of networked workstations, we need to choose between a shared database (containing data concerning the cardinality of M(j), A(j), U(j), c_x, a_x, d_x, c_y, a_y, and d_y) and direct communication among competence modules. If we are to assume a direct communication model, a given module would need to maintain a communication link to each other module that held pertinent information to it (i.e. would be returned by any of the functions M(j), A(j), U(j) or would be involved in the calculation of the cardinality of a_x, d_x, c_y, a_y, and d_y). Additionally, a module would need some way of being notified when a new module was added to the network, and have some way of establishing a communication link to that new module. In the limit, this implies that every module would need to maintain *n-1* communication links, where the network was composed of *n* modules. Although necessary values to calculate the spreading of activation could be gotten and cached by each agent, to implement the robustness implied in the mathematical model, we need to recalculate each assertion for each proposition for each agent every time step. This implies a communication bottleneck, and semi-formidable synchronization issues.

the implementation informs the theory

Since one of my stated goals in this thesis work concerns the efficient implementation of any action selection algorithm, I concern myself more than a theorist might with implementation issues.

Alternatively, if it was implemented by a shared database or global memory, each agent would need only a single connection to the shared database. Some process external to the agents could mangage the connection of new agents to the database and removal of agents which become disabled. This would allow the agents not to have to worry about the integrity of the other members of the network, and would reduce the complexity of the communication involved to O(n). Given that the an agent's accesses to the database are known (i.e. a given agent would need to access the database the same

number of times as any other agent in the network), synchronization could be handled by a simple round-robin scheme, where each agent's request was handled in turn. When an agent wished to add itself to a given action selection network, it would need to *register* itself with the shared database by giving it information about itself (i.e. the contents of its condition, add, and delete-list). This would allow the database to answer questions from other agents about M(j), A(j), U(j) and the cardinality of a_x, d_x, c_y, a_y, and d_y. Such a *registry* could also have a way of marking agents which didn't respond to its requests for updated information, and perhaps even have the ability to remove agents from the network which didn't respond or whose communication channels break down.

In either method, there needs to be some agreed upon method of synchronizing messages so that activation flow proceeds according to the algorithm, and that only one action is selected at a given time step. If we postulate some agency which dispatches which action is selected, we fly in the face of Maes' assertion of no global forms of control. Unfortunately, if we are to implement the algorithm in the distributed fashion described so far, I don't see any way around having such a *task fragment dispatcher*.

No Parallel Skill Execution

Another problem is the assumption built into the algorithm that no competence module takes very long to execute. This seems implicit in the fact that Maes does not seem to consider the lack of a method for having parallel executing modules as a problem. For my purposes, this is a serious problem, since without such a capability, I could never have a virtual actor that could walk and chew gum at the same time. More specifically, a network containing a "walk" competence module and a "chew-gum" module could never have both of them executing in parallel. Maes' view of the granularity of time in the system is very fine, while my view is that there should be some parameter which allows control from fine to coarse.

4 The Skill Network

An Implementation of an Action Selection Network

*This isn't a blue sky outfit, you know. We **build** things around here.*
Andy Lippman

Overview

I have developed a system that implements the algorithm for action selection networks as discussed in the last chapter, including many of the proposed extensions. This particular sort of action selection network is what my advisor David Zeltzer and I call a *skill network*, since it consists of a set of motor skills corresponding to the simulated skills of a virtual actor. The implementation is written in C, and runs on a variety of vendors' UNIX workstations.

The Skill Network

The motor skills of the skill network have been implemented as a distributed set of UNIX processes, referred to as *skill agents*. These skill agents are quite similar to Maes' competence modules. The perceptions of the virtual actor are handled by another set of UNIX processes, referred to as *sensor agents*. The goals of the virtual actor are handled by another set of processes known as *goal agents*. The interconnections among all of these agents is handled by yet another UNIX process called the *registry/dispatcher*. The *registry/dispatcher* is the nexus of information flow among all the agents composing the virtual actor's skill network. It acts as both a shared database and a message router, maintaining information about each agent and directing the flow of activation energy among them. Since a given task (i.e., "open-the-door") may entail the execution of many different motor skills, I view the registry/dispatcher as less a task *manager* than a task *fragment dispatcher*, hence the latter part of its name. The goal, sensor, and skill agents can run on any machine on a network, and connect to the registry/dispatcher via the *action selection network (asn) daemon*, that is a UNIX process that listens for messages at a known network address, and maintains a connection to the registry/dispatcher.

agent vs. dude

The reader will recall that I use the term *agent* to refer to one of the component processes, and the collective entity composed of these agents and their interconnections is referred to as a *dude* or *virtual actor*.

Implementation Design Considerations and Motivations

The implementation of the skill network was designed around several important real-world considerations:

- It should be portable to different vendors' workstations.
- Computation should be distributed over a network of workstations in or-

der to execute efficiently.

- The skill network must be robust, such that if a skill becomes disabled during execution, the system should find an efficient workaround, if one exists, and avoid selecting that skill.

- The skill network should be opportunistic, i.e. if a better skill is added during execution, the system should allow for that skill to be selected.

- The implementation should allow efficient experimentation with large systems, easy experimentation with different interconnections among skills, and efficient experimentation with multiple actors in a shared virtual environment.

Agents Communicate via the Registry/Dispatcher

When distributing the computation involved in the skill network over a network of machines, it becomes important to minimize the communication among its various parts. I chose not to implement the skill network as a set of agents with many direct interconnections between them because of the need for global knowledge (as discussed in **Chapter 3**), shared among the agents. Since this would have involved broadcasts among all the agents several times a time step, I instead chose to centralize the communication among the component processes of the skill network (the agents) in one process—the registry/dispatcher. The registry/dispatcher maintains a shared database among all the agents. While the notion of such a shared database may seem to present a bottleneck, one finds in practice it is very efficient (also discussed in **Chapter 3**). The registry/dispatcher alleviates much of the message traffic that would have to occur if it were not present.

When the agents first communicate with the registry/dispatcher, they *register* themselves by giving the registry/dispatcher enough information to enable it to act as a selective message router, spreading activation through a locally maintained version of the skill network, and only passing messages on to agents when absolutely necessary. In order to do this effectively, the agents must supply the registry/dispatcher with enough information to construct a complete skill network. The following section details the information an agent must supply to the registry/dispatcher.

Representation of Agents in the Registry/Dispatcher

Sensor Agents

Since the skill network maintains no explicit world model, all of its data about its relationship to the outside world (albeit a virtual one), comes to the actor via its sensor agents in the form of signs and signals. Sensor agents measure signs and signals using what I call *proposition-value pairs* (pv-pair).

A *pv-pair* consists of exactly that: a proposition (i.e., a-door-is-nearby, ambient-temperature-is) and a value (i.e. TRUE, 72.0). Pv-pairs, in fact, represent the **signs** and **signals** to which the virtual actor attends. For our purposes, this is a more useful atomic unit of information than just allowing a proposition to be either true or false, and allows for continuous quantities (distance, color, etc.) to be measured in discrete (but greater than binary) units, without necessitating a separate proposition for each case.

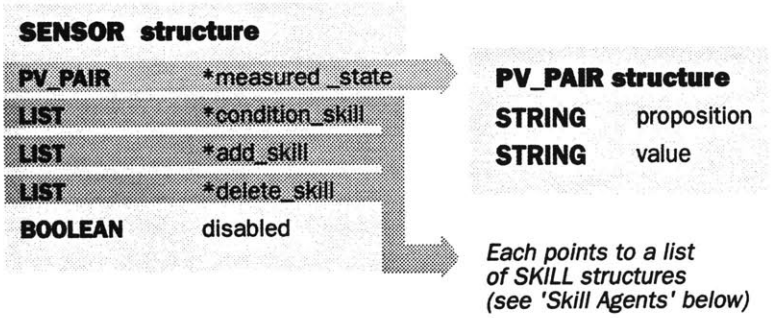
signs & signals

Signals represent sensor data—e.g. heat, pressure, light—that can be processed as continuous variables. Signs are facts and features of the environment or the organism. (**Rasmussen 1983**)

In addition to the pv-pair that the sensor agent measures, the registry/dispatcher maintains three lists for each sensor agent. The first list has pointers to all the skill agents that have on their condition-list (see **Skill Agents** below) the pv-pair which this sensor agent measures. The second list has pointers to all the skill agents that have on their add-list the pv-pair which this sensor agent measures, and the third list has pointers to all the skill agents with that pv-pair on their delete-list.

(Rasmussen 1983)

Rasmussen, J. *Skills, Rules and Knowledge; Signals, Signs and Symbols and other Distinctions in Human Performance Models* IEEE Trans. on Systems, Man, and Cybernetics, vol. SMC-13, no. 3 (May/June 1983).



Finally, the registry/dispatcher maintains a boolean flag as to whether or not it considers this agent **disabled**. The registry/dispatcher considers an agent disabled when the agent has been sent a message and has ignored it. Since this might not be the agent's fault (i.e., the network connection between the registry/dispatcher and the agent might be temporarily hung, or the agent might be busy doing something else momentarily), it doesn't make sense for the registry/dispatcher to sever its connection with the agent (thereby removing it from the skill network). Using the same reasoning, it doesn't make sense for the registry/dispatcher to waste resources spreading activation to/from an agent which is not currently available (see **Activation Flow in the Skill Network** below).

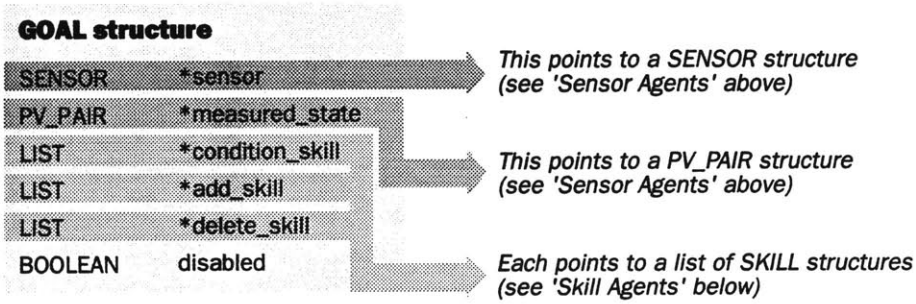
disabled agents

An agent which has been disabled for some time and still does not respond is removed from the skill network by the registry/dispatcher. This process is said to ***amputate*** the agent from the network.

Goal Agents

Goal agents are represented in the registry/dispatcher as a desired state, i.e. a pv-pair that the goal agent desires to be true. For each goal agent, the registry/dispatcher also maintains a pointer to the sensor agent that measures the pv-pair the goal agent desires satisfied. Note that if the virtual actor has some goal for which it does not have a corresponding sensor agent (i.e., the pointer is NULL), it has no way of ever knowing if that goal has been satis-

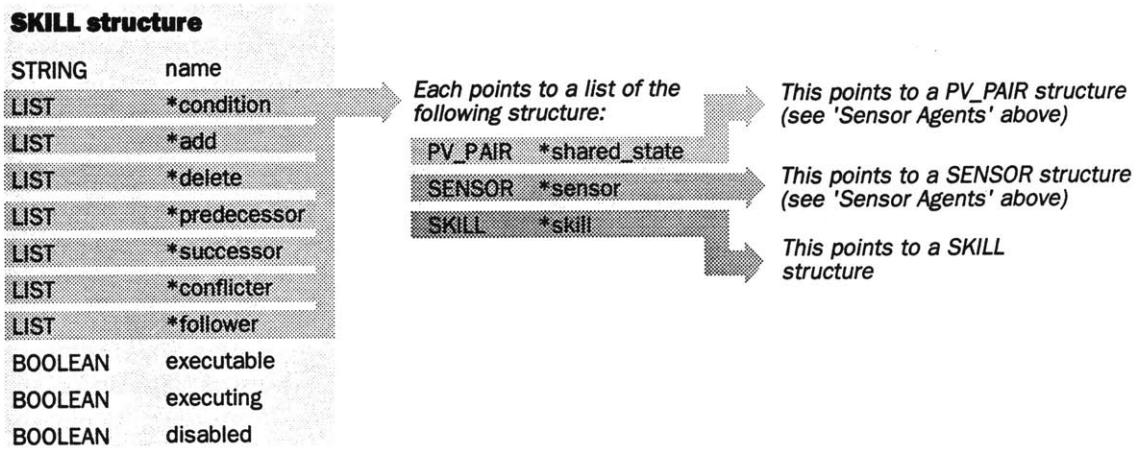
fied. The registry/dispatcher also maintains three lists for each goal agent, similar to the ones it has for each sensor agent. The first list has pointers to all the skill agents that have on their condition-list the pv-pair that this goal agent wishes satisfied. The second list has pointers to all the skill agents that have on their add-list the pv-pair this goal agent wishes satisfied, and the third list has pointers to all the skill agents with that pv-pair on their delete-list.



Skill Agents

Skill agents are represented in the registry/dispatcher by their name, a set of preconditions necessary for the skill to execute, and a set of predictions that the skill makes about the state of the world when the skill is finished executing. The preconditions are implemented by a list of structures, called the *condition-list*. The predictions are represented as two lists: the *add-list*, which is a list of structures containing, among other things, a pointer to a pv-pair that will be made true, and the *delete-list*, which is a list of structures containing, among other things, a pointer to a pv-pair that will be made false. Note that these two lists are only predictions—if the corresponding sensor agents measure something differently the next time step, that is what the virtual actor will believe. Each of the aforementioned structures contains three pointers: a pointer to a pv-pair, a pointer to the skill involved in the relation, a pointer to the sensor that measures that pv-pair.

Also, the registry/dispatcher maintains four other lists for each skill agent: *successors, predecessors, conflicters, followers*. The *successors* list has pointers to all the skill agents that have on their condition-list a pv-pair that is on this skill agent's add-list The *predecessors* list has pointers to all the skill agents that have on their add-list a pv-pair that is on this skill agent's condition-list. The *conflicters* list has pointers to all the skill agents that have on their delete-list a pv-pair that is on this skill agent's condition-list. The *followers* list has pointers to all the skill agents that, in a particular context, have been known to succeed the invocation of this skill agent.



For each skill agent, the registry/dispatcher also maintains three boolean flags. The first corresponds to whether or not the skill agent is considered *executable*. A skill agent is considered executable if all of the pv-pairs in its condition-list match the pv-pairs currently measured by the corresponding sensor agents. The second flag marks whether or not a skill agent is currently *executing*. The third flag marks whether or not the registry/dispatcher considers this agent *disabled*. This is the same as the disabled flag for the other two types of agents.

The Registration Process:

How an agent connects to a skill network

Agents connect to a registry/dispatcher by sending a message to an asn daemon, a UNIX process that is listening for connections at some known network address. The agent connects to the asn daemon and sends it a message requesting registration information (what host on the network is the registry/dispatcher running on, what port should it attempt to connect on) about a particular registry/dispatcher. The asn daemon, which maintains a list of registry/dispatchers that it has connections with, checks that list for the requested registry/dispatcher. If the daemon has a valid connection to that registry/dispatcher, it sends it a message saying that some agent wishes to register with it. The registry/dispatcher then sends a message back to the daemon with the necessary information, which the daemon then sends back to the agent. The agent then breaks its connection to the daemon and connects to the registry/dispatcher. If the daemon doesn't know about the requested registry/dispatcher, or its connection to that registry/dispatcher is no longer valid (perhaps the registry/dispatcher process died, or the network connection between the two machines has gone down), it knows how to start a new one up, either locally (on the same machine as the daemon) or remotely (on some other machine on the network). The daemon subsequently starts up a registry/dis-

patcher with enough information for the registry/dispatcher to call the daemon back once it has started up. At this point, the daemon sends it a message telling it about the agent that wishes to register with it, and things proceed as described above.

Once the agent connects to the registry/dispatcher, it registers itself in the skill network by sending the registry/dispatcher a message containing all of the information the registry/dispatcher needs to add this agent to the skill network (see **Sensor Agents, Goal Agents**, and **Skill Agents** above). After an agent has registered, the registry/dispatcher updates its database of connections between the agents, constructing all of the lists in the agents' structures to reflect the implicit relationships among all the currently registered agents connected to this registry/dispatcher. The registry/dispatcher updates the skill network each time a new agent registers, or a known agent is *amputated* from it (see **Sensor Agents** above).

Unregistered Agents

To the registry/dispatcher, any process which satisfies the following is considered an agent:

- Sends a "request for registration info" via the asn daemon to the registry/dispatcher,
- subsequently connects to the socket at the port number the registry/dispatcher has allocated because of the "request for registration info" request.

This leads to agents that request registration information, connect to the registry/dispatcher, and subsequently never register. In practice, this seemingly anomolous behavior is very useful. *Unregistered agents* (as such agents are called) can send messages to the registry dispatcher that are evaluated at a higher priority than registered agents (See **Appendix A** for more details). This allows the user (or other processes) to connect to a given skill network and send messages to other agents or the registry/dispatcher. More exactly, this capability is used by the asna (see below) to influence activity in the skill network. This influence is used when activation flow is calculated and when the value of global parameters in a given skill network, as well as getting information for the user (or other programs) about a particular agent. In the examples given in **Chapter 5**, this is how the flow of activation through each skill network was controlled.

tcl

tcl is the t*ool command language*, a simple programming language which can be linked into application programs. I used it for all the interpreters in all the applications in Build-a-Dude. See **(Ousterhout 1990)** for more information.

(Ousterhout 1990)

Ousterhout, J.K. *Tcl: an Embeddable Command Language*, Winter Usenix Conference Proceedings (1990).

