DRAFT

WavesWorld: A Parallel, Distributed Testbed for Autonomous Animated Characters

This paper discusses the software comprising WavesWorld. It traces the history of the system, from its inception in the author's SMVS thesis up to its current state. Various aspects of the testbed are discussed, and an example is given to illustrate how it works and is used.

The intended audience is a reader interested in a broad technical overview of WavesWorld, as well as a potential technical user of some or all aspects of it.

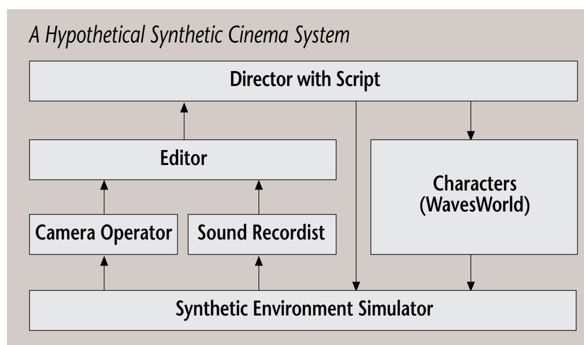
Michael B. Johnson **wave@media.mit.edu**

I: Introduction

WavesWorld is a collection of software for designing, building, and debugging autonomous animated characters. It is intended to be used to quickly build a graphically simulated virtual actor which can act autonomously in a networked virtual environment. It is part distributed artificial intelligence and part virtual environments, part parallel programming and part visual programming, with more than a liberal dash of synthetic cinema tossed in. This paper is intended to be a broad overview of the WavesWorld system — the ideas behind it, some new ideas that have come out of it, and some examples showing how it works.

1.1 What WavesWorld Is Not

It is important to realize that WavesWorld is *not* a a task-level animation system[1] or a wholly realized synthetic cinema system[2]. Such a system would have a notion of how to edit image and sound sequences in order to tell a story. It would have facilities for moving from a natural language script to a completed animation with both sound and images. For example, here's a high-level diagram of such a system:



WavesWorld is intended to be a testbed to explore issues surrounding how to build autonomous characters that might be integrated into such a system. Using WavesWorld, I hope to understand the requirements such characters would place on the other parts of the system. WavesWorld provides a way of experimenting with characters that can then be used to tell stories. A user does not use WavesWorld to tell stories; rather they take the characters built using the system and bring them into their own narrative construction system. WavesWorld provides one of the necessary building blocks to building a task-level animation or synthetic cinema system.

1.2 Why WavesWorld Is Interesting

WavesWorld is notable on several accounts, in areas ranging from reactive planning to visualization techniques. These areas include:

- *an extended reactive planner*
- *a modular perception model with variable sampling rates*
- *a novel approach to collaboratively building autonomous animated characters*
- *a computational economic model built in at the lowest level*
- *full integration into an advanced commercial development environment*
- *a sophisticated process model amenable to asynchronous MIMD programming*
- *facilities for trading computational power for communications bandwidth*
- *a powerful set of multi-modal debugging tools*

WavesWorld uses the planning algorithm originally developed by Maes[3], significantly extended to deal with asynchronous action selection and parallel skill execution. In addition, a comprehensive sensing structure has been added to the planner which complements the high level controls the reactive planner gives by allowing time varying sampling rates to be neatly integrated into the perceptual mechanisms.

WavesWorld emphasizes the collaborative nature of the character construction and animation process. It provides tools for building characters out of **malleable media**. Malleable media are designed to be reshaped,

1. Zeltzer, D. *Task-level Graphical Simulation: Abstraction, Representation, and Control* in "Making Them Move", ed. Badler, N.I., Barsky, B.A., and Zeltzer, D., Morgan Kaufmann, Palo Alto, CA, 1991.
2. Drucker, S.M., Galyean, T.A., Zeltzer, D. *CINEMA: A System for Procedural Camera Movements* in Proceeding of ACM SIGGRAPH 1992 Symposium on Interactive 3D Graphics, Cambridge, MA. March 29-April 1. Special issue of Computer Graphics. pp 67-71.
3. Maes, P. *How to Do the Right Thing*, Connection Science, Vol. 1, No. 3, 1989.

appropriated, recombined with other media, and finally reconstructed into larger pieces which may bear only passing resemblance, if any, to their original source.