Agents are Managed by the asna

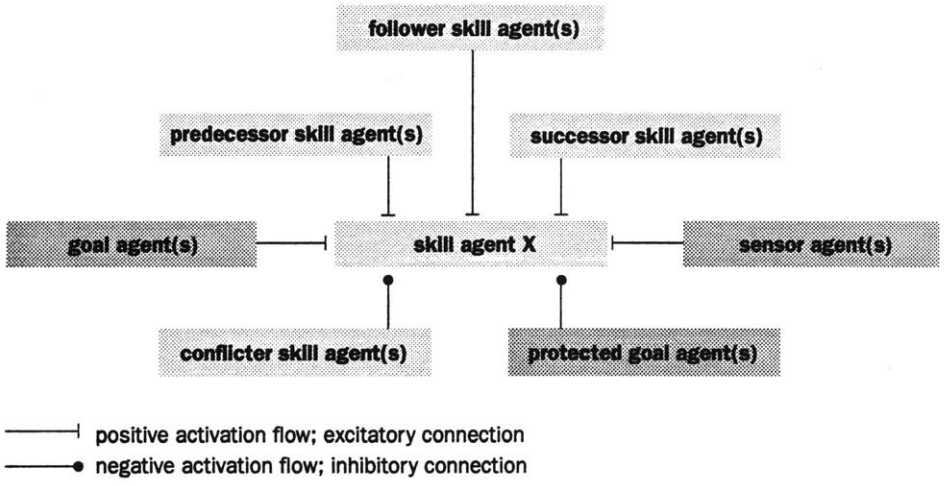
Agents are managed by a program called the *asna* (the action selection network agent manager). The asna is a **tcl** based interpreter that, among other

things, maintains a list of agent structures. These agents can be allocated and deallocated dynamically, and each agent can be connected to an arbitrary registry/dispatcher. Every time step, the asna polls the keyboard for user input, and checks for new messages for any currently active agents. Each agent maintains its own augmented tcl interpreter. Any messages received by the asna for a particular agent are passed onto that agent for execution by its own interpreter. The user has access to a history mechanism that maintains all messages received or sent by the asna. Using the readline package from the Free Software Foundation, the user can call up any previous commands and edit them, as well as having access to command and file name completion. The user can also selectively disable any agent so that it ignores messages from the registry/dispatcher. This is especially useful for constructing scenarios to test the robustness of the registry/dispatcher's failure mechanisms.

Activation Flow in the Skill Network

Once an agent has registered with the registry/dispatcher, it participates in the spread of activation in the skill network. Activation is spread through the network from seven different sources, all flowing to the individual skill agents. A sensor agent sends activation to each skill agent that has on its condition-list the pv-pair that sensor agent measures. A goal agent has a pv-pair that needs to be satisfied, and it sends activation to each skill that has this pv-pair on its add-list. Goal agents can also have protected goals which should remain true once achieved. For each protected goal, the associated goal agent sends negative activation, or inhibition, to each skill agent that has that goal on its delete-list.

An executable skill spreads activation forward to each of its successors for which their shared pv-pair is currently not measured to be true. A non-executable skill spreads activation backwards to each of its predecessors for which their shared pv-pair is currently not measured to be true. Each skill spreads inhibition to each of its conflicters for which the shared pv-pair is true. Each skill spreads activation forward to each of the members of its follower list.



Note that no activation flows to or from any skill that the registry/dispatcher has marked as disabled (see **Sensor Agents** above). The registry/dispatcher periodically checks disabled agents to see if they have become enabled again.

A Virtual Actor's World: Skill Agents, Sensor Agents, and an IGSP

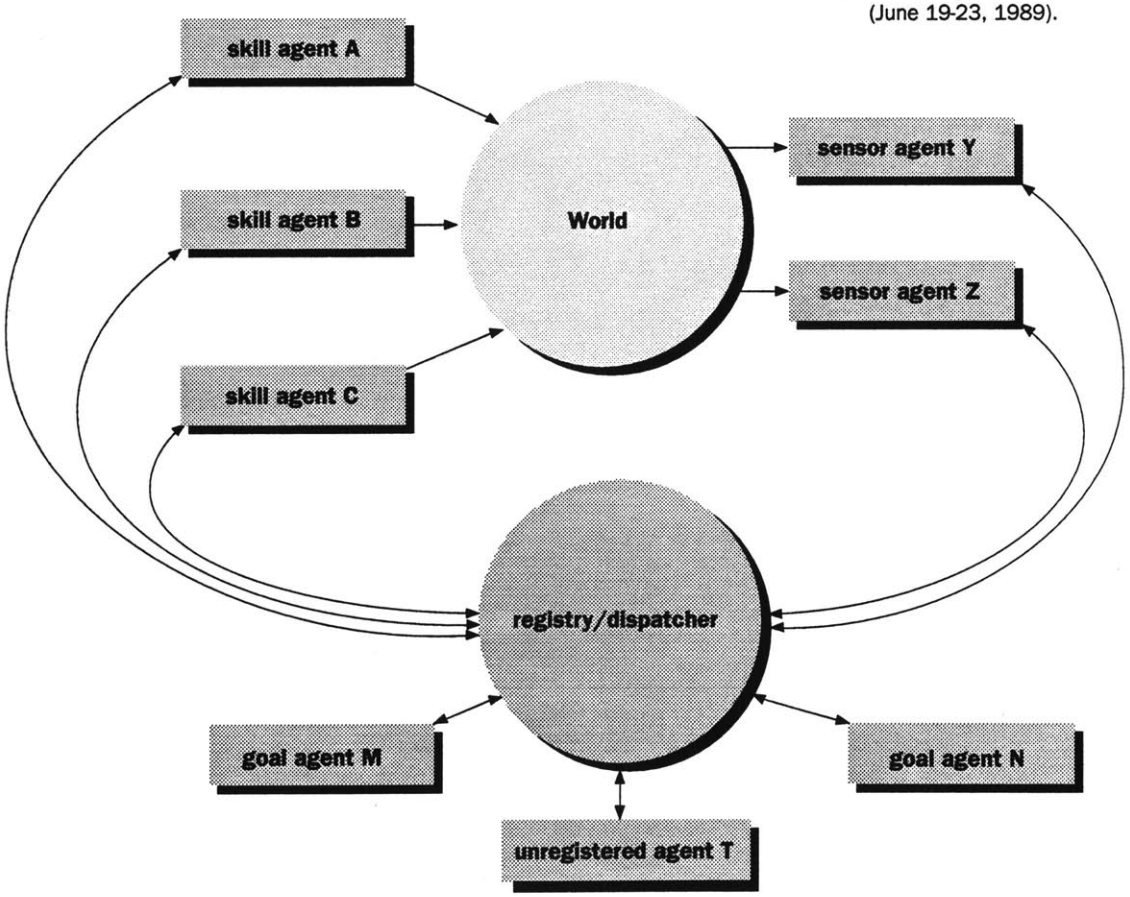
In addition to maintaining connections to the registry/dispatcher, sensor and skill agents are also connected to an integrated graphical simulation platform (IGSP) which acts as the virtual actor's world. The IGSP sends information to the sensor agents whenever the event that the sensor agent measures changes value. The skill agents send information to the IGSP whenever they wish to change the properties (position, angle, orientation, etc.) of the graphical simulation the skill agent is partially controlling in the IGSP. Another way to look at it is that each agent has a one-way connection to the IGSP: information flows to the IGSP from the skills, and from the IGSP to the sensors.

IGSP

An integrated graphical simulation platform called bolio is the software testbed we use here in the Computer Graphics & Animation Group for our virtual environment work. (**Zeltzer 1989**)

(Zeltzer 1989)

Zeltzer, D., S. Pieper and D. Sturman. *An Integrated Graphical Simulation Platform.* Proceedings of Proc. Graphics Interface '89 (June 19-23, 1989).



5 Results
A Benchmark and Some Detailed Examples

Do I contradict myself? Very well then, I contradict myself,
(I am large, I contain multitudes).

Walt Whitman

Overview

The work performed for this thesis led to the development of a theory of action selection as outlined and explained in the previous chapters. Using this theory, I designed and built a set of software tools, which I refer to interchangeably as the *ASN apps*, the *Build-a-Dude system*, or simply *Build-a-Dude*. This chapter chronicles some of the example scenarios I have experimented with using these tools. Since Build-a-Dude is an evolving system, these results represent a snapshot of the system's capabilities as of December 1990. Included in each section is a brief discussion of what this particular example is intended to show, a synopsis of what happened when it was run through the Build-a-Dude system, and a discussion of those results.

Maes' Robot Sprayer/Sander — a B-a-D benchmark

In Maes' paper (**Maes 1989**) which outlines the action selection algorithm upon which mine is based, she gives a detailed example using a robotic spray painter from Charniak & McDermott's AI book (**Charniak 1985**) to illustrate the algorithm's operation. Since I had access to both Maes' paper and her code, I chose this example as my benchmark to test that my implementation was complete. Since I was using Maes' algorithm as my starting point, I needed to make sure that I had implemented the capability embodied in her system before I could extend the algorithm in any way. This example seemed particularly good since it shows off Maes' original algorithm and provides a reasonably rigorous test case.

(Maes 1989)

Maes, P. *How to Do the Right Thing*, A.I. Memo 1180. Massachusetts Institute of Technology (December 1989).

(Charniak 1985)

Charniak, E. and McDermott, D. *Introduction to Artificial Intelligence*, Addison-Wesley (1985).

Scenario: Robot Sprayer/Sander

In this example, a two-handed robot is faced with the task of spray painting itself and sanding a board. It needs to be relatively smart about performing the task. It must either use both hands or a vise that is available to it, and it also must sand the board first, since spray painting itself would render it inoperable.

The first problem I faced was how to phrase the problem equivalently in my

system, since mine was a unique implementation of Maes' algorithm. Maes's system read a file of LISP code consisting of a list of initial goals, a set of propositions relating to the initially measured state of the environment, and a definition of some set of competence modules.

```
G(0)  =  (board-sanded,  self-painted)

S(0)  =  (hand-is-empty,  hand-is-empty,  sander-somewhere,  sprayer-somewhere,  operational,
          board-somewhere)
```

```
(defmodule  PICK-UP-SPRAYER
       :condition-list  '(sprayer-somewhere  hand-is-empty)
      :add-list  '(sprayer-in-hand)
       :delete-list  '(sprayer-somewhere  hand-is-empty))
(defmodule  PICK-UP-SANDER
       :condition-list  '(sander-somewhere  hand-is-empty)
      :add-list  '(sander-in-hand)
       :delete-list  '(sander-somewhere  hand-is-empty))
(defmodule  PICK-UP-BOARD
       :condition-list  '(board-somewhere  hand-is-empty)
      :add-list  '(board-in-hand)
       :delete-list  '(board-somewhere  hand-is-empty))
(defmodule  PUT-DOWN-SPRAYER
       :condition-list  '(sprayer-in-hand)
       :add-list  '(sprayer-somewhere  hand-is-empty)
      :delete-list  '(sprayer-in-hand))
(defmodule  PUT-DOWN-SANDER
       :condition-list  '(sander-in-hand)
       :add-list  '(sander-somewhere  hand-is-empty)
      :delete-list  '(sander-in-hand))
(defmodule  PUT-DOWN-BOARD
       :condition-list  '(board-in-hand)
       :add-list  '(board-somewhere  hand-is-empty)
      :delete-list  '(board-in-hand))
(defmodule  SAND-BOARD-IN-HAND
       :condition-list  '(operational  board-in-hand  sander-in-hand)
      :add-list  '(board-sanded)
     :delete-list  '())
(defmodule  SAND-BOARD-IN-VISE
       :condition-list  '(operational  board-in-vise  sander-in-hand)
     :add-list  '(board-sanded)
     :delete-list  '())
(defmodule  SPRAY-PAINT-SELF
       :condition-list  '(operational  sprayer-in-hand)
      :add-list  '(self-painted)
      :delete-list  '(operational))
(defmodule  PLACE-BOARD-IN-VISE
       :condition-list  '(board-in-hand)
       :add-list  '(hand-is-empty  board-in-vise)
       :delete-list  '(board-in-hand))
```

Build-a-Dude, on the other hand, required the definition of some set of skill agents, a set of goal agents, and a set of sensor agents (any of which could be ommitted) as a set of **tcl** commands. Also, Maes' system read in LISP code and then printed out what was happening as the activation spread through the system. Build-a-Dude was composed of a set of interacting programs,

note to non Lisp hackers

defmodule is a macro which places its arguments in a "competence module" structure in Maes' system. The :xxx construct denotes a structure member.

For example, the definition for the module **SAND-BOARD-IN-VISE** has a condition list with the following propositions: **operational**, **board-in-vise**, **sander-in-hand**. Its add-list contains the single proposition **board-sanded**, and its delete list is empty.

tcl

tcl is the t*ool command language*, a simple programming language which can be linked into application programs. I used it for all the interpreters in all the applications in Build-a-Dude. See **(Ousterhout 1990)** for more information.

(Ousterhout 1990)

Ousterhout, J.K. *Tcl: an Embeddable Command Language*, Winter Usenix Conference Proceedings (1990).

some of which were interactive with the user, and some of which were controlled automatically by other programs. Each of the ASN apps involved in Build-a-Dude maintained a log file on disk to which it wrote a record of every interesting action it took. This allowed a user to see what the various programs were doing, both during execution and as a record for later perusal. Agents were controlled by a program called *asna*, the action selection network agent manager program. A skill network could have an arbitrary number of asnas involved. An asna process reads tcl commands, that have some of the same flavor of LISP, except that tcl uses { } to enclose lists while LISP uses ( ). The following shows the robot sprayer/sander expressed in Build-a-Dude as a tcl **proc**:

```
proc dtrt-example {host port registry} {

   ASNA-become-goal $host $port $registry board-sanded T
   ASNA-become-goal $host $port $registry self-painted T

   ASNA-become-sensor $host $port $registry sprayer-somewhere T
   ASNA-become-sensor $host $port $registry sprayer-in-hand F
   ASNA-become-sensor $host $port $registry sander-somewhere T
   ASNA-become-sensor $host $port $registry sander-in-hand F
   ASNA-become-sensor $host $port $registry board-somewhere T
   ASNA-become-sensor $host $port $registry board-in-hand F
   ASNA-become-sensor $host $port $registry hand-is-empty T
   ASNA-become-sensor $host $port $registry operational T
   ASNA-become-sensor $host $port $registry self-painted F
   ASNA-become-sensor $host $port $registry board-sanded F
   ASNA-become-sensor $host $port $registry board-in-vise F

   ASNA-become-skill $host $port $registry pick-up-sprayer
      {{{sprayer-somewhere T} {hand-is-empty T}}}
     {{{sprayer-in-hand T}}}
      {{{sprayer-somewhere F} {hand-is-empty F}}}
   ASNA-become-skill $host $port $registry pick-up-sander
      {{{sander-somewhere T} {hand-is-empty T}}}
     {{{sander-in-hand T}}}
      {{{sander-somewhere F} {hand-is-empty F}}}
   ASNA-become-skill $host $port $registry pick-up-board
      {{{board-somewhere T} {hand-is-empty T}}}
     {{{board-in-hand T}}}
      {{{board-somewhere F} {hand-is-empty F}}}
   ASNA-become-skill $host $port $registry put-down-sprayer
     {{{sprayer-in-hand T}}}
      {{{sprayer-somewhere T} {hand-is-empty T}}}
     {{{sprayer-in-hand F}}}
   ASNA-become-skill $host $port $registry put-down-sander
     {{{sander-in-hand T}}}
      {{{sander-somewhere T} {hand-is-empty T}}}
     {{{sander-in-hand F}}}
   ASNA-become-skill $host $port $registry put-down-board
     {{{board-in-hand T}}}
      {{{board-somewhere T} {hand-is-empty T}}}
     {{{board-in-hand F}}}
   ASNA-become-skill $host $port $registry sand-board-in-hand
      {{{operational T} {board-in-hand T} {sander-in-hand T}}}
     {{{board-sanded T}}}
    {{{}}}
```

note to non tcl hackers

proc in tcl is a routine that defines a new user callable function. The name of the new function is the first argument, and the variables which follow can be used in the function definition by prepending them with a $.

note to non asna users

ASNA-become-goal is a function that attempts to connect to a registry/dispatcher named `$registry` as a goal agent, giving the proposition value pair that it desires. It calls up an asn daemon listening at port `$port` on host `$host`.

ASNA-become-sensor is a function that attempts to connect to a registry/dispatcher named `$registry` as a sensor agent, giving the proposition value pair that it measures. It calls up an asn daemon listening at port `$port` on host `$host`.

ASNA-become-skill is a function that attempts to connect to a registry/dispatcher named `$registry` as a skill agent, giving the proposition value pairs in its condition-list, add-list, and delete-list respectively. It calls up an asn daemon listening at port `$port` on host `$host`.

```
  ASNA-become-skill $host $port $registry sand-board-in-vise
      {{{operational T} {board-in-vise T} {sander-in-hand T}}}
     {{{board-sanded T}}}
    {{{}}}
   ASNA-become-skill $host $port $registry spray-paint-self
     {{{operational T} {sprayer-in-hand T}}}
     {{{self-painted T}}}
     {{{operational F}}}
   ASNA-become-skill $host $port $registry place-board-in-vise
     {{{board-in-hand T}}}
      {{{hand-is-empty T} {board-in-vise T}}}
     {{{board-in-hand F}}}
}
```

Since I was trying to reproduce the results Maes had obtained, I set the global parameters that control the activation flow in the network the same:

- influence from goals, $\gamma = 70.0$
- influence from state, $\phi = 20.0$
- influence from achieved goals, $\delta = 50.0$
- mean activation level, $\pi = 20.0$
- threshold for action selection, $\theta = 45.0$

sidenote

See Chapter 3 for a detailed discussion of these global parameters.

Synopsis: Robot Sprayer/Sander

After starting up an asn daemon, I started asna, the action selection network agent manager program. I then sourced the file containing the above tcl proc `dtrt-example`. Since I had previously started up the daemon on the host archy listening at port 9500, I invoked the proc with the following arguments:

sidenote

For the reader's sake, these same values for the global parameters will be used for all examples discussed in this chapter.

```
dtrt-example archy 9500 dtrt
```

The first agent that started up, the goal agent `board-sanded`, caused the asn daemon to start up a registry/dispatcher called `dtrt` somewhere on the net (in this case, on the host archy). Once the registry/dispatcher started up, it began accepting connections from agents, registering each one in turn, until it successfully registered all 23 agents. At this point, no activation was flowing through the skill network. The registry/dispatcher was in its inner loop (see **Appendix A**), constantly checking for messages from the asn daemon. In order to send it messages that it would evaluate outside of the activation spreading loop, I added one more agent, but didn't register it, so that the registry/dispatcher will continue to listen for messages from it:

Appendix A

Appendix A discusses in some detail the workings of the registry/dispatcher's inner loop. This is probably only of interest to the reader who wishes to implement the algorithm in a fashion similar to the way I have.

```
ASNA-become-unregistered-agent archy 9500 dtrt
```

From the asna command line, I then sent messages to the registry/dispatcher to inititiate the spreading of activation through the skill network. The results

exactly matched Maes' reported output, which I corroborated in more detail by running the example side by side using her LISP code vs. Build-a-Dude on the same platform.

Discussion: Robot Sprayer/Sander

It took about 4 days from start to finish to run this example to completion, including all bug fixes. Since it was the first example of any kind that I had run through the system, I was quite pleased. After those four days, Build-a-Dude could run the whole example in a few seconds, which was quite heartening, since no explicit optimization had been done at this point. As a very rough benchmark, it ran the first 10 steps of activation spreading (including printing out all pertinent comments to a file) in a second or two. For comparison, Maes' implementation, running on the same hardware platform (an HP-9000 835), took 15 times longer. This is not to denigrate Maes' implementation, rather to point out the efficiency of this one.

This did point out an interesting difference in philosophy, which I didn't discover to be a problem until much later, namely the fact that Maes used the existence of a proposition as the atomic unit defining the competence modules, while I used proposition value pairs for the definition of the agents in Build-a-Dude. At the time I felt this would lead to more expressiveness, but as we'll see later, was actually something of a dead end.

Another interesting note in hindsight: this example uncannily slipped just under the 25 internet domain socket limit (this example uses 24) as I discovered later (see **Discussion: Dude Take 1**).

Gallistel's cat walk cycle fragment

In (**Gallistel 1980**), Gallistel discusses the activation of two different reflexes during a cat's walk cycle:

> *The particular flexion and extension reflex that I have chosen for the present illustration have exactly the same adequate stimulus. Both of these reflexes are activated by a tap on the top or front of the animal's foot—the part ot the foot that is most likely to strike against something that threatens to trip the animal or sweep its foot out from under it. Flexion and extension reflexes with this common adequate stimulus have been demonstrated in the cat by* ***Forssberg, Grillner, and Rossignol (1975).*** *The flexion reflex, which has the effect of lifting the swinging leg higher off the ground, is seen during the "swing" phase (lift and advance phase) of the stepping cy-*

(Gallistel 1980)

Gallistel, C. R. *The Organization of Action: A New Synthesis.* Lawrence Erlbaum Associates (1980).

(Forssberg et al. 1975)

Forssberg, H.S., Grillner, S., and Rossignol, S. *Phase Dependent Reflex Reversal During Walking in Chronic Spinal Cats.* Brain Research, 85, (1975).

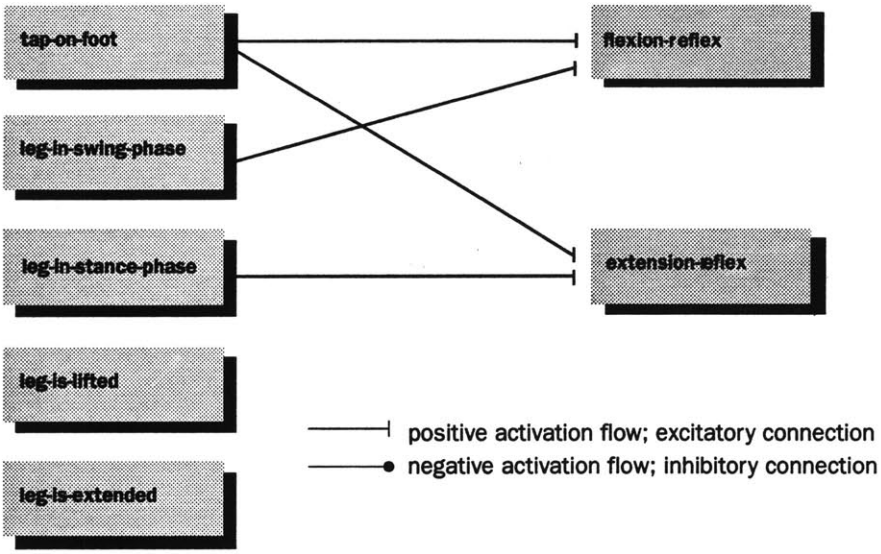
cle. If one taps the leading edge of the cat's paw as the cat swings its leg forward, the tap elicits flexion of the leg joints—the toe, the ankle, the knee, and the hip. In a movie made of this experiment, one can see that the flexion has the effect of lifting the swinging leg up and over a stick that would otherwise have arrested the swing and tripped the cat. The extension reflex, on the other hand, is seen during the stance phase of the stepping cycle when the leg supp-orts and propels the cat. If one taps the leading edge of the paw during this phase, the tap elicits extension of the leg joints. In the movie, one can see that this extension has the effect of hastening the completion of the stance phase, so that a moving object that would otherwise have swept the cat's foot out from under it does not do so.

Scenario: Cat Walk Cycle

This example is intended to demonstrate how the action selection algorithm could effectively model relatively low-level animal behavior in a way consistent with ethological theories. To model the portion of the cat behavior described by Gallistel, I wrote a tcl proc consisting of five sensor agents and two skill agents (as well as the ubiquitious unregistered agent which is used to send messages to the registry/dispatcher):

```
proc cat-step {port machine registry} {

    ASNA-become-unregistered-agent  $machine  $port  $registry

   ASNA-become-sensor  $machine  $port  $registry  tap-on-foot  F
   ASNA-become-sensor  $machine  $port  $registry  leg-in-swing-phase  F
   ASNA-become-sensor  $machine  $port  $registry  leg-in-stance-phase  F
   ASNA-become-sensor  $machine  $port  $registry  leg-is-lifted  F
   ASNA-become-sensor  $machine  $port  $registry  leg-is-extended  F

   ASNA-become-skill  $machine  $port  $registry  flexion-reflex
      {{{tap-on-foot  T}  {leg-in-swing-phase  T}}}
     {{{leg-is-lifted  T}}}
    {{{}}}
   ASNA-become-skill  $machine  $port  $registry  extension-reflex
      {{{tap-on-foot  T}  {leg-in-stance-phase  T}}}
     {{{leg-is-extended  T}}}
    {{{}}}
}
```

Keep in mind that this skill network represents a fragment of a larger network which comprises our virtual cat's ability to complete a walk cycle. The above small network used in this example is intended to show the activation flow at a particular portion of that larger network.



Synopsis: Cat Walk Cycle

I first made sure an asn daemon was running, and then started up asna, the action selection network agent manager program. From its command line I invoked the above tcl proc `cat-step`. The asn daemon received a message from an agent wishing to register with a registry/dispatcher called cat-step. The daemon checks its list of active registry/dispatchers, and notes that such a registry/service does not currently exist. It binds a socket, forks, and then execs a registry/dispatcher with the socket information as parameters to it. In this case, the port number for the socket was 5003, and the host on which the daemon was running was on a workstation named `archy`. The registry/dispatcher connects to the daemon and subsequently interperts any messages from it as commands to be evaluated.

global parameters

$\gamma = 70.0$
$\phi = 20.0$
$\delta = 50.0$
$\pi = 20.0$
$\theta = 45.0$

```
Attempting to connect to server archy at port 5003...

Successfully connected to a registry daemon at port 5003.
Evaluating command: ASNR-registration-info-request uagent-1

AC ERROR: failed to bind socket ([11] unable to bind socket)
    resetting ac_errno to 0

ERROR: attempt to bind a socket at port 5001 failed.
    Incrementing port value and trying again...
```

This continues for some time until it is finally successful..

```
Successfully setup a socket at port 5004 for agent <uagent-1>

Evaluating command:  ASNR-registration-info-request sensor:tap-on-foot
```

```
Successfully setup a socket at port 5005 for agent <sensor:tap-on-foot>

Evaluating command:  ASNR-register-sensor tap-on-foot F
```

The registry/dispatcher registers all the agents successfully, and then proceeds to update its local copy of the skill network.

```
added the skill <flexion-reflex> and the pv <tap-on-foot> = <T> to sensor <tap-on-foot>'s sk_c list.
added the skill <extension-reflex> and the pv <tap-on-foot> = <T> to sensor <tap-on-foot>'s sk_c list.
added the skill <flexion-reflex> and the pv <leg-in-swing-phase> = <T> to sensor <leg-in-swing-phase>'s sk_c
list.
added the skill <extension-reflex> and the pv <leg-in-stance-phase> = <T> to sensor <leg-in-stance-phase>'s
sk_c list.
added the skill <flexion-reflex> and the pv <leg-is-lifted> = <T> to sensor <leg-is-lifted>'s sk_a list.
added the skill <extension-reflex> and the pv <leg-is-extended> = <T> to sensor <leg-is-extended>'s sk_a list.
```

It then receives a message (which happens to come from the unregistered agent) to spread activation for 10 steps.

```
Evaluating command:  ASNR-spread-for 10

**** Time Step [1]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 0.000000
skill <extension-reflex> = 0.000000

lowering threshold to 40.500000
```

This continues for 10 steps, at which time the reflex skill agents still don't have any activation energy, as you would expect.

```
**** Time Step [10]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 0.000000
skill <extension-reflex> = 0.000000

lowering threshold to 15.690530
```

At this point, the sensor agent which measures whether or not the foot has been tapped sends a message changing the value of what it is measuring to T. Remember that we are seeing a small portion of the larger network comprising our virtual cat's walking ability. So in the simulated cat's world, some process has put pressure on the cat's paw, causing this sensor to note this. The registry/dispatcher then gets a message to spread activation for one timestep and we see, as we would hope, that each of the reflex skills get some activation sent to them.

```
Evaluating command:  ASNR-update-sensor-value {tap-on-foot T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
```

```
        pv pair:
                proposition = tap-on-foot
                value = T
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.

Evaluating command:  ASNR-spread-for 1

**** Time Step [11]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.

Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 5.000000
skill <extension-reflex> = 5.000000

lowering threshold to 14.121477
```

Now the tap goes away, and the activation spreading loop is invoked again. As you would hope, no further activation energy is spread to the skill agents because the stimulus (i.e. the tap on the foot) has been removed.

```
Evaluating command:  ASNR-update-sensor-value {tap-on-foot F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = tap-on-foot
                value = F
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.

Evaluating command:  ASNR-spread-for 10

**** Time Step [12]:
```

```
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 5.000000
skill <extension-reflex> = 5.000000

lowering threshold to 12.709330
```

The loop continues for 10 time steps...

```
**** Time Step [21]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 5.000000
skill <extension-reflex> = 5.000000

lowering threshold to 4.923854

Evaluating command:  ASNR-update-sensor-value {tap-on-foot T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = tap-on-foot
                value = T
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.
```

The registry/dispatcher then receives a message via the asna to spread activation for ten time steps.

```
Evaluating command:  ASNR-spread-for 10

**** Time Step [22]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.

Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 10.000000
skill <extension-reflex> = 10.000000

lowering threshold to 4.431469
```

The loop continues for 10 time steps...

```
**** Time Step [31]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.
```

```
Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 25.000000            skill <flexion-reflex> = 20.000000
skill <extension-reflex> = 25.000000          skill <extension-reflex> = 20.000000

lowering threshold to 1.716842
```

At this point, each reflex skill agent has much more activation than it needs to execute, but neither of them has all of their triggering stimuli met. In our simulation, we tell the process which was tapping the simulated cat's paw to stop, and the sensor agent `tap-on-foot` reports that the the tap is no longer present.

```
Evaluating command:  ASNR-update-sensor-value {tap-on-foot F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = tap-on-foot
                value = F
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.
```

The activation spreading loop is then invoked for two more times, and, as expected, no further activation flows to any of the skill agents. The activation level of the skill agents currently equals the global coefficient π, so none of the activation levels decay.

```
Evaluating command:  ASNR-spread-for 2

**** Time Step [32]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 20.000000
skill <extension-reflex> = 20.000000

lowering threshold to 1.545158

**** Time Step [33]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 20.000000
skill <extension-reflex> = 20.000000

lowering threshold to 1.390642
```

The sensor agent which measures whether or not the leg is in swing phase suddenly reports that it is, and when a message is received to spread activation for two time steps, we see that activation is spread to the skill agent `flexion-reflex`.

```
Evaluating command:  ASNR-update-sensor-value {leg-in-swing-phase T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = leg-in-swing-phase
                value = T
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = leg-in-swing-phase
                        value = T
    sk_a list is empty.
    sk_d list is empty.

Evaluating command:  ASNR-spread-for 2

**** Time Step [34]:
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 30.000000            skill <flexion-reflex> = 24.000000
skill <extension-reflex> = 20.000000          skill <extension-reflex> = 16.000000

lowering threshold to 1.251578

**** Time Step [35]:
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 34.000000            skill <flexion-reflex> = 27.200001
skill <extension-reflex> = 16.000000          skill <extension-reflex> = 12.800000

lowering threshold to 1.126420
```

The sensor agent `tap-on-foot` then reports that yes, there is a tap present. When activation is further spread through the network, since the skill agent `flexion-reflex` has all of its preconditions now met, and its activation level is higher than the threshold θ, it is sent a message from the registry/dispatcher to execute.

```
Evaluating command:  ASNR-update-sensor-value {tap-on-foot T}
the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = tap-on-foot
                value = T
    sk_c list:
        sk_c member:
```

```
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.

Evaluating command:  ASNR-spread-for 1

**** Time Step [36]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 42.200001            skill <flexion-reflex> = 28.133335
skill <extension-reflex> = 17.799999          skill <extension-reflex> = 11.866667

skill <flexion-reflex> has been determined to be active.

resetting threshold to 45.000000
```

The skill agent `flexion-reflex` proceeds to lift the leg up. The sensor agent which measures this effect in the world (`leg-is-lifted`) notes the change, and sends a message to that effect to the registry/dispatcher. The skill agent `flexion-reflex` has since completed its task, and sends a message to the registry/dispatcher to let it know that it is available to be called again. The registry/dispatcher notes that all of the predictions that the skill agent `flexion-reflex` made about the state of the world when it completed, namely that (`{leg-is-lifted T}`), have come to pass, so it resets the skill agent's activation level to 0.0.

```
Evaluating command:  ASNR-update-sensor-value {leg-is-lifted T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = leg-is-lifted
                value = T
    sk_c list is empty.
    sk_a list:
        sk_a member:
                skill name = flexion-reflex
                pv pair:
                        proposition = leg-is-lifted
                        value = T
    sk_d list is empty.

Evaluating command:  ASNR-mark-skill-as-completed flexion-reflex
```

```
just reset skill <flexion-reflex>'s activation level to 0.0
```

Some other agency in the world causes the leg to be put back down, and the sensor agent concerned duly notes this by sending a message back to the registry/dispatcher. Activation is then spread through the network for two more timesteps.

```
Evaluating command:  ASNR-update-sensor-value {leg-is-lifted F}
the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = leg-is-lifted
                value = F
    sk_c list is empty.
    sk_a list:
        sk_a member:
                skill name = flexion-reflex
                pv pair:
                        proposition = leg-is-lifted
                        value = T
    sk_d list is empty.
Evaluating command:  ASNR-spread-for 2

**** Time Step [37]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 15.000000
skill <extension-reflex> = 16.866667

lowering threshold to 40.500000

**** Time Step [38]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 30.000000            skill <flexion-reflex> = 23.136246
skill <extension-reflex> = 21.866667          skill <extension-reflex> = 16.863752

lowering threshold to 36.450001
```

The pressure on the cat's foot, which was being interpreted as a tap, finally goes away.

```
Evaluating command:  ASNR-update-sensor-value {tap-on-foot F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = tap-on-foot
                value = F
```

```
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.
```

When activation continues to spread through the network, we see that it is only due to the fact that the leg is in swing phase.

```
Evaluating command:  ASNR-spread-for 2

**** Time Step [39]:
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 33.136246            skill <flexion-reflex> = 26.508997
skill <extension-reflex> = 16.863752          skill <extension-reflex> = 13.491002

lowering threshold to 32.805000

**** Time Step [40]:
sending [10.000000] activation to skill <flexion-reflex> from sensor <leg-in-swing-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 36.508995            skill <flexion-reflex> = 29.207199
skill <extension-reflex> = 13.491002          skill <extension-reflex> = 10.792803

lowering threshold to 29.524500
```

When the leg goes out of swing phase, and the activation spreading loop is invoked for two more time steps, we see that no activation flows, as we would expect.

```
Evaluating command:  ASNR-update-sensor-value {leg-in-swing-phase F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = leg-in-swing-phase
                value = F
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = leg-in-swing-phase
                        value = T
    sk_a list is empty.
    sk_d list is empty.
```

```
Evaluating command:  ASNR-spread-for 2

**** Time Step [41]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 29.207199
skill <extension-reflex> = 10.792803

lowering threshold to 26.572050

**** Time Step [42]:
Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <flexion-reflex> = 29.207199
skill <extension-reflex> = 10.792803

lowering threshold to 23.914845
```