The system integrates Malone's computational economics[4] at the lowest level of the system and all processes in WavesWorld are "bid" on by a disparate set of computational resources at run-time.

WavesWorld has been seamlessly integrated into arguably the most advanced commercial software development environment, NEXTSTEP[5]. This integration allows developers to take advantage of a wide range of high quality commercial and academic software. This has a significant impact on the "quality of life" of the WavesWorld programmer, facilitating rapid prototyping and an implementation level impossible in other systems.

II: Motivation

This work stems from a long cherished idea of being able to experiment with building characters — autonomous animated simulacra who actively sense and interact with a virtual environment. I am interested in the notion of ubiquitous character animation — lots of characters, generated by many people, being exchanged, cut up, reassembled and appropriated by a networked community of artisans.

Animation today remains basically the painful process it has always been. The computer is just beginning to become the tool that computational graphicists have always said it could be. I wanted to accelerate that process, and combine animation with parallelism and distributed computing to begin to explore some of the interesting social dynamics that such a system would make possible.

The question of building an animation system that allows autonomous animated characters to be built is a difficult and complex one. From a computational graphicist's point of view, there are many compelling issues (modeling the geometry, light, shading and behavior of a character and its world) as well as a number of more mundane ones. Probably the two most mundane (again, from a computational graphicist's point of view) are:

- *infrastructure*
- *integrating sensing with planning*

Infrastructure refers to the underlying computational system that the animation system is situated in. How does it take advantage of a disparate collection of computing resources? How does it distribute tasks among the system? How does it aid the animation designer to concentrate on the task at hand? Since many of these questions are areas of current research in other fields (HCI, Distributed Artificial Intelligence, systems engineering, etc.) the issues involved in choosing an appropriate underlying computational structure to base a networked animation system is both tedious and difficult.

Integrating sensing with planning refers to the difficult issues arising from trying to hook up an animated character with its animated world. Does the character have memory? Does it have a world model? How often does it sample the world? Is sensing centrally controlled or completely distributed? Are sensing mechanisms shared? If so, how? Again, these issues are all areas of active research in the AI and A-Life communities, so it is fairly difficult to decide on a particular approach that is best.

4. Malone, T.W. et.al *Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments,* in "The Ecology of Computation", ed. B.A. Huberman, Elsevier Science Publishing, 1988.
5. NeXT Computer, Inc., "NeXT Development Tools and Techniques Release 3", Addison-Wesley, Reading, MA. 1993.

DRAFT

III: First There Was Build-a-Dude

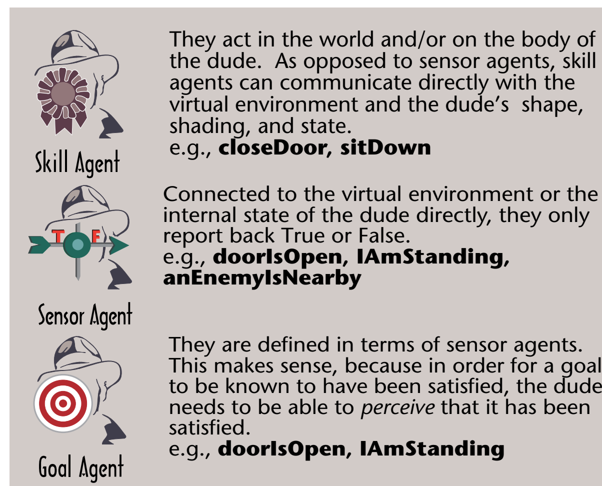
My earlier work, **Build-a-Dude**[6], attempted to address these issues. The central idea behind that system was:

> *"What tools/infrastructure do you need to build an autonomous animated character?"*

I put aside all the issues involved in actually building an animated environment for the character to live in, and concentrated on its behavior. Stripped to its essentials, I decided that a character (or **dude**[7]) was really a collection of ***goals***, ***sensors***, and ***skills***. I developed a distributed implementation that allowed the various goals, sensors, and skills to reside on different machines[8]. Since each of these goals, sensors and skills were well defined little black boxes, I referred to them as ***agents,*** using the "classic" definition of agent by Marvin Minsky:

> *"Any part or process of the mind that by itself is simple enough to understand — even though the interactions among groups of such agents may produce phenomena that are much harder to understand."*[9]

A dude, therefore, was a collection of goal agents, sensor agents, and skill agents, connected together in an action selection network.



I implemented a variation of Maes' "spreading activation" planner[10], wherein the various agents registered with the planner and periodically updated it regarding their status, with the planner dispatching "execute" messages to skill agents as the algorithm (and the current state of the various agents) dictated. A reasonable understanding of the planning algorithm will greatly aid the reader in understanding WavesWorld. **Appendix A** goes into some detail for the reader unfamiliar with this work.

3.1: A Distributed Implementation

One idea that I tried to implement at a low-level in Build-a-Dude was the idea that parts of the system would be distributed over a potentially wide area network of heterogenous computing resources. This was done for pragmatic performance considerations, and because I was interested in building tools situated in a wide area networked environment. I did not want to simulate the networking, or graft it on afterwards. I also wanted to make it possible to take advantage of parallelism in a single character.

3.2: Parallelism is Vital

In implementing this system so that the various agents could be running transparently on any machine on the network, I discovered that sometimes it made sense to have one agent (a complicated sensor agent, for example) have its own separate process. In most cases, it made sense to group many agents into one process. This led me to think very clearly about where I needed real parallelism (a set of asynchronous processes running on separate computational resources) and where I could simulate it (time-slicing in a single process). As in the previous section, it could be argued that issues like this are "implementation issues," and not interesting from a research perspective. Since parallelism is becoming an increasingly important part of software applications and most software in the future will be implemented in a distributed fashion, I felt that confronting these difficult issues was actually central to much of my work, and could not be shrugged off as simply "implementation details."

3.2.1: Active Objects and Agents

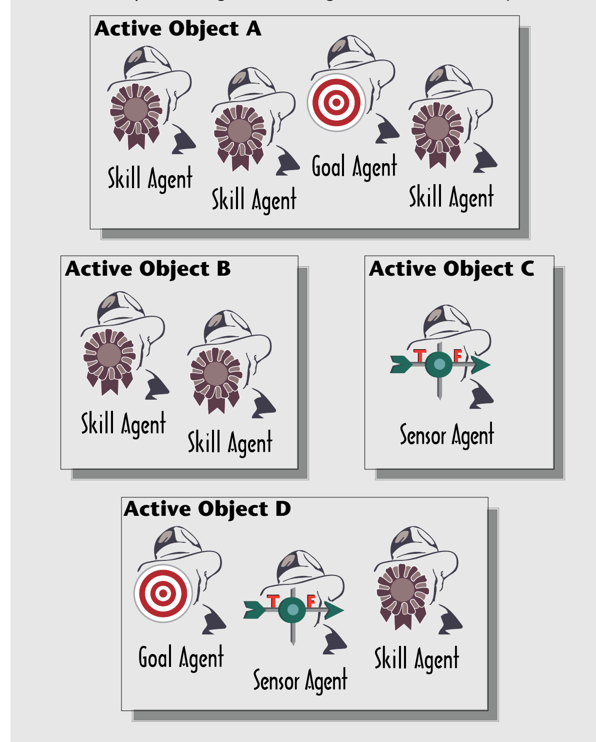
It became clear that I needed an additional abstraction if I wanted to integrate parallelism and agents in a given dude. In addition to the notion of a goal, sensor, and skill agent, I needed some notion of "process" or "autonomous computing resource" which was orthogonal to the notion of agent. In other words, I wanted the freedom to group a set of agents in one process, or have one agent be running across several processes. My ongoing solution to this problem is the use of **active objects**, which I will discuss in some detail later (see