The sensor agent which measures whether or not the leg is in stance phase, suddenly reports that it is, and when a message is received to spread activation for two time steps, we see that activation is spread to the skill agent `extension-reflex`.

```
Evaluating command:  ASNR-update-sensor-value {leg-in-stance-phase T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = leg-in-stance-phase
                value = T
    sk_c list:
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = leg-in-stance-phase
                        value = T
    sk_a list is empty.
    sk_d list is empty.

Evaluating command:  ASNR-spread-for 2

**** Time Step [43]:
sending [10.000000] activation to skill <extension-reflex> from sensor <leg-in-stance-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 29.207199            skill <flexion-reflex> = 23.365759
skill <extension-reflex> = 20.792803          skill <extension-reflex> = 16.634243

lowering threshold to 21.523359

**** Time Step [44]:
sending [10.000000] activation to skill <extension-reflex> from sensor <leg-in-stance-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 23.365759            skill <flexion-reflex> = 18.692608
skill <extension-reflex> = 26.634243          skill <extension-reflex> = 21.307394

lowering threshold to 19.371023
```

The sensor agent `tap-on-foot` then reports that yes, there is a tap present. When activation is further spread through the network, since the skill agent `extension-reflex` has all of its preconditions now met, and its activation level is higher than the threshold θ, it is sent a message from the registry/dispatcher to execute.

```
Evaluating command:  ASNR-update-sensor-value {tap-on-foot T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = tap-on-foot
                value = T
    sk_c list:
        sk_c member:
                skill name = flexion-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
        sk_c member:
                skill name = extension-reflex
                pv pair:
                        proposition = tap-on-foot
                        value = T
    sk_a list is empty.
    sk_d list is empty.
```

The skill agent `extension-reflex` proceeds to extend the leg out. The sensor agent which measures this effect in the world (`leg-is-extended`) notes the change, and sends a message to that effect to the registry/dispatcher. The skill agent `extension-reflex` has since completed its task, and s ends a message to the registry/dispatcher to let it know that it is available to be called again. The registry/dispatcher notes that all of the predictions that the skill agent `extension-reflex` made about the state of the world when it completed, namely that (`{leg-is-extended T}`), have come to pass, so it resets the skill agent's activation level to 0.0.

```
Evaluating command:  ASNR-spread-for 2

**** Time Step [45]:
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.
sending [10.000000] activation to skill <extension-reflex> from sensor <leg-in-stance-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 23.692608            skill <flexion-reflex> = 15.795071
skill <extension-reflex> = 36.307396          skill <extension-reflex> = 24.204929

skill <extension-reflex> has been determined to be active.

resetting threshold to 45.000000

**** Time Step [46]:
```

```
sending [5.000000] activation to skill <flexion-reflex> from sensor <tap-on-foot>.
sending [5.000000] activation to skill <extension-reflex> from sensor <tap-on-foot>.
sending [10.000000] activation to skill <extension-reflex> from sensor <leg-in-stance-phase>.

Skills' activation level before decay:        Skills' activation level after decay:
skill <flexion-reflex> = 20.795071            skill <flexion-reflex> = 13.863380
skill <extension-reflex> = 39.204929          skill <extension-reflex> = 26.136620

lowering threshold to 40.500000

Evaluating command:  ASNR-update-sensor-value {leg-is-extended T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = leg-is-extended
                value = T
    sk_c list is empty.
    sk_a list:
        sk_a member:
                skill name = extension-reflex
                pv pair:
                        proposition = leg-is-extended
                        value = T
    sk_d list is empty.

Evaluating command:  ASNR-mark-skill-as-completed extension-reflex

just reset skill <extension-reflex>'s activation level to 0.0
```

Discussion: Cat Walk Cycle

The algorithm worked very well for this example. It's interesting to note how well the low-level neurological notion of activation and inhibition are mirrored by the spreading of positive and negative activation flow in the skill network. The direct but subtle relationship between perception and action is especially interesting viewed on such a step-by-step basis.

A Door Openin' Dude

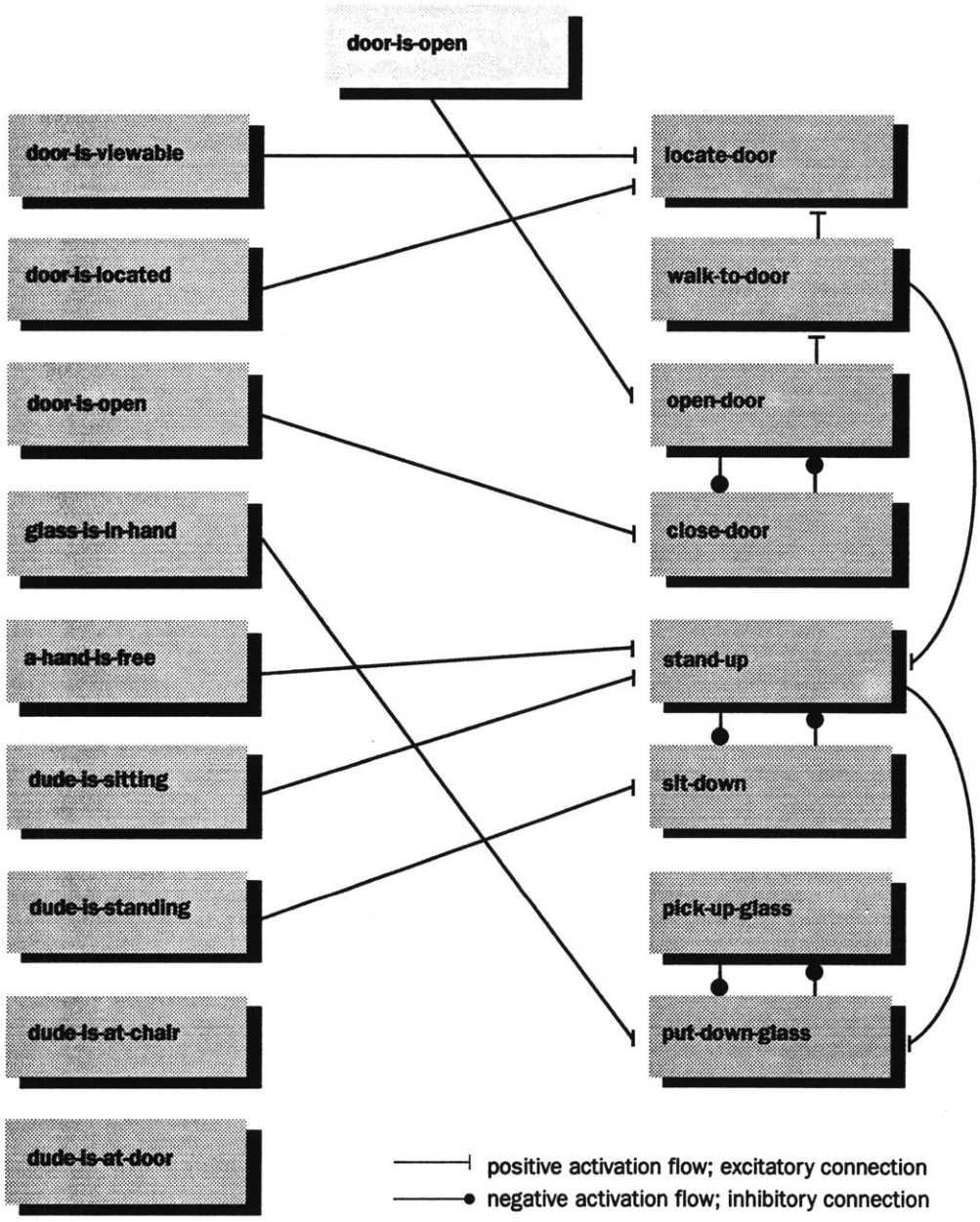
Given that Build-a-Dude is designed to be used to create autonomous creatures that exist in virtual environments, it makes sense to give an example of one. This example (all three parts) is intended to serve as a simple demonstartion of a situation a prototypical virtual actor might find itself in; namely "Open a door." The example is broken up into three sections, or "takes", which show off progressively more complex behavior. The first take concerns a straightforward situation in which the dude is given the goal of getting the door open. In the second, we give it the addtional task of closing the window. Finally, in the third take, we remove its ability to walk, and see if it can take advantage of the fact that it still knows how to crawl to get to the door and open it.

Scenario: Dude Take 1

So how does one describe a skill network to open a door? The one used for this example is given below:

```
proc door-closin-dude-uagent {port host registry} {

  ASNA-become-unregistered-agent $host $port $registry
}

proc door-closin-dude-goals {port host registry} {

  ASNA-become-goal $host $port $registry door-is-open T
}

proc door-closin-dude-sensors {port host registry} {

  ASNA-become-sensor $host $port $registry door-is-viewable T
  ASNA-become-sensor $host $port $registry door-is-located F
  ASNA-become-sensor $host $port $registry door-is-open F
  ASNA-become-sensor $host $port $registry glass-is-in-hand T
  ASNA-become-sensor $host $port $registry a-hand-is-free F
  ASNA-become-sensor $host $port $registry dude-is-sitting T
  ASNA-become-sensor $host $port $registry dude-is-standing F
  ASNA-become-sensor $host $port $registry dude-is-at-chair T
  ASNA-become-sensor $host $port $registry dude-is-at-door F
}

proc door-closin-dude-skills {port host registry} {

  ASNA-become-skill $host $port $registry locate-door
    {{{door-is-viewable T} {door-is-located F}}}
    {{{door-is-located T}}}
    {{{}}}
  ASNA-become-skill $host $port $registry walk-to-door
    {{{door-is-located T}
     {dude-is-standing T}
     {dude-is-at-door F}}}
    {{{dude-is-at-door T}}}
    {{{}}}
  ASNA-become-skill $host $port $registry open-door
    {{{dude-is-at-door T}
     {door-is-located T}
     {door-is-open F}}}
    {{{door-is-open T}}}
    {{{}}}
  ASNA-become-skill $host $port $registry close-door
    {{{dude-is-at-door T}
     {door-is-located T}
     {door-is-open F}}}
    {{{}}}
    {{{door-is-open F}}}
  ASNA-become-skill $host $port $registry stand-up
    {{{dude-is-standing F} {a-hand-is-free T}}}
    {{{dude-is-standing T}}}
    {{{}}}
  ASNA-become-skill $host $port $registry sit-down
    {{{dude-is-sitting F}}}
    {{{dude-is-sitting T}}}
    {{{}}}
```

```
  ASNA-become-skill $host $port $registry pick-up-glass
    {{{glass-is-in-hand F} {a-hand-is-free T}}}
    {{{glass-is-in-hand T}}}
    {{{a-hand-is-free F}}}
  ASNA-become-skill $host $port $registry put-down-glass
    {{{glass-is-in-hand T}}}
    {{{a-hand-is-free T}}}
    {{{glass-is-in-hand F}}}
}
```



Given this description of our dude, what will we see in this example? Note that the dude has a goal agent which wants the door open. According to its

sensor agents, the dude starts out sitting in a chair with a glass in its hand. In order to open the door, the dude needs to locate the door, put down the glass (it seems to be a one-armed dude), stand up, walk over to the door, and then open the door. Given that this is but one of a myriad of possible action selection sequences that could conceivably be selected from the above skill network, let's see if the algorithm brings this chain of events about.

Synopsis: Dude Take 1

We'll begin just after the registry/dispatcher has been started up by the asn daemon, due to a "registration-info-request" from an agent called uagent-1. The registry/dispatcher was told to call back the asn daemon at port 5002:

global parameters

$\gamma = 70.0$
$\phi = 20.0$
$\delta = 50.0$
$\pi = 20.0$
$\theta = 45.0$

sidenote

In the following examples, three ellipses (... ... ...) are used to denote where parts of the log file have been ommitted fror brevity's sake.

```
Attempting to connect to server archy at port 5002...

Successfully connected to a registry daemon at port 5002.
Evaluating command:

ASNR-registration-info-request uagent-1


AC ERROR: failed to bind socket ([11] unable to bind socket)
    resetting ac_errno to 0

ERROR: attempt to bind a socket at port 5001 failed.
    Incrementing port value and trying again...
```

This continues for some time until it is finally successful...

```
Successfully setup a socket at port 5003 for agent <uagent-1>
```

The registry/dispatcher registers all the agents successfully, and then proceeds to update its local copy of the skill network.

```
added the skill <locate-door> and the pv <door-is-viewable> = <T> to sensor <door-is-viewable>'s sk_c list.
added the skill <locate-door> and the pv <door-is-located> = <F> to sensor <door-is-located>'s sk_c list.
added the skill <locate-door> and the pv <door-is-located> = <T> to sensor <door-is-located>'s sk_a list.
... ... ...
added the sensor <a-hand-is-free> to skill <put-down-glass>'s successor list.
added the sensor <a-hand-is-free> to skill <pick-up-glass>'s predecessor list.
```

The registry/dispatcher receives a message to spread activation. Note that since this is the first time step that activation is flowing, there is only a contribution from the sensor agents and the goal agents, since none of the skill agents have any activation of their own yet.

```
Evaluating command:  ASNR-spread-for 1

**** Time Step [1]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
sending [2.500000] activation to skill <locate-door> from sensor <door-is-located>.
... ... ...
```

```
skill <open-door> decreases (inhibits) skill <close-door> with 0.000000 for <door-is-open>.

Skills' activation level before decay:        Didn't need to decay skills' activation level this time.
skill <locate-door> = 12.500000
skill <walk-to-door> = 2.222222
skill <open-door> = 73.333336
skill <close-door> = 3.333333
skill <stand-up> = 5.000000
skill <sit-down> = 0.000000
skill <pick-up-glass> = 0.000000
skill <put-down-glass> = 10.000000

lowering threshold to 40.500000
```

It then receives a message to spread activation for twenty time steps. Very quickly (in one time step, which is less than .07 seconds wall clock time in the current implementation) a skill agent is selected. Note that although the skill agent has been sent a message to begin executing, the registry/dispatcher continues to spread activation in spite of the fact that it has no indication that the skill has completed.

```
Evaluating command:  ASNR-spread-for 20

**** Time Step [2]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
skill <locate-door> has been determined to be active.

resetting threshold to 45.000000

**** Time Step [3]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:        Skills' activation level after decay:
skill <locate-door> = 138.033295              skill <locate-door> = 50.333614
skill <walk-to-door> = 75.808105              skill <walk-to-door> = 27.643303
skill <open-door> = 117.004852                skill <open-door> = 42.665630
skill <close-door> = 4.555625                 skill <close-door> = 1.661201
skill <stand-up> = 54.540867                  skill <stand-up> = 19.888239
skill <sit-down> = 0.000000          skill <sit-down> = 0.000000
skill <pick-up-glass> = 2.535991              skill <pick-up-glass> = 0:924745
skill <put-down-glass> = 46.300148   skill <put-down-glass> = 16.883274

lowering threshold to 29.524500
```

Some time has now passed, and it seems that the skill agent `locate-door` was successful. Regardless of whether or not it was due to the skill agen"s intervention, the sensor agent `door-is-located` reports a T value indicating that the door has been located. Soon after, a message arrives from the skill agent `locate-door` notifying the registry/dispatcher that, for better or worse, as far as the skill agent is concerned, it has completed its task. Note that this is independent of its add-list and delete-list—those were only predictions about how it thought it would affect the world, only what comes to the dude via its sensor agents counts. Since the registry/dispatcher notes that the

predictions the skill agent made have exactly coincided with what the sensor agents have reported, it resets the skill agent's activation level to 0.0. If the state of the world had been different, the registry/dispatcher would have set the skill agent's activation back to some fraction of its current activation, based on what percentage of its predictions had come true, multiplied by some constant. The constant would be derived from how many times the skill agent had been called recently, versus how many times it can be invoked, and obviously this value would be agent-dependent.

```
Evaluating command:  ASNR-update-sensors-values (door-is-located T)

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = door-is-located
                value = T
    sk_c list:
        sk_c member:
                skill name = locate-door
                pv pair:
                        proposition = door-is-located
                        value = F
        sk_c member:
                skill name = walk-to-door
                pv pair:
                        proposition = door-is-located
                        value = T
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = door-is-located
                        value = T
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = door-is-located
                        value = T
    sk_a list:
        sk_a member:
                skill name = locate-door
                pv pair:
                        proposition = door-is-located
                        value = T
    sk_d list is empty.

Evaluating command:  ASNR-mark-skill-as-completed locate-door

just reset skill <locate-door>'s activation level to 0.0

**** Time Step [7]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
skill <put-down-glass> has been determined to be active.

resetting threshold to 45.000000

**** Time Step [10]:
```

```
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
**** Time Step [12]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Evaluating command:  ASNR-update-sensors-values {a-hand-is-free T} {glass-is-in-hand F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = a-hand-is-free
                value = T
    sk_c list:
        sk_c member:
                skill name = stand-up
                pv pair:
                        proposition = a-hand-is-free
                        value = T
        sk_c member:
                skill name = pick-up-glass
                pv pair:
                        proposition = a-hand-is-free
                        value = T
    sk_a list:
        sk_a member:
                skill name = put-down-glass
                pv pair:
                        proposition = a-hand-is-free
                        value = T
    sk_d list:
        sk_d member:
                skill name = pick-up-glass
                pv pair:
                        proposition = a-hand-is-free
                        value = F

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = glass-is-in-hand
                value = F
    sk_c list:
        sk_c member:
                skill name = pick-up-glass
                pv pair:
                        proposition = glass-is-in-hand
                        value = F
        sk_c member:
                skill name = put-down-glass
                pv pair:
                        proposition = glass-is-in-hand
                        value = T
    sk_a list:
        sk_a member:
                skill name = pick-up-glass
                pv pair:
                        proposition = glass-is-in-hand
                        value = T
    sk_d list:
        sk_d member:
```

```
                skill name = put-down-glass
                pv pair:
                        proposition = glass-is-in-hand
                        value = F
```