6. Johnson, M.B. *Build-a-Dude: Action Selection Networks for Computational Autonomous Agents*, SMVS Thesis, Massachusetts Institute of Technology, Feb. 1991.
7. I chose the seemingly facetious term *dude* to refer to my particular implementation and approach. The terms *actor, agent, character, object, synthespian*, etc. have become so overloaded that it's hard to use them and make clear the context you're using them in.
8. Johnson, M.B. *Build-a-Dude*.
9. Minsky, M. "The Society of Mind", Simon and Schuster, N.Y, N.Y. 1987.
10. Maes, P. *How to Do the Right Thing*.

DRAFT

Appendix B).

An Example Arrangement of Agents and Active Objects



3.3: Environments for Understanding

Finally, the most important lesson I learned from Build-a-Dude is that a powerful development environment supporting debugging is essential to building any system entailing the sort of complexity that I was imagining WavesWorld to contain. This is a central issue, and one I believe many software systems fall short on. I found that an environment that supported debugging at all levels was vital[11], and that no real progress could be made until that was dealt with.

Debugging in this context does not mean finding and fixing a problem with the software system — it has more to do with understanding what has been built. Even after all the bugs are out of the computational system, you will still be finding and removing the bugs in *your mental model* of the system. It is this sort of debugging, this ongoing visualization and meaning-making process, that I found most interesting.

3.4: What Could it Do?

When I began Build-a-Dude, I hoped to be able to build a system which could be used to build fully autonomous animated characters that could exist for long periods of time in networked virtual environments, engaged in interesting behavior. As I began to actually use the system, I realized that the sort of behavior it was capable of was less than I'd hoped for. As Zeltzer and I pointed out[12], you might ask a dude to get you a beer from the kitchen, but you wouldn't expect it to sit down and play a game of chess with you. The reactive planner I used is powerful and flexible, but it is *still* a reactive planner. It does not really do symbolic reasoning (although the agents might), it can learn, but in rather limited ways, etc. In other words, I don't feel that the reactive planning system here can be used for long-form character animation unassisted.

If we consider Build-a-Dude as one component of a task-level graphical simulation system or synthetic cinema system, though, the contribution of the work becomes clearer. Build-a-Dude provides a foundation upon which the character animation component for either system could be built. The work on distributed and parallel infrastructure is vital as it helps outline the requirements such a character animation component places on the rest of the system. WavesWorld, as I'll now describe, is my current attempt to extend the work of Build-a-Dude to build such a component.

11. Zeltzer, D. *Task-level Graphical Simulation: Abstraction, Representation, and Control* in "Making Them Move", ed. Badler, N.I., Barsky, B.A., and Zeltzer, D., Morgan Kaufmann, Palo Alto, CA, 1991.

12. Zeltzer, D. and M.B. Johnson, *Motor Planning: Specifying and Controlling the Behavior of Autonomous Animated Agents*. Journal of Visualization and Computer Animation, April-June 1991, 2(2), pp. 74-80.

DRAFT

IV: Beyond Build-a-Dude

Three years ago, at the end of Build-a-Dude, I had a flexible, reactive planner that ran distributed over a network of heterogeneous computational resources, but I still had not tackled several hard problems that needed to be dealt with before I could use the software as a testbed for building autonomous animated characters[13]. The two most pressing issues I didn't deal with in Build-a-Dude were:

- *How does a character really perceive its environment?*
- *What does the character's body look like?*

4.1: Perceiving Itself and Its Environment

In Build-a-Dude, a virtual actor was completely described by its agents: skills, goals, and sensors. From an animation perspective, skill agents seem to be the most compelling part; that was where the actual proactive work of the character happens. The planning algorithm took care of the reactive part, i.e., when to call which skill agent.

Unfortunately, when you start to actually build a dude, you realize that the reactive planning part is driven largely by the sensor agents. The goal agents *can* play an important role, but realistically, a given virtual actor doesn't have more than a handful of explicit goals at any given point in time, while it usually has at least an order of magnitude more sensor agents.

To deal with this problem, a character builder needs to confront the issue of what exactly a sensor agent is — how is it connected to the world, what does it know, how often does it get its information updated, who is responsible for getting it the info it needs, etc. As discussed in my SMVS thesis[14], Maes' algorithm requires that sensor agents (or state, in her system) generate strict booleans: the cup is in the hand or it is not, the actor is standing or he is not, the soup is too hot or it is not, etc. Unfortunately, the world is usually fuzzier than that, and when implementing perceptual mechanisms of a character, it seems advantageous to capture that fuzziness. For example, although the soup may be too hot (i.e., the sensor agent **soupIsTooHot** is True), it may be just a few degrees too hot, not several hundred.

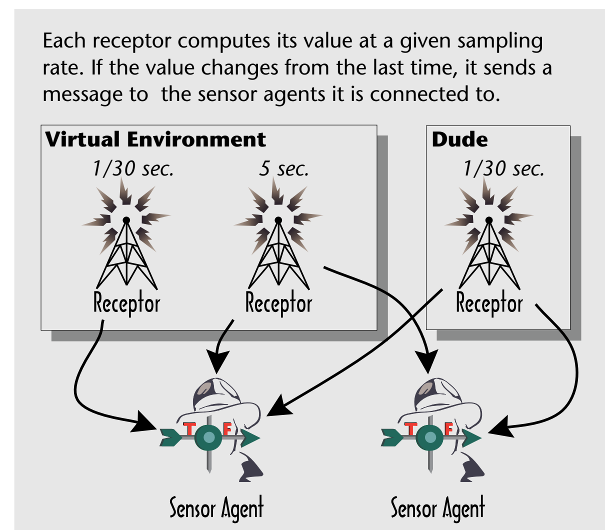
Another difficulty arises when you actually try and implement this in a networked environment. A given sensor agent probably uses information that other sensor agents also use. For efficiency, you would like to have that information gathered once and then sent out to each sensor agent that needs it, rather than having each sensor agent gather it independently.

4.1.1: Perception = Receptors + Sensor Agents

The solution I developed was inspired by a notion from Rasmussen who talked about the signals, signs, and symbols to which a virtual actor attends:

> *"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."*[15]

I realized that sensor agents corresponded directly to **signs**, but I needed some sort of representation for **signals**. In WavesWorld, these would be something that digitally sampled continuous signals in either the virtual actor or the virtual environment. I termed these perceptual samplers **receptors**. Receptors are code fragments that are "injected" by sensor agents into either the virtual actor or the virtual environment. Each receptor has a sampling frequency associated with it that can be modified by the sensor agent which injected it. These probes sample at some rate (say, every 1/30 of a second or every 1/2 hour) and if their value changes from one sample to the next they send a message to the sensor agent, which causes the sensor agent to recalculate itself.



Receptors can be shared among sensor agents when they are exactly the same (i.e., sampling frequency and code fragment). When a sensor agent injects a receptor, it can specify whether the receptor can be shared with

13. **A note on terms**: In WavesWorld, I tend to use the terms *character* or *virtual actor* as opposed to *dude*. Since WavesWorld is still evolving, it seems more appropriate to use such fuzzier terms, as what capabilities I'm proposing for a virtual actor/character are also evolving.
14. Johnson, M.B., *Build-a-Dude*, (pp 37 &72).
15. 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).

DRAFT

other sensor agents or not. If the sensor agent is willing to share its receptor, it can also specify a group name which corresponds to how it will be shared. This allows sensor agents in one virtual actor to share receptors with sensor agents in another virtual actor, as well as the more ubiquitous case of sensor agents in the same character sharing receptors.