Evaluating command: ASNR-mark-skill-as-completed put-down-glass

```
just reset skill <put-down-glass>'s activation level to 0.0

**** Time Step [13]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:      Skills' activation level after decay:
skill <locate-door> = 38.872704             skill <locate-door> = 17.927721
skill <walk-to-door> = 91.974380            skill <walk-to-door> = 42.417706
skill <open-door> = 123.806763              skill <open-door> = 57.098499
skill <close-door> = 5.000000               skill <close-door> = 2.305952
skill <stand-up> = 76.656906                skill <stand-up> = 35.353436
skill <sit-down> = 0.000000                 skill <sit-down> = 0.000000
skill <pick-up-glass> = 10.205845           skill <pick-up-glass> = 4.706838
skill <put-down-glass> = 0.411689           skill <put-down-glass> = 0.189867
```

skill <stand-up> has been determined to be active.

```
resetting threshold to 45.000000

**** Time Step [14]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
**** Time Step [16]:
... ... ...
```

Evaluating command: ASNR-update-sensors-values {dude-is-standing T}

```
the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = dude-is-standing
                value = T
    sk_c list:
        sk_c member:
                skill name = walk-to-door
                pv pair:
                        proposition = dude-is-standing
                        value = T
        sk_c member:
                skill name = stand-up
                pv pair:
                        proposition = dude-is-standing
                        value = F
    sk_a list:
        sk_a member:
                skill name = stand-up
                pv pair:
                        proposition = dude-is-standing
                        value = T
    sk_d list is empty.
```

Evaluating command: ASNR-mark-skill-as-completed stand-up

```
just reset skill <stand-up>'s activation level to 0.0
```

```
**** Time Step [17]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:        Skills' activation level after decay:
skill <locate-door> = 50.007641               skill <locate-door> = 25.726070
skill <walk-to-door> = 105.443024             skill <walk-to-door> = 54.244404
skill <open-door> = 128.706787                skill <open-door> = 66.212280
skill <close-door> = 6.348065                 skill <close-door> = 3.265717
skill <stand-up> = 5.000000                   skill <stand-up> = 2.572214
skill <sit-down> = 0.000000                   skill <sit-down> = 0.000000
skill <pick-up-glass> = 14.892086             skill <pick-up-glass> = 7.661127
skill <put-down-glass> = 0.618499             skill <put-down-glass> = 0.318182

skill <walk-to-door> has been determined to be active.

resetting threshold to 45.000000

**** Time Step [18]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
**** Time Step [21]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Evaluating command:  ASNR-update-sensors-values {dude-is-at-door T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = dude-is-at-door
                value = T
    sk_c list:
        sk_c member:
                skill name = walk-to-door
                pv pair:
                        proposition = dude-is-at-door
                        value = F
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
    sk_a list:
        sk_a member:
                skill name = walk-to-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
    sk_d list is empty.

Evaluating command:  ASNR-mark-skill-as-completed walk-to-door

just reset skill <walk-to-door>'s activation level to 0.0

**** Time Step [22]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
```

```
skill <open-door> has been determined to be active.

resetting threshold to 45.000000

**** Time Step [23]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Evaluating command:  ASNR-update-sensors-values {door-is-open T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = door-is-open
                value = T
    sk_c list:
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = door-is-open
                        value = F
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = door-is-open
                        value = F
    sk_a list:
        sk_a member:
                skill name = open-door
                pv pair:
                        proposition = door-is-open
                        value = T
    sk_d list:
        sk_d member:
                skill name = close-door
                pv pair:
                        proposition = door-is-open
                        value = F

Evaluating command:  ASNR-mark-skill-as-completed open-door

just reset skill <open-door>'s activation level to 0.0
```

Discussion: Dude Take 1

In constructing this example, I originally designed a network consisting of about 35 agents. I had skill agents for picking up and putting down other things besides a glass, and for going places other than the door. I also had sensor agents for detecting all these things. When I tried to run it through the system, however, I ran into a problem. For some reason, I could never allocate more than 25 TCP/IP stream sockets per process, which meant I couldn't construct a network of more than 25 agents. The reasons for this are still not clear, but I currently believe it is related to the fact that processes under UNIX are limited to a finite (sometimes 32, sometimes 64, but almost never more than 256) number of file descriptors. The fix for this problem is to treat a socket as a more precious resource than I had; unfortunately this entailed writing, or rather, re-

writing, more code than I had time to do. As I'll talk about in Chapter 6 (**Agents as Agencies**), this is at the head of the list of things to work on as soon as this document is finished.

Although it's true that the selection of the global parameters is still an unknown quantity, I observerved running this ex-ample was how forgiving the algorithm is if you have enough time to let it run. In other words, although action selection may occur faster or slower, within a fairly broad range of values, it will continue to do the right thing, albeit more or less optimally.

Perhaps the greatest lesson I learned from this example was how important it is to start thinking how to implement template skills. Given that many of the skill agents used in this example are quite amenable to functional abstraction (`walk-to(x)`, `pick-up(y)`, `put-down(z)`, etc.), it became clear that just having such template skills around wouldn't be enough. Given that we had skill agents `walk-to(x)`, `run-to(m)`, and `slither-to(s)`, each one of these skill agents would be receiving the same amount of activation in the skill network. What is needed, then, is some notion of costs associated with the invocation of different skill agents. Since a given skill agent can have some notion of what sort of cost it will incur only as a function of a prediction about its effect on the state of the world, it seems that the best way to deal with this would be by the instantiation of goal agents concerned with conserving resources. We'll see an example which solves this same problem in a different way in **Dude Take 3**, where the skill agent `crawl-to-door` "costs" more because the dude has to call more skill agents to get to it than it does to just call theskill agent `walk-to-door`.

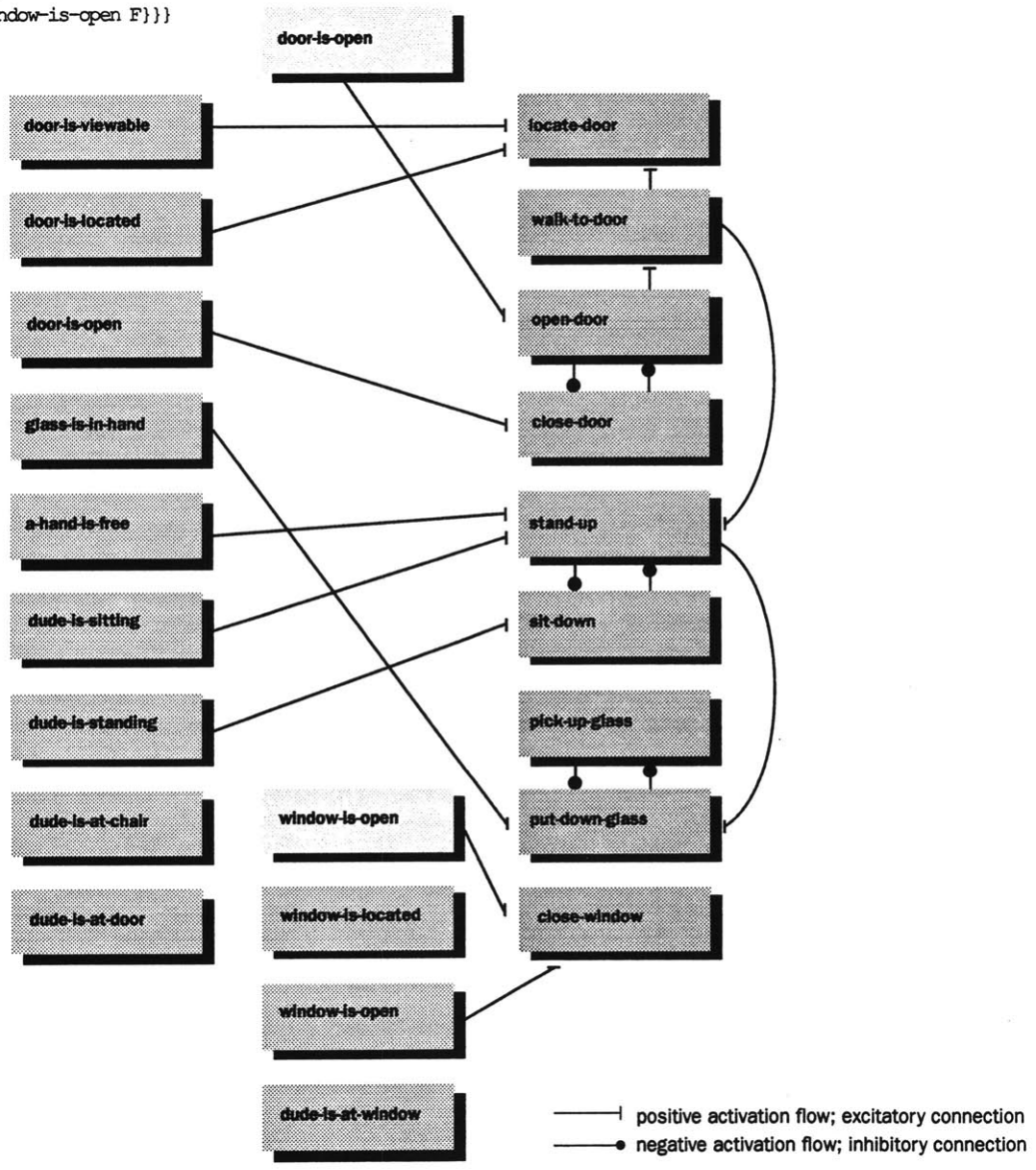
A Door Openin', Window Closin' Dude: Dude Take 2

In the previous example, we showed how the action selection algorithm worked for a simple prototypical virtual actor scenario. But in a "real" virtual world, virtual actors should be able to walk and chew gum at the same time. In the last example, we saw how the registry/dispatcher continued to spread activation asynchronous to the execution of a given skill agent. What we didn't see, though, was the registry/dispatcher sending an execute message to a skill agent before the currently executing skill had completed. This example is intended to show how my algorithm, and its subsequent implementation, can successfully coordinate parallel execution of skills, which was not implemented by Maes. In this example I'll gift my little dude with the ability to close windows in addition to the skills it already has, and put it in a situation where it has the opportunity to close a window on its way to opening the door.

Scenario: Dude Take 2

To extend our dude from Take 1 to close windows, we need to add five new agents: a goal agent to instantiate the desire to close the window, a sensor agent to note when the window's location is known, one to measure when the dude is near the window, one to detect whether the window is open or not, and finally a skill agent to actually close the window. The requisite commands to asna look like this:

```
ASNA-become-goal $host $port $registry window-is-open F
ASNA-become-sensor $host $port $registry window-is-located T
ASNA-become-sensor $host $port $registry window-is-open T
ASNA-become-sensor $host $port $registry dude-is-at-window F
ASNA-become-skill $host $port $registry close-window
   {{{dude-is-at-window T} {window-is-located T} {window-is-open T}}}
   {{{}}}
   {{{window-is-open F}}}
```



Synopsis: Dude Take 2

global parameters

$\gamma = 70.0$
$\phi = 20.0$
$\delta = 50.0$
$\pi = 20.0$
$\theta = 45.0$

Things proceed basically the same as Take 1, except that once the `walk-to-door` skill agent is called, I explicitly cause it to take longer to complete. Let's take a look at the registry/dispatcher's log file from that point and see what happened:

```
skill <walk-to-door> has been determined to be active.

resetting threshold to 45.000000

Evaluating command:  ASNR-spread-for 1

**** Time Step [11]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:        Skills' activation level after decay:
skill <locate-door> = 59.919205               skill <locate-door> = 26.447865
skill <walk-to-door> = 141.229050             skill <walk-to-door> = 62.337391
skill <open-door> = 146.083466                skill <open-door> = 64.480095
skill <close-door> = 6.889864                 skill <close-door> = 3.041132
skill <stand-up> = 10.221228                  skill <stand-up> = 4.511569
skill <sit-down> = 0.000000                   skill <sit-down> = 0.000000
skill <pick-up-glass> = 16.844547             skill <pick-up-glass> = 7.435050
skill <put-down-glass> = 1.117624             skill <put-down-glass> = 0.493310
skill <close-window> = 25.495697              skill <close-window> = 11.253600

lowering threshold to 40.500000
```

Activation flow continues for several time steps, with no other actions (i.e. skill agents) being selected. Then a message arrives from the sensor agent `dude-is-at-window`. Note that the dude has still not arrived at the door (i.e. the sensor agent `dude-is-at-door` has not reported a value of T yet), nor has the `walk-to-door` skill completed yet.

```
Evaluating command:  ASNR-update-sensors-values {dude-is-at-window T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = dude-is-at-window
                value = T
    sk_c list:
        sk_c member:
                skill name = close-window
                pv pair:
                        proposition = dude-is-at-window
                        value = T
    sk_a list is empty.
    sk_d list is empty.


**** Time Step [21]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:        Skills' activation level after decay:
```

```
skill <locate-door> = 75.854149          skill <locate-door> = 32.508289
skill <walk-to-door> = 127.103065        skill <walk-to-door> = 54.471687
skill <open-door> = 136.247635           skill <open-door> = 58.390713
skill <close-door> = 6.767537            skill <close-door> = 2.900317
skill <stand-up> = 32.427864             skill <stand-up> = 13.897387
skill <sit-down> = 0.000000              skill <sit-down> = 0.000000
skill <pick-up-glass> = 10.619451        skill <pick-up-glass> = 4.551105
skill <put-down-glass> = 0.661410        skill <put-down-glass> = 0.283456
skill <close-window> = 30.327065         skill <close-window> = 12.997062

lowering threshold to 14.121477
```

Suddenly, the window is within reach of the dude. Regardless of the fact that it is moving across the room on its way to the door, it now has the opportunity to close the window, which would satisfy the goal agent who wants the proposition `window-is-open` to have a value of `F`.

```
**** Time Step [22]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:   Skills' activation level after decay:
skill <locate-door> = 75.016579          skill <locate-door> = 32.255135
skill <walk-to-door> = 124.714203        skill <walk-to-door> = 53.623795
skill <open-door> = 135.119980           skill <open-door> = 58.098000
skill <close-door> = 6.729260            skill <close-door> = 2.893403
skill <stand-up> = 32.794773             skill <stand-up> = 14.100881
skill <sit-down> = 0.000000              skill <sit-down> = 0.000000
skill <pick-up-glass> = 10.608535        skill <pick-up-glass> = 4.561388
skill <put-down-glass> = 0.650158        skill <put-down-glass> = 0.279551
skill <close-window> = 32.997063         skill <close-window> = 14.187860

skill <close-window> has been determined to be active.

resetting threshold to 45.000000

Evaluating command:  ASNR-update-sensors-values {window-is-open F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = window-is-open
                value = F
    sk_c list:
        sk_c member:
                skill name = close-window
                pv pair:
                        proposition = window-is-open
                        value = T
    sk_a list is empty.
    sk_d list:
        sk_d member:
                skill name = close-window
                pv pair:
                        proposition = window-is-open
                        value = F

Evaluating command:  ASNR-mark-skill-as-completed close-window

just reset skill <close-window>'s activation level to 0.0
```

Evaluating command: ASNR-update-sensors-values {dude-is-at-door T}

```
the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = dude-is-at-door
                value = T
    sk_c list:
        sk_c member:
                skill name = walk-to-door
                pv pair:
                        proposition = dude-is-at-door
                        value = F
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
    sk_a list:
        sk_a member:
                skill name = walk-to-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
    sk_d list is empty.
```

Evaluating command: ASNR-mark-skill-as-completed walk-to-door

```
just reset skill <walk-to-door>'s activation level to 0.0

**** Time Step [23]:
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:      Skills' activation level after decay:
skill <locate-door> = 74.510269             skill <locate-door> = 46.996788
skill <walk-to-door> = 5.000000             skill <walk-to-door> = 3.153712
skill <open-door> = 138.086792              skill <open-door> = 87.097198
skill <close-door> = 9.988794               skill <close-door> = 6.300356
skill <stand-up> = 33.201759                skill <stand-up> = 20.941759
skill <sit-down> = 0.000000                 skill <sit-down> = 0.000000
skill <pick-up-glass> = 10.605364           skill <pick-up-glass> = 6.689253
skill <put-down-glass> = 0.651627           skill <put-down-glass> = 0.411009
skill <close-window> = 13.333334            skill <close-window> = 8.409900
```

skill <open-door> has been determined to be active.

```
resetting threshold to 45.000000
```

Evaluating command: ASNR-update-sensors-values {door-is-open T}

```
the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = door-is-open
```

```
                value = T
    sk_c list:
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = door-is-open
                        value = F
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = door-is-open
                        value = F
    sk_a list:
        sk_a member:
                skill name = open-door
                pv pair:
                        proposition = door-is-open
                        value = T
    sk_d list:
        sk_d member:
                skill name = close-door
                pv pair:
                        proposition = door-is-open
                        value = F

Evaluating command:  ASNR-mark-skill-as-completed open-door

just reset skill <open-door>'s activation level to 0.0
```

Discussion: Dude Take 2

As hoped, this example showed the implementation's ability to deal with parallel execution of tasks. This example took under an hour to think up, write and run. The actual example ran at about 13Hz using five asnas, one registry/dispatcher, and one asn daemon, all running on the same Stardent Titan 1500.