4.1.2: Perfect Knowledge can be Boring: Aliasing Builds Character

One interesting issue in a networked autonomous animation system is how to deal with the question of the validity of the sensor data the character has. In a real distributed, parallel system, perfect sampling can be difficult to achieve without unacceptable performance compromises. If one component takes an appreciable time to respond to an information request, the entire system performance be dragged down, thereby obviating many of the advantages of building a parallel system in a distributed fashion. On the other hand, if the faster components proceed without getting the requested information from the slower component, they may be operating under false assumptions (i.e. that the requested data from that component has changed in some known way since the last request was answered).

As I approached this problem, I tried to think of a way to turn this difficulty to my advantage. I started with the assumption that since WavesWorld is not constrained to operate in real-time (as opposed to a real task-level animation system, which might be), it should be possible to make sure that all perceptual information was completely up to date before the next frame was rendered. This would potentially entail a combinatorial explosion in the number of messages in the system as the number of characters grew, but it *would* be possible, especially for a few simple characters.

What would it mean for a virtual actor to have out-of-date information? Imagine the following scenarios:

- *Martin is out dancing on the dance floor, oblivious to his neighbors. Suddenly the song finishes. He continues dancing for a few seconds, then suddenly realizes the song is over, and sheepishly makes his way off the dance floor.*
- *Mary pulls out a chair and turns around to sit down in it. Jane pulls the chair out of the way. Mary continues to sit down and falls as she goes to put her weight on a chair that's no longer there.*

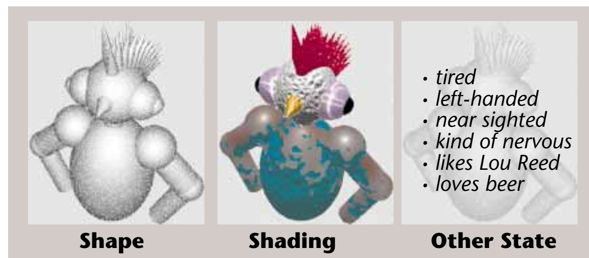
Clearly we can think of situations where it would be useful and interesting for a character to have out-of-date, incorrect perceptual information.

Allowing the character to realize that they have out of date information is also interesting. This can be accomplished by have higher level facilities that are monitoring the success of executing skill agents. Considering the previous examples, it would be useful to have Martin feel embarrassed and starts paying more attention to the music, and have Mary gets mad at Jane and become paranoid about sitting down, always checking and rechecking when she sits down in the future. In other words, the characters should be able to control the apportionment of their attention resources. If they want to be totally oblivious, fine. If they want to be totally paranoid, well, that's an option too. If they'd like to be very attentive to certain parts of their environment and a bit less attentive about others, well, that's an option too. The facility that receptors enable in building *character* into a virtual actor is a powerful one.

DRAFT

4.2: Body Parts

Up until now I've only discussed the behavioral components of a virtual actor. But that only covers the "autonomous character" part of what a virtual actor is, and has not addressed the "animated" component. In this section I'll discuss the shape, shading, and other state which comprise the visible and invisible animatable components of a virtual actor that the agents operate on.



4.2.1: Shape

In order for a virtual actor to be visible on-screen, it has to have some renderable geometric shape information. In WavesWorld, I refer to this as the **shape** of the virtual actor, and it corresponds to the hierarchical, geometric description of the character's body parts. Currently WavesWorld supports polygons, quadrics, patches, and trimmed NURBS as valid geometry. WavesWorld has built-in conversion tools to allow geometric models from most commercial modeling packages to be automatically imported. This allows users to use both models they construct, as well as commercially available clip objects from many different vendors.

4.2.2: Shading

Shading has to do with how the shape is actually rendered on a display. I use a way of describing shading taken from the RenderMan™ interface[16], which provides a special purpose programming language for describing shading. Commercially available shaders currently in use in WavesWorld allow users to simulate a wide variety of surfaces; from chipped, painted metal to linoleum floors, wood planking to painted stucco walls.