A Door Openin', Crawlin' Dude: Dude Take 3

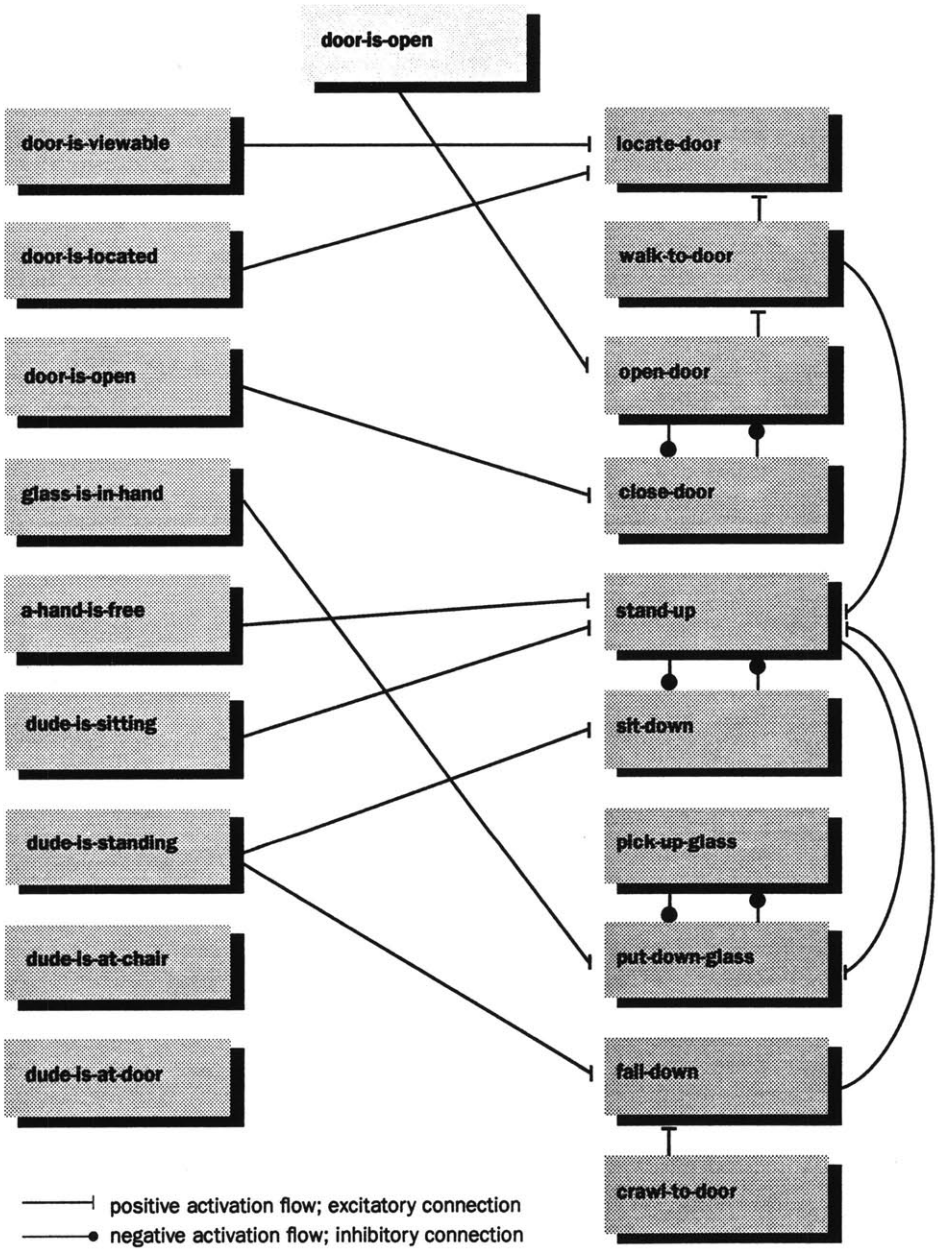
This example is intended to show how my algorithm, and its subsequent implementation, can successfully handle a situation in which a skill agent becomes disabled, and subsequently choose actions which satisfy the goal agent(s). This example is notable for the fact that it explicitly shows that my implementation can deal with disabled agents, and amputates them from the skill network, as discussed in Chapters 3 and 4.

Scenario: Dude Take 3

The scenario is similar to Take 1, except that when the dude decides to call the skill agent walk-to-door, the registry/dispatcher discovers that the skill agent does not respond to any messages. The registry/dispatcher needs to mark the skill disabled, and choose another course of action to open the door. To make this possible, we give our dude two more skill agents: one to fall to the ground, and one to crawl to the door. The

requisite commands to the asna look like this:

```
ASNA-become-skill $host $port $registry fall-down
   {{{dude-is-standing T}}}
   {{{}}}
   {{{dude-is-standing F}}}

ASNA-become-skill $host $port $registry crawl-to-door
   {{{door-is-located T}
    {dude-is-standing F}
    {dude-is-at-door F}}}
   {{{dude-is-at-door T}}}
   {{{}}}
```



Synopsis: Dude Take 3

global parameters

$\gamma = 70.0$
$\phi = 20.0$
$\delta = 50.0$
$\pi = 20.0$
$\theta = 45.0$

Things proceed basically the same as Take 1, except that once the `walk-to-door` skill agent is called, I explicitly cause the skill agent to ignore any messages from the registry/dispatcher, effectively rendering it incommunicado with the rest of the skill network. Let's take a look at the registry/dispatcher's log file and see what happened:

```
skill <walk-to-door> has been determined to be active.

resetting threshold to 45.000000

I'm checking to see if any previously marked disabled agents need to be amputated...

**** Time Step [11]:
I'm marking any agents which I find to be disabled...

sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
**** Time Step [12]:
I'm marking any agents which I find to be disabled...

NOTE: agent <walk-to-door> has not responded to a message sent 10.0 seconds ago -- marking it disabled

sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Evaluating command:  ASNR-registration-info-request skill:crawl-to-door
```

The registry/dispatcher gets a message from the asn daemon about registering a new agent. This new agent will give the dude the ability to get to the door, albiet in a less efficient way (by crawling there). Where did this skill come from? In this case, I explicitly started up a new asna which sent a registration info request to the asn daemon, but it could have come from some user or program giving advice to this skill network. In other words, this shows Build-a-Dude doing, albeit in a very simple way, *learning by example.*

The registry/dispatcher binds a socket in preparation and accepts the connection from the agent, which has not yet registered itself.

```
Successfully setup a socket at port 5025 for agent <skill:crawl-to-door>
```

The registry/dispatcher then receives a registration request from the unregistered agent communicating over port 5025.

```
Evaluating command:  ASNR-register-skill crawl-to-door {{door-is-located T}
                                                        {dude-is-standing F}
                                                        {dude-is-at-door F}}
                                                       {{dude-is-at-door T}}
                                                       {{}}
```

the registry/dispatcher successfully registers the new skill agent, `crawl-to-door`, and updates its agent database (the transcript of which is ommitted for brevity's sake). It then receives another message to spread activation for one time step.

```
**** Time Step [13]:
I'm marking any agents which I find to be disabled...
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
lowering threshold to 32.805000
```

As the registry/dispatcher does every few time steps, it checks to see if any agents it has previously marked disabled need to be amputated. In this case, the agent `walk-to-door` has ignored repeated messages sent to it, so the registry/dispatcher considers it dead, and it amputates it from the skill network. There are two cases in which the registry explicitly amputates an agent from the skill network. The first is when the agent repeatedly ignores messages sent to it. The registry/dispatcher has a variable threshold of how many messages it considers too many. The second case, which is not shown in this example, is when the communication channel (i.e. the file descriptor associated with the socket being used by the agent to talk to the registry/dispatcher) becomes invalid. This would happen if the process associated with the agent, or the machine on which the process was running, or the network connection between the two machines went down. Either way, when a skill agent is amputated from a skill network by a registry/dispatcher, the registry/dispatcher explicitly closes the communication channel (i.e. freeing up the file descriptor for use by others), it explicitly removes it from its skill agent list, it frees all memory associated with it, and updates its agent database, which is its locally maintained copy of the skill network, containing all the links through which activation flows.

```
I'm checking to see if any previously marked disabled agents need to be amputated...

Amputating skill agent 8 <walk-to-door>...
...deleting it from skill agent list...done
...freeing its associated memory...done
...updating the agent database...done
The skill agent is now amputated

Evaluating command:  ASNR-spread-for 1

**** Time Step [14]:
I'm marking any agents which I find to be disabled...
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
lowering threshold to 29.524500
```

Activation spreading continues for some time with no skill being selected,

until finally the skill agent `fall-down` is selected, since the only option it has left to get to the door is to crawl there, and it can't crawl standing up.

```
Evaluating command:  ASNR-spread-for 1

**** Time Step [34]:
I'm marking any agents which I find to be disabled...
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:        Skills' activation level after decay:
skill <locate-door> = 45.710590               skill <locate-door> = 17.817162
skill <open-door> = 122.906883                skill <open-door> = 47.906879
skill <close-door> = 5.000000                 skill <close-door> = 1.948910
skill <stand-up> = 178.072449                 skill <stand-up> = 69.409424
skill <sit-down> = 0.000000                   skill <sit-down> = 0.000000
skill <fall-down> = 10.925056                 skill <fall-down> = 4.258389
skill <pick-up-glass> = 10.521286             skill <pick-up-glass> = 4.101007
skill <put-down-glass> = 0.585858             skill <put-down-glass> = 0.228357
skill <crawl-to-door> = 88.074554             skill <crawl-to-door> = 34.329868

skill <fall-down> has been determined to be active.

resetting threshold to 45.000000

I'm checking to see if any previously marked disabled agents need to be amputated...

Evaluating command:  ASNR-update-sensors-values {dude-is-standing F}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = dude-is-standing
                value = F
    sk_c list:
        sk_c member:
                skill name = stand-up
                pv pair:
                        proposition = dude-is-standing
                        value = F
        sk_c member:
                skill name = fall-down
                pv pair:
                        proposition = dude-is-standing
                        value = T
        sk_c member:
                skill name = crawl-to-door
                pv pair:
                        proposition = dude-is-standing
                        value = F
    sk_a list:
        sk_a member:
                skill name = stand-up
                pv pair:
                        proposition = dude-is-standing
                        value = T
    sk_d list:
        sk_d member:
                skill name = fall-down
                pv pair:
```

```
                        proposition = dude-is-standing
                        value = F
Evaluating command:  ASNR-mark-skill-as-completed fall-down

just reset skill <fall-down>'s activation level to 0.0

**** Time Step [35]:
I'm marking any agents which I find to be disabled...
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:      Skills' activation level after decay:
skill <locate-door> = 45.634323             skill <locate-door> = 22.316496
skill <open-door> = 123.996719              skill <open-door> = 60.637962
skill <close-door> = 6.089837               skill <close-door> = 2.978105
skill <stand-up> = 81.047966                skill <stand-up> = 39.634785
skill <sit-down> = 0.000000                 skill <sit-down> = 0.000000
skill <fall-down> = 6.610422                skill <fall-down> = 3.232686
skill <pick-up-glass> = 10.521286           skill <pick-up-glass> = 5.145211
skill <put-down-glass> = 0.585858           skill <put-down-glass> = 0.286501
skill <crawl-to-door> = 93.590080           skill <crawl-to-door> = 45.768242

skill <crawl-to-door> has been determined to be active.

resetting threshold to 45.000000

I'm checking to see if any previously marked disabled agents need to be amputated...

Evaluating command:  ASNR-update-sensors-values {dude-is-at-door T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = dude-is-at-door
                value = T
    sk_c list:
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
        sk_c member:
                skill name = crawl-to-door
                pv pair:
                        proposition = dude-is-at-door
                        value = F
    sk_a list:
        sk_a member:
                skill name = crawl-to-door
                pv pair:
                        proposition = dude-is-at-door
                        value = T
    sk_d list is empty.

Evaluating command:  ASNR-mark-skill-as-completed crawl-to-door

just reset skill <crawl-to-door>'s activation level to 0.0
```

```
**** Time Step [36]:
I'm marking any agents which I find to be disabled...
sending [10.000000] activation to skill <locate-door> from sensor <door-is-viewable>.
... ... ...
Skills' activation level before decay:        Skills' activation level after decay:
skill <locate-door> = 54.632996               skill <locate-door> = 35.261055
skill <open-door> = 140.747711                skill <open-door> = 90.840942
skill <close-door> = 10.109744                skill <close-door> = 6.524999
skill <stand-up> = 53.088173                  skill <stand-up> = 34.264000
skill <sit-down> = 0.000000                   skill <sit-down> = 0.000000
skill <fall-down> = 3.774741                  skill <fall-down> = 2.436282
skill <pick-up-glass> = 10.654017             skill <pick-up-glass> = 6.876282
skill <put-down-glass> = 0.735030             skill <put-down-glass> = 0.474401
skill <crawl-to-door> = 5.147136              skill <crawl-to-door> = 3.322048

skill <open-door> has been determined to be active.

resetting threshold to 45.000000

I'm checking to see if any previously marked disabled agents need to be amputated...

Evaluating command:  ASNR-update-sensors-values {door-is-open T}

the newly updated sensor looks like this:
sensor info:
    sensor pv:
        pv pair:
                proposition = door-is-open
                value = T
    sk_c list:
        sk_c member:
                skill name = open-door
                pv pair:
                        proposition = door-is-open
                        value = F
        sk_c member:
                skill name = close-door
                pv pair:
                        proposition = door-is-open
                        value = F
    sk_a list:
        sk_a member:
                skill name = open-door
                pv pair:
                        proposition = door-is-open
                        value = T
    sk_d list:
        sk_d member:
                skill name = close-door
                pv pair:
                        proposition = door-is-open
                        value = F

Evaluating command:  ASNR-mark-skill-as-completed open-door

just reset skill <open-door>'s activation level to 0.0
```

Discussion: Dude Take 3

As hoped, this example showed the implementation's ability to robustly deal with failure, and showed how I already have implemented hooks for learning

by example. Unfortunately, in addition to highlighting the implementation's good points, it also uncovered a problem that I hadn't expected. Up until now, I had the utilized the notion of the pv-pair in a rather limited way—the only value I ever used was T or F. I realized after this example that the action selection algorithm, as currently stated, can only deal with true propositions. The problem is tied up in the notion of the add-list and the delete-list vs. the condition-list. In a sense, the condition-list defines a set of predictions about the state of the world before a skill agent executes. The add-list and the delete-list define a set of predictions about the state of the world after a skill agent executes. The problem stems from the fact that the condition-list implictly accepts only true propositions; the negation of a proposition must be explicitly stated. In other words, if you define a member of the condition-list for the skill `walk-to-door` as `{dude-at-door F}`, no activation gets sent due to that pv-pair. The add-list and delete-list explicitly point out the proposition which will added to and deleted from the state of the environment; they implicitly imply the existence of a proposition as T and the absence as F. This is a subtle point I missed for a long time. It doesn't mean pv-pairs are a bad idea altogether; as I'll discuss in Chapter 6, if we extend the idea of sensor agents to include receptor agents (see **Chapter 6: Different Kinds of Agents**), we can keep the extensibility I originally intended when I thought to use pv-pairs.

6 Final Remarks

Limitations, Possible Improvements, Conclusion

in the future...

the single most commonly used expression at the Media Lab

While Build-a-Dude is a fully functioning action selection system, as promised, there are certain limitations that need to be pointed out. There is also still plenty of room for improvement. In this final chapter, I'll discuss possible directions for future research and development of the system; both work that is going on right now, and things I'd like to see done at some point. I'll discuss limitations of the current system, and finish up with a statement of what this thesis has accomplished.

Improving the Backend

Implementing Graphically Simulated Motor Skills

One of the areas that I originally hoped to be able to make more progress than I have is in implementing graphically simulated motor skills. Unfortunately, the distributed implementation of the action selection algorithm proved to be enough of an endeavor that I have only recently reached the point where I could realistically begin to work on this problem. With the advent of such systems as Dave Chen's 3d (**Chen 1991**), that allows easy experimentation of graphically simulated kinematic and dynamic motor skills, I hope to be able to make significant headway in building up a library of motor skills amenable to dude construction. Also, the work being done by Bruce Blumberg on a simulated physics toolkit on the NeXT machine in the spirit of the Interface Builder promises to make the task of creating dynamically simulated motor skills a much simpler one (**Blumberg 1990**). Also, I have been in contact with Martin Friedman of the Vision & Modeling Group here at the MIT Media Lab about possibly using the simulation system, ThingWorld, to implement certain classes of motor skills (**Friedman 1991**). All of these systems will be considered during the coming year as possible candidates for graphically simulated motor skill implementations.

(Chen 1991)

Chen, D.T. *Pump it Up: Simulating Muscle Shape from Skeleton Kinematics, A Dynamic Model of Muscle for Computer Animation*, Ph.D. thesis, Massachussetts Institute of Technology, (in preparation.

(Blumberg 1990)

Blumberg, B. personal communication, (1990).

(Friedman 1991)

Friedman, M. personal communication, (1991).

Improving the Frontend

Natural Language

Obviously, if we are to reach the point of being able to interact with Build-a-Dude's creatures by saying "What happened, dude?", we have to consider the question of a natural language front-end. The work being done by Strass-

mann here at the MIT Media Lab, with his Divadlo system (**Strassmann 1991**), will be bear watching over the next few months. It promises to bring the accessibility and power of natural language to the animation environment. The software he is using for natural language, Huh (**Haase 1990**), was written by Ken Haase, also here at the MIT Media Lab. Haase's group, which is working on story understanding, are also working on problems which could be used to improve the front end of such a system as Build-a-Dude, and merit close watching.

(Strassmann 1991)

Strassmann, S. *Desktop Theater: A System for Automatic Animation Generation,* Ph.D. thesis, Massachusetts Institute of Technology, (in preparation).

The NeXT Machine: Interface Builder, Improv and Mach

I am currently pursuing acquiring a NeXT machine to further develop Build-a-Dude on. I believe that the NeXT machine offers a unique platform as a networked Build-a-Dude frontend. First of all, the construction of an agent browser would be an invaluable aid for observing the execution of a skill network. The ability to visualize the network and graph activation flow would be useful in both debugging and constructing networks. Maes had a simple activation level graphing mechanism on the Symbolics Lisp Machine, but I am interested in building a much more comprehensive network/agent browser. I spent quite a bit of time with Stardent's AVS system, and also looked briefly at Paragon Imaging's Visualization Workbench, but found both lacking in the tools I needed to construct such a browser. From my current understanding of the NeXT Machine's Interface Builder, I believe that it offers a unique platform to build such a browser and, eventually, a network/agent editor.

(Haase 1990)

Hasse, K. *Huh Internals Manual,* Massachusetts Institute of Technology Media Lab, (1990).

Another possibility provoked by the notion of moving to a NeXT machine involves adding statistically-based learning methods to the network, similar to what Maes and Brooks did for the robot Ghengis (**Maes 1990B**). Improv, a financial modeling and analysis program from Lotus, runs on the NeXT, and offers the potential as a compute engine for doing statistically based analysis without having to write the code. Also, Mathematica (**Wolfram 1988**) also runs on the NeXT, and offers similar capability for this particular application.

(Maes 1990B)

Maes, P. and Brooks, R. A. *Learning to Coordinate Behaviors,* Proceedings of AAAI-90, (1990).

(Wolfam 1988)

Wolfram, S. *Mathematica: A System for Doing Mathematics by Computer,* Addison-Wesley, (1988).

Finally, the operating system running on the NeXT machine is Mach, which is in essence a very clean rewrite of 4.3 BSD UNIX with built-in hooks for message passing and transparent, kernel managed distributed processing. Given that I wrote my own portable messaging passing library (see **Appendix B**), I could easily add extensions to it to take advantage of Mach's message passing.

Improving the Inside

Increasing Parallelism

I am currently experimenting with multi-threaded processes, where each thread is an independently executing agent managed by the asna. This is to take advantage of local parallelism available on multi-processor workstations, in addition to the network level parallelism I have already implemented using multiple workstations.

Another area I am looking into is implementing the "registry" part of the registry/dispatcher as a form of networked shared memory. Such commercially available distrbuted computing paradigms as **Linda** might be appropriate to further hide the distributed nature of the implementation from future users.

Implement Template Skills

The current implementation does not support parameterized motor skills, or what I referred to as *template skills* in Chapter 3. This is a serious drawback, since it requires each skill to be bound to the execution details at definition time. For example, I must define a skill `pickup-the-paint-sprayer` rather than a more general skill `pickup-object-X` where the name and location of `object-X` are filled in at runtime. Motor units in general are shared, which means they must be parameterized somehow so that I can invoke the same motor units for different purposes. This is a central notion in ongoing research in movement physiology and psychology on generalized motor programs. Additionally, any future implementation that uses graphical simulation systems to control to simulated geometry of a dude would be useful for a large set of motor skills. Dave Chen's 3d system, for example, could be used for a large class of inverse kinematic and forward dynamic skills such as reaching and grasping, and there is no reason why a large set of motor skills could not communicate to a single invocation of 3d, rather than each managing its own.

More interesting and relevant behavior could be generated by the addition of coefficients on the flow of activation from each agent. This would allow, for example, one goal agent to make itself more important to the overall decision making process than another goal agent. This would also allow the various relationships among skill agents (sucessor, predecessor, conflicter, follower) to vary in importance to each other, and allow skill agents to vary in importance to each other. Such periodic control strategies as circadian rhythms could be introduced into the behavior of dudes in a straightforward manner. Finally, it would allow for experiments with faulty or noisy sensor data, by allowing the virtual actor to weight how useful it thought a particular sensor was, by adjusting the coefficient of activation flow from it.

threads

Threads are lightweight UNIX processes, each of which can potentially run on its own physical processor. Currently available machines from Stardent, SGI and NeXT support threads.

Linda

Linda supports the notion of a networked shared memory, called *tuple space*, where processes can put things into, take things out of, read from, and write to tuple space. Linda is commercially available for several platforms. See (**Gelernter 1988**) or (**Carriero 1990**) for more details.

(Gelertner 1988)

Gelertner, D. *Getting the Job Done* Byte, (November, 1988).

(Carriero 1990)

Carriero, N., and Gelertner, D. *How to Write Parallel Programs* MIT Press, (1990).

A similar extension I have planned is to allow each skill agent to maintain its own activation decay function. This would allow skill agents to have activation levels which would decay differently. Classic displacement behavior could perhaps be simulated using skill agents whose activation decayed very slowly. Note that most of the aforementioned capabilites already exist in the current implementation of Build-a-Dude; there just hasn't been enough time to conduct the myriad of experiments possible with the current toolkit.

Different Kinds of Agents

The addition of agents other than goal, sensor, and skill agents is an interesting possibility. Maes' has suggested the notion of *perception* agents, which would combine most of the features of my goal and sensor agents (**Maes 1990C**). Agents which manipulated the flow of activation between agents is also a possibility. Finally, agents that acted as critics, that gave advice to skill agents on their condition, add, and delete lists, as well as more forceful agents, that could actually go in and change an agent are also under consideration.

(Maes 1990C)

Maes, M. personal communication, 1990.

Finally, one idea that I will implement very soon is the notion of a *receptor* agent. Although the original design (and the current implementation) allow sensor agents to measure the state of the world as pv-pairs, this proved too much flexibility. The reason has to do with the way the current algorithm deals with condition, add, and delete-lists. As discussed in Chapter 5 (**Discussion: Dude Take 3**), the way the skill network is constructed makes assumptions about the existence of a proposition as reason enough to construct certain links, and it makes no sense for the sensor agents to do more than report on the existence or not of a given proposition. This leads to the notion of a *receptor* agent. A receptor agent would be an agent that measures some set of continuous quantities in the world; what I referred to as **signals** in Chapter 4. No processing is done by a receptor agent; it merely reports signal strength every timestep. Sensor agents would have connections to an arbitrary number of receptor agents, and could do an aribrtrary amount of processing on the signals they received. The difference would be that the sensor agent would present only a true or false proposition to the network, and goal agents and skill agents would subsequently only use these propositions. Sensor agents would deliver **signs** to the registry/dispatcher that it would then use to construct the skill network.

signs & signals

Signals represent sensor data—e.g. heat, pressure, light—that can be processed as continuous variables. Signs are facts and features of the environment or the organism. (**Rasmussen 1983**)

(Rasmussen 1983)

Rasmussen, J. *Skills, Rules and Knowledge; Signals, Signs and Symbols and other Distinctions in Human Performance Models* IEEE Trans. on Systems, Man, and Cybernetics, vol. SMC-13, no. 3 (May/June 1983).

Agents as Agencies

In the current implementation, every agent (goal, sensor, skill) has a unique two-way communication connection link to the regsitry/dispatcher. This has

simplified the implementation, since there is no question of who a message is from on a given communication path. Unfortunately, I have run into some hard limits in UNIX networking software that have caused me to rethink this philosophy. In a desire to build a more robust, portable system, I have decided to allocate a communication link per agency, instead of agent, where *agency* is defined as per Minsky (**Minsky 1987**). This will have several advantages. The first, and most obvious to the current users of the system, will be the freedom to construct networks of much larger size (i.e. many more agents). Currently, I am limited to less than 25 agents on the machine that I ususally run on, a Stardent Titan 1500. This also promises to increase performance slightly due to certain low-level implementation issues. The cost of this added functionality and performance comes at a relatively inexpensive addition to the message passing protocol, that now will need to include a source and destination tag so that it's obvious which agent sent a message and which agent a message was intended for. This also might be alleviated by moving to a NeXT machine, or other Mach based system, at the cost of portability. Either way, it will pave the way for a much more powerful and extensible system.

agency

An assembly of parts considered in terms of what it can accomplish as a unit, wothout regard to what each of its parts does by itself.
(Minsky 1987)

(Minsky 1987)

Minsky, M. *Society of Mind*, Simon Schuster, 1987.

Learning

Learning could be introduced by allowing the virtual actor to change the composition of its skill network. I plan to do this with the addition of new types of agents which observe the flow of activation in the skill network and edit links between agents. I'm also very excited by the positive results Maes and Brooks have obtained using statistical methods for learning in action selection networks. As discussed above, a move to a platform such as NeXT with accessibl, robust, statistical analysis software makes the prospect not only possible but perhaps pleasant. Learning by example is yet another learning method that is interesting, especially in the context of dudes in virtual environments interacting with human users.

some thoughts on learning

Without learning, life is but an image of death.
Cato

Learning without thought is useless; thought without learning is dangerous.
Confucius

Conclusion

I have presented the ubiquitous problem of selecting the next appropriate action in a given situation, with an emphasis on its application for a computational autonomous agent in an animation/simulation system. I reviewed the relevant literature. I described an algorithm for action selection, derived from one originally presented by Maes and Zeltzer. I described extensions to this algorithm, with an emphasis on efficient distributed implementations of the algorithm. I presented a parallel distributed implementation which encompasses both the original algorithm and many of my proposed extensions. I informally verified that the implementation satisfied the mathematical model, and gave several detailed examples of the implementation in use with an

emphasis on showing the extensions I have made. I discussed some of the limitations of the current theory, the implementation, and current and future directions of this work toward alleviating some of these problems.

Tinsley Galyean, a fellow graduate student here in the Computer Graphics & Animation said it best: "You know, we must really love graphics, since we get to spend so little time doing it." The work in this thesis has run the gamut from animal behavior research to researching TCP/IP sockets, from language design, to looking into the state-of-the art in parallel distributed processing. I came to this thesis as a computational graphicist with an AI background; I leave it a sadder and wiser UNIX systems hacker. I now know a reasonable amount about animal behavior as reported in the literature. I've gained a lot of practical knowledge about distributing computation over UNIX networks. I learned little new about graphics compared to what I learned about building software systems. Don't get me wrong—I consider that a good thing. Pretty pictures that matter are always the result of an amazing amount of behind-the-scenes, successful systems design.

Build-a-Dude is a large system, and will continue to grow as I continue my research towards a Ph.D. This thesis document represents a snapshot of the system and some of the ideas I've come up with as of January 1991. What I've accomplished is significant, if only because it represents a real step towards implementing a system which considers all of the factors in trying to build the mind of a virtual actor. One of the key aspects of future virtual environment systems will be in their distributed nature. Build-a-Dude was designed with distributed processing at the ground floor, and is efficiently implemented on current workstations. Its robust nature is a sign of systems to come. In the future, all systems which control autonomous agents or mediate the components of simulation systems will be running on large numbers of machines at once, and some sort of fault tolerance like Build-a-Dude has will be standard issue on all such systems. *You can bet on it.*

Sources

Agha, G. (1985). *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA.

Agre, P. and Chapman, D. (1987). *Pengi: An Implementation of a Theory of Situated Action.* Proceedings of AAAI-87.

Badler, N.I. and B.L. Webber. (1991). *Animation from Instructions.* in *Making Them Move: Mechanics, Control and Animation of Articulated Figures.* Morgan Kaufmann.

Beer, R.D., L.S. Sterling and H.J. Chiel. (January 1989). *Periplaneta Computatrix: The Artificial Insect* Project Tech. Report TR 89-102. Case Western Reserve University.

Bernstein, N. (1967). *The Coordination and Regulation of Movements.* Pergamon Press, Oxford.

Blumberg, B. (1990). personal communication.

Brooks, R.A. (1986). *A Robust Layered Control System for a Mobile Robot.* IEEE Journal of Robotics and Automation 2:1, 14-23.

Brooks, R.A. (1989). *The Whole Iguana.* in *Robotics Science.* MIT Press, Cambridge, MA.

Carrero, N. and D. Gelertner (1990). *How to Write Parallel Programs.* MIT Press, Cambridge, MA.

Charniak, E. and McDermott, D. (1985). *Introduction to Artificial Intelligence.* Addison-Wesley.

Chen, D. (1991, in preparation). *Pump it Up: Simulating Muscle Shape from Skeleton Dynamics, A Dynamic Model of Muscle for Computer Animation.* Ph.D thesis, Massachusetts Institute of Technology, Cambridge, MA.

Friedman, M. (1991). personal communication.

Forssberg, H.S., Grillner, S., and Rossignol, S. (1975). *Phase Dependent Reflex Reversal During Walking in Chronic Spinal Cats.* Brain Research, 85, 103-107.

Gallistel, C.R. (1980). *The Organization of Action: A New Synthesis.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Gelertner, D. (1988). *Getting the Job Done.* Byte, November.

Girard, M. and A.A. Maciejewski. (July 1985). *Computational Modeling for the Computer Animation of Legged Figures.* Computer Graphics 19: 3, 263-270.

Goldberg, D.E. (1989). *Genetic Algorithms,* Addison-Wesley.

Greene, P.H. (1972). *Problems of Organization of Motor Systems.* in Progress in Theoretical Biology 2: 303-338.

Greene, P.H. (June 1982). *The Organization of Natural Movement,* Journal of Motor Behavior.

Haase, K. (1990). *Huh Internals Manual.* Massachusetts Institute of Technology, Cambridge, MA.

Maes, P. (December 1989). *How to Do the Right Thing.* A.I. Memo 1180. Massachusetts Institute of Technology, Cambridge, MA.

Maes, P. and Brooks, R.A. (1990). *A Learning to Coordinate Behaviors.* Proceedings of AAAI-90.

Maes, P. (1990). *Situated Agents Can Have Goals.* Journal of Robotics and Autonomous Systems 6: 1&2.

Maes, P. (1990). personal communication.

McFarland, D.J., and R.M. Sibly (1975). *The Behavioral Common Path.* Phil. Trans. Roy. Soc. London.

McKenna, M., S. Pieper and D. Zeltzer. (March 25-28, 1990). *Control of a Virtual Actor: The Roach. Proc. 1990 Symposium on Interactive 3D Graphics.* 165-174. Snowbird, UT.

McKenna, M. (January 1990). *A Dynamic Model of Locomotion for Computer Animation.* S.M. Thesis. Massachusetts Institute of Technology. Cambridge, MA.

Miller, G. (August 1988). *The Motion Dynamics of Snakes and Worms.* Computer Graphics 22: 4, 169-178.

Minsky, M. (1987). *The Society of Mind.* Simon and Schuster, New York.

Reynolds, C.W. (July 1987). *Flocks, Herds and Schools: A Distributed Behavioral Model.* Computer Graphics 21: 4, 25-34.

Schank, R. and R. Abelson (1977). *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Associates.

Sherington, C.S. (1906). *The Integrative Action of the Nervous System.* Yale University Press.

Sims, K. (June 1987). *Locomotion of Jointed Figures Over Complex Terrain.* S.M.V.S Thesis. Massachusetts Institute of Technology. Cambridge, MA.

Strassmann, S. (1991, in preparation). *Desktop Theater: A System for Automatic Animation Generation.* Ph.D Thesis. Massachusetts Institute of Technology. Cambridge, MA.

Tinbergen, N. (1951). *The Study of Instinct.* Oxford University Press, London.

Travers, M. (1989). *Agar: an Animal Construction Kit.* S.M. Thesis. Massachusetts Institute of Technology. Cambridge, MA.

Turvey, M.T. (1977). *Preliminiaries to a Theory of Action with Reference to Vision,* in *Perceiving, Acting and Knowing.* Lawrence Erlbaum Associates.

Wilson, S. (1987). *Classifier Systems and the Animat Problem.* Machine Learning. 2(3).

Wolfram, S. (1988). *Mathematica: A System for Doing Mathematics on Computer,* Addison-Wesley.

Zeltzer, D. and Johnson, M.B. (to appear). *Motor Planning: An Architecture for Specifyingand Controlling the Behavior of Virtual Actors.* Journal of Visualization and Computer Animation, 2(2).

Zeltzer, D., S. Pieper and D. Sturman. (June 19-23, 1989). *An Integrated Graphical Simulation Platform.* Proc. Graphics Interface '89 266-274. London, Ontario.

Zeltzer, D. (April 1983). *Knowledge-Based Animation,* Proc. ACM SIGGRAPH/SIGART Workshop on Motion.

Zeltzer, D. (May 14-16 1987). *Motor Problem Solving for Three Dimensional Computer Animation.* Proc. L'Imaginaire Numerique Saint-Etienne, France.

Zeltzer, D. (August 1984). *Representation and Control of Three Dimensional Computer Animated Figures.* Ph.D. Thesis. The Ohio State University. Columbus, OH.

Zeltzer, D. (December 1985). *Towards an Integrated View of 3-D Computer Animation.* Visual Computer 1: 4.

Appendix A
the registry/dispatcher's inner loop

Once the registry/dispatcher is started up, it goes into an endless loop. When first invoked, the registry/dispatcher has a single connection to the outside world—the asn daemon. As agents contact the daemon and then connect to the registry/dispatcher, the list of connections grows. This section examines, in some detail, what happens in this loop. Here is the C code which makes up the inner loop:

```
for(;;)
{  while  (check_for_daemon_message(&myself))
   {    service_daemon_message(&myself);
   }
   while  (check_for_unregistered_agents_messages(&myself,  &ret_fd_set))
   { service_unregistered_agents_messages(&myself,  &ret_fd_set);
   }
   while  (check_for_executing_skills_messages(&myself,  &ret_fd_set))
   { service_executing_skills_messages(&myself,  &ret_fd_set);
   }
   if  ((myself.spread_activation)  ||  (myself.spread_for_n_steps))
   { if  (myself.spread_for_n_steps)
     { myself.spread_for_n_steps--;
     }
     if  (appropriate_time_to_mark_disabled_agents(&myself))
     { mark_disabled_agents(&myself);
     }

     zero_out_intermediate_activation_levels(&myself);

     update_goals_list(&myself);
     update_sensor_list(&myself);

     spread_activation_from_sensors(&myself);
     spread_activation_from_goals(&myself);
     spread_inhibition_from_protected_goals(&myself);

     mark_executable_skills(&myself);

     spread_activation_backwards_to_predecessors(&myself);
     spread_activation_forwards_to_successors(&myself);
     spread_activation_forwards_to_followers(&myself);
     spread_inhibition_to_conflicters(&myself);

     sum_activation(&myself);

     decay_activation(&myself);

     active_skill  =  determine_active_skill(&myself);
     if  (active_skill  !=  NULL)
     { notify_active_skill(&myself,  active_skill);
       reset_threshold(&myself);
     }
     else
     { lower_threshold_by_some_amount(&myself);
     }
     if  (appropriate_time_to_check_disabled_agents(&myself))
     {    check_disabled_agents(&myself);
     }
     myself.current_time++;
   }
}
```

Well, that's a bit overwhelming, but I'll go through it line by line, and try to explain it. I must warn you, though, the following discussion assumes a rudimentary grasp of the C programming language.

First off, there is a a structure which contains all the state information for the registry/dispatcher. This is the structure `myself`, which is of the following type:

```
typedef  struct  {
        char    *name;
        char    *my_hostname;
        char    *daemon_hostname;
        int     port;
        int     fd;
        struct sockaddr_in      registry_server_info;
        unsigned long   current_time;
        int     spread_activation;
        int     spread_for_n_steps;
        int     mark_every_n_steps;
        int     check_every_n_steps;
        struct timeval  disabled_timeout;
        struct timeval  amputated_timeout;
        int     timesteps_to_amputation;
        int     next_port_to_try;
        Tcl_Interp       *daemon_interp;
        int     current_unregistered_agent_index;
        Tcl_Interp      *unregistered_agent_interp;
        Tcl_Interp      *goal_interp
        Tcl_Interp      *sensor_interp;
        Tcl_Interp      *skill_interp;
        FILE    *log_fp;
        LIST    *unregistered_agent_list;
        LIST    *goal_list;
        LIST    *sensor_list;
        ASNR_skill_info_t       *tmp_skill;
        ASNR_proposition_value_t        *tmp_pv;
        LIST    *skill_list;
        float   pi;
        float   initial_theta;
        float   theta;
        float   phi;
        float   gamma;
        float   delta;
} ASNR_state_t;
```

The general notion is that we can think of this state structure *as* the registry/dispatcher, since it contains all the information composing the registry/dis-

patcher. That's why the instantiation of this structure in the main program is called `myself`. Most of the routines called in the inner loop get handed a pointer to this state structure, so it's important that you have some notion of what it contains. Several of the types of variables defined in the state structure are probably unfamiliar to the average reader, and merit further explanation. Keep in mind that this is not intended to be an exhaustive treatment of the implementation, just an explanation of the registry/dispatcher's inner loop, so I'll confine myself to details pertinent to that.

```
while  (check_for_daemon_message(&myself))
{    service_daemon_message(&myself);
}
while  (check_for_unregistered_agents_messages(&myself,  &ret_fd_set))
{    service_unregistered_agents_messages(&myself,  &ret_fd_set);
}
while  (check_for_executing_skills_messages(&myself,  &ret_fd_set))
{    service_executing_skills_messages(&myself,  &ret_fd_set);
}
if  ((myself.spread_activation)  ||  (myself.spread_for_n_steps))
  {
```

By default, the registry/dispatcher just services messages from its network connections to the asn daemon and the agents, rebuilding its internal network of connections between the agents each time a new agent registers with it. Initially, it doesn't spread activation between the nodes. Once the registry/dispatcher receives a "start-spreading-activation" message or "spread-activation-for-n-steps" from one of the agents or the daemon, it takes the appropriate action and begins spreading activation, either continuing until it gets a "stop-spreading-activation" message or it has spread for n steps.

```
if  (myself.spread_for_n_steps)
{    myself.spread_for_n_steps--;
}
```

If the registry/dispatcher has received a message to spread activation for some n time steps, it decrements its counter of how many steps left appropriately.

```
if  (appropriate_time_to_mark_disabled_agents(&myself))
{    mark_disabled_agents(&myself);
}
```

A very real consideration in this implementation is that of robustness and graceful degradation when parts of the skill network fail. For this reason, the

registry/dispatcher maintains a notion of the reliability of the network connections it has to all the agents. When spreading activation through the skill network, the registry/dispatcher needs to take care that it doesn't waste its time spreading activation to a skill agent that, once it's called, has actually been disabled or dead for some time. The intuitive idea is that we don't wish to "waste our time" sending activation energy to eventually activate some skill, which, when the registry/dispatcher sends it a message to execute it, it isn't able to receive the message (the network or the machine is down), or it isn't able to execute the message (the process has died or is busy doing something else). The analog situation on a real robot is that the planner doesn't want to try to utilize some manipulator which is broken, or temporarily disconnected (perhaps being repaired). An agent is considered disabled if it doesn't acknowledge receipt of a message from the registry/dispatcher in some reasonable amount of time (where "reasonable" is obviously network and context dependent). An agent is considered dead if the registry/dispatcher gets an error sending a message to that agent. Therefore, at an "appropriate time", the registry/dispatcher sends a message to each agent, and marks it disabled if it doesn't respond as described above or amputates it if the registry/dispatcher gets an error sending or receiving a message from that agent. "Appropriate time" is a value which the registry/dispatcher determines based on how many agents have become disabled or dead over time.

The registry/dispatcher starts off with some default notion of the reliability of the network connection between the agents and itself. Over time, the registry/dispatcher can change its assessment of this situation by noticing that either (1) the network is very reliable because it has not lost any connections to any agents, or (2) the network connection is rather unreliable because it has lost connections to some agents. If (1), the registry/dispatcher can decrease the frequency with which it marks the disabled agents. If (2), the registry/dispatcher can increase the frequency. As time goes on, the registry/dispatcher is free to revise this opinion, either up or down, depending on if any agents become disabled or die. Note that a disabled agent will not participate in the spreading or receiving of activation, although its existence will still affect the flow of activation, since it will still be a member of all the lists in the database.

```
zero_out_intermediate_activation_levels(&myself);
```

Next the intermediate activation accumulators are cleared. These are variables that each skill maintains that correspond to the activation received from the various links in the action selection network. They exist for the future ability to keep statistics on how much activation was received from the vari-

ous links. These statistics could be used by the skill agents to modify their pre- and post-conditions, thereby enabling a virtual actor to learn.

```
update_goals_list(&myself);
```

The goal agents are checked for any messages updating their state. The intuitive idea is that a goal agent represents more than just its desired proposition-value pair, i.e. that it can change its mind. If the goal agent is masking a user, she might change her mind given the current state of what the virtual actor is doing, and change her goals via the goal agent. If a goal agent is masking a skill from another network, it might no longer be active, and so the goal is withdrawn. Goals could also mask higher level skills, for example, a path planning skill which is in some other skill network. The path planner plans a path and knows what steps to take to navigate a collision free path in space. It in turn passes these directions as goals over time to another network composed of lower level motor skills of the dude (turn left... go straight, ..., go left)

```
update_sensor_list(&myself);
```

The sensor agents are checked for any messages updating their state. The intuitive idea is that the sensor agents deliver a new value to the registry/dispatcher everytime the proposition they measure changes. The registry/dispatcher can accept one message per time step from each sensor agent.

```
while (check_for_executing_skills_messages(&myself, &ret_fd_set))
   service_executing_skills_messages(&myself, &ret_fd_set);
```

Any currently executing skill can send a message to the registry/dispatcher and expect it to get serviced at a relatively high priority. The registry/dispatcher loops over its list of currently executing skills, servicing their messages until their are no more outstanding. The intuitive idea is that the currently executing skill(s) have priority over the selection of new actions to take, so therefore they can monopolize the resources of the registry/dispatcher if they desire. The most common message to be received from an executing skill is one to the effect that the skill has completed. Once a skill completes, the registry/dispatcher does an analysis of the state of the world as predicted by that skill's add and delete-list and the state of the world as currently measured by the sensors. The registry/dispatcher then resets the skill's current activation based on the delta between those two. The intuitive idea is that if the skill was completely successful (i.e. its add and delete-list were a correct prediction of what the world would be when it finished), the skill's current activation would be reset to zero since it had accomplished exactly what it set out to do. If, however, the skill had completely failed (i.e.

none of the proposition value pairs predicted by the skill's add and delete list exist in the world as measured by the sensors), the skill's current activation should be very close to its current value. The reason for this is that the next skill the registry/dispatcher is likely to choose to execute would be this one. Given that a skill supplies a maximum-number-of-invocations value for itself to the registry/dispatcher, and that the registry/dispatcher maintains information about how many times a skill has been called in succession, the weighting looks something like this:

```
a = this skill's current activation;
max = maximum number of consecutive invocations of this skill;
/*
   think of this as the hysterisis associated with the skill.
  Unfortunately, this is not only skill dependent, it is also domain dependent.
 */
cur = current number of consecutive invocations of this skill;
true = how many predictions made which are currently measured true;
made = how many total predictions made by this skill;
/* the sum of the lengths of the add list and the delete list */
 if (cur < max)
{  a *= (1.0 - ((max/(max - cur)) * (true/made)));
else
{  a = 0.0;
}
```

A possible extension to this would be to allow differing coefficients on each member of the add list and the delete list. This would allow for situations where one or more of the predictions was very important, while others would be less so.

```
spread_activation_from_sensors(&myself);
spread_activation_from_goals(&myself);
spread_inhibition_from_protected_goals(&myself);
```

Activation, both positive and negative, is calculated from the sensors, and the goals, the protected goals, and put into the intermediate accumulators for each skill agent.

```
mark_executable_skills(&myself);
```

Any skill which has all have the proposition-value pairs in its condition list matching the currently measured values by the corresponding sensor agents is marked "executable". This will affect which skills spread activation forward and backward.

```
spread_activation_backwards_to_predecessors(&myself);
spread_activation_forwards_to_successors(&myself);
spread_activation_forwards_to_followers(&myself);
```

```
spread_inhibition_to_conflicters(&myself);
```

Activation, both positive and negative, is now spread from each skill agent to each of the members of its predecssors, successors, followers, and conflicters.

```
sum_activation(&myself); decay_activation(&myself);
```

All the intermediate activation accumulation values are summed, and decayed by some amount. Currently, the registry/dispatcher conserves the sum of activation energy in the system.

```
active_skill = determine_active_skill(&myself);
if (active_skill != NULL)
{  notify_active_skill(&myself, active_skill);
     reset_threshold(&myself);
}
else
  {    lower_threshold_by_some_amount(&myself);
}
```

The active skill is selected. If a skill is selected (all its preconditions are met, its activation is higher than all other skills, it is exectuable, it is not disabled, it is not executing, and its activation level is higher than the threshold), it is sent a message to start executing, and the threshold value is reset. If no skill was selected, the threshold is lowered by some amount (user settable).

```
if  (appropriate_time_to_check_disabled_agents(&myself))
{  check_disabled_agents(&myself);
}
```

If the time is appropriate, the registry/dispatcher sends a message to each of the agents it has marked as disabled. If the disabled agent still doesn't respond, or the communication channel is corrupted, the agent is amputated. The "appropriate time" is context dependent, and could range from every time step to never.

```
myself.current_time++;
```

The current time counter is incremented, and the loop continues.

Appendix B
appcom: an application communication library

In order to implement the algorithm described in this thesis, I found it necessary to design and implement some sort of message passing library. I strongly would have preferred to use an existing package, but unfortunately could find none which satisfied my criteria:

- It should be portable to different vendors' workstations.
- Source code must be freely available.
- All functionality should be acessible from the C language.
- It should allow for message passing over a network (i.e. from one workstation to another) transparently.
- It should impose a minimum of performance overhead on the calling application.

The obvious choice was Berkely sockets, since they are supported on all machines running BSD derived implementations of UNIX, and are supported as "Berkeley extensions" on most System V machines. Another advantage of using sockets was the fact that synchronous (i.e. blocking) and asynchronous (i.e. nonblocking) communication was built-in since sockets were uniquely identified with real UNIX file descriptors, so calls to **fcntl**() could be used to make them non-blocking or blocking at will, just like any other UNIX file. Unfortunately, using sockets effectively is somewhat daunting for many UNIX programmers, and their use tends to be a nontrivial addition to any application. The other problem with sockets is that they are always (to my knowledge) implemented as kernel extensions, and a **read**() or **write**() to a socket involves a call to the kernel (i.e. a significant performance cost). If one was to use them, and use them with abandon, some sort of buffering would need to be done to ensure that low level reads and writes were done only when necessary.

fcntl()

a standard low-level UNIX routine for manipulating the attributes of a file. It is useful in this context since you can use it to change the manner in which the **read**() or **write**() routine deal with a socket. If the socket's attributes are set to *non-blocking*, each routine will return immediately if there is no data available. If the socket is set to *blocking* (the default), the routine will wait until the requested data is available.

Well, given that sockets could serve as the transport layer, there was still the question of what format the message should take. UNIX provides message passing, but it is very limited, both in number of messages allowed and length of messages. Also, there is no standard message passing available in UNIX that is network transparent. Mach, the operating system with a UNIX com-

patible kernel used on the NeXT machine, does provide such a facility, but remember, I wanted this package to be portable to many machines.

Since such a complete package which met my requirements was not available, I designed and implemented my own. As one would imagine, I based my message structures on UNIX's built-in ones, but made it more extensible and cleaner. For my transport layer, I chose sockets running over TCP/IP, since this allowed me to communicate locally or over the net with the same calls, giving me network transparency. The library evolved from my particular needs, and is not really comprehensive. Having said that, I wrote this library five months ago, and have only added a few routines two months ago, and haven't needed to add or change any software since. In addition to the software written for this thesis, this library has been used for several distributed visualization applications written for the Connection Machine 2 System. This library has been compiled and used on the following UNIX workstations: HP 9000-835, HP 9000-350, SGI 240GTX, Stardent Titan 1500, SUN 4/370, VAX 6700.

The library is neither elegant or complete; but it is portable, functional, efficient, and has served the needs of its users (there are a few other people that use this library; the list is actually growing). There are basically three sets of public functions; those dealing with the setting up of a connection between a message passer/receiver and its companion message receiver/passer, and those dealing with the sending and receiving of messages, an error handling/reporting facility. Since the communication model is based on Berkeley sockets, it imposes the same sort of client/server model that sockets have.

personal C style note

for all libraries I write, I choose some short set of letters which are then capitalized and used for all typedef's, public defines, and public routine names. Internal routines are usually prefaced with `XXXI_`, where `XXX` is the chosen identifier for public routines. The appcom library uses `AC_` for its preface.

the public defines are defined are listed below:

```
#define  MAX_AC_MSG_DATA_LENGTH      (4096 - 16)
#define  AC_MSG_HEADER_SIZE  \
              (sizeof(AC_msg_t)  -  MAX_AC_MSG_DATA_LENGTH)
#define  AC_MSG_TYPE_BLOCKING                1
#define  AC_MSG_TYPE_NONBLOCKING             2
#define  AC_MSG_TYPE_STRING                  4
#define  AC_MSG_TYPE_REPLY_EXPECTED          8
#define  AC_ERRNO_NOT_SET                    0
#define  AC_ERRNO_WRITE_FAILED               1
#define  AC_ERRNO_READ_FAILED                2
#define  AC_ERRNO_READ_HEADER_TOO_SMALL      3
#define  AC_ERRNO_WROTE_HEADER_TOO_SMALL     4
#define  AC_ERRNO_READ_DATA_TOO_SMALL        5
#define  AC_ERRNO_WROTE_DATA_TOO_SMALL       6
#define  AC_ERRNO_DATA_TOO_LARGE             7
#define  AC_ERRNO_DATA_TYPE_MISMATCH         8
#define  AC_ERRNO_UNKNOWN_HOST               9
#define  AC_ERRNO_NO_STREAM_SOCK             10
#define  AC_ERRNO_BIND_FAILED                11
```

```
#define  AC_ERRNO_LISTEN_FAILED          12
#define  AC_ERRNO_ACCEPT_FAILED          13
#define  AC_ERRNO_CONNECT_FAILED         14
```

personal C style note

All typedef's end with `_t`, to emphasize the fact that they are indeed typedefs.

the public data structures which are defined are listed below:

```
typedef  struct
{   unsigned  int      length;
    unsigned  int      type;
    char               data[MAX_AC_MSG_DATA_LENGTH];
}  AC_msg_t;

typedef  struct
{   int        fd;
    int        need_to_swap;
}  AC_object_t;
```

Communication initialization:

```
int
AC_tcp_socket_server_setup_and_connect(port);
int port;

int
AC_detailed_tcp_socket_server_setup_and_connect(port,
                                          server_info,
                                          client_info,
                                          sockfd)
int  port;
struct  sockaddr_in   *server_info,
                      *client_info;
int  *sockfd;

int
AC_detailed_tcp_socket_server_setup_and_bind(port,
                                          server_info,
                                          sockfd)
int  port;
struct  sockaddr_in  *server_info;
int  *sockfd;

int
AC_detailed_tcp_socket_server_bind(port,
                                   server_info,
                                   sockfd)
int  port;
struct  sockaddr_in  *server_info;
int  *sockfd;

int
AC_detailed_tcp_socket_server_setup2(server_info)
struct  sockaddr_in  *server_info;

int
AC_detailed_tcp_socket_server_connect(client_info,
                                          sockfd)
struct  sockaddr_in  *client_info;
int  sockfd;
```

```
int
AC_tcp_socket_client_setup_and_connect (hostname, port)
char *hostname;
int port;
```

Message passing/receiving routines:

```
int
AC_write_msg_to_fd (msg_ptr, fd)
AC_msg_t *msg_ptr;
int fd;

int
AC_read_msg_from_fd (msg_ptr, fd)
AC_msg_t *msg_ptr;
int fd;

int
AC_write_str_as_msg_to_fd (str_ptr, fd)
char *str_ptr;
int fd;
char
*AC_write_str_as_msg_to_fd_with_reply_p (str_ptr, fd, reply_p)
char *str_ptr;
int fd;
int reply_p;

char
*AC_read_str_as_msg_from_fd (fd)
int fd;

char
*AC_read_str_and_reply_p_as_msg_from_fd (fd, reply_p)
int fd;
int *reply_p;
```

error handling/reporting:

```
int
AC_perror (str)
char *str;

int
AC_fperror (fp, str)
FILE *fp;
char *str;
```