4.2.3: Other State

Shape and shading are really just two very specific instances of animatable shared state information in the virtual actor. There is other state in the virtual actor which doesn't neatly correspond to either of these categories, and it is lumped together under the rather vague heading of **other state**. This is purposely an open-ended area, as the kind of shared state (mental state, emotional state, short-term memories, long-term memories, etc.) maintained is highly object- and scenario-dependent. For example, a radio in the VE might have a list of channels it can tune in or a set of sound data that corresponds to the list of songs it plays. A character that knows how to dance might have a list of songs it likes and dislikes as part of its state, or even small samples corresponding to portions of the songs it likes to hum to itself. A little robot might remember what cup it sensed was nearby, as well as the fact that it is left-handed.

4.2.3.1: No Variables? No Problem!

One of the interesting uses of this other shared state is as short term memory that sensor agents can use to communicate to skill agents. Recall that the planning algorithm used in WavesWorld is based on Maes', which does not allow variables. It depends on deictic references[17] to achieve the same effect. In other words, if the skill agent **pickUpTheCup** is called, it is assumed that the cup in question is one that is nearby. When building physical robots that can use the world as a convenient place to hold state, that is a reasonable method. Unfortunately, in the purely synthetic worlds that WavesWorld is concerned with, I need some place to store such information. The "other state" of the virtual actor is used for this purpose.

For example, suppose you have a character with a sensor agent **aCupIsNearby** and a skill agent **pickUpCup**. The skill agent keys off (among other things) that sensor agent. When the sensor agent computes itself, it has access to the virtual environment via its receptors. If it computes itself to be True, it has, *at that time*, access to information concerning the particular cup that it has decided "is nearby." It stores this information by sending a message to the virtual actor and then returns True. That message is stored in the character's shared state as short term memory. Some time later, after the appropriate activation has flowed around the action selection network, the skill agent **pickUpCup** is called. The first thing the skill agent does is retrieve the necessary information about the particular cup it is supposed to be picking up from shared state of the virtual actor. This information might be stored as some static piece of data regarding some aspect of the object (i.e., where it was when it was sensed by the sensor that put it there) or it might be a piece of active data, like a pointer or handle to the actual object in the environment. Either way, this allows agents to communicate via the shared state of the virtual actor, using it as a blackboard.[18]

4.2.4: Building Bodies

In thinking about how to build the shape, shading and

16. Upstill, S. "The RenderMan™ Companion", Addison-Wesley, Reading, MA 1989.

17. A deictic word or phrase is "defined as expressions having the property that the manner of determining their references in various circumstances depends crucially on the context in which they are used." Miller G., *Some Problems in the Theory of Demonstrative Reference*, "Speech, Place, and Action", ed. R.J. Jarvella and W. Klein, John Wiley & Sons, 1982.

DRAFT

state that will constitute the virtual actor's body, there are several points I needed to keep in mind.

First of all, the description should be amenable to transport over a network, i.e., it should be a compact representation. It should be well integrated into the rest of the development environment, since the task of modeling is a highly iterative one, much like software development. Finally, it should allow users to import shapes and shading data from outside sources, such as commercial modelers like (say) Alias PowerAnimator™ or MacroMedia's MacroModel™.

4.2.2 Eve: A Modeling Language

With these requirements in mind, I designed a little modeling language called **eve**. Eve is essentially tcl[19] with a full RenderMan™[20] binding. Unlike tcl, though, eve has types (albeit optional), and also has a simple constraint propagation system, with a run-time system which maintains the constraints. Eve code is compiled at run-time by the eve compiler, **ec**. When ec compiles an eve file, it builds a hierarchical shape, shading and state description which can be efficiently rendered and animated. Sometimes, though, it would be useful if portions of the model were left interpreted, so that if the expressions they depend on change the model would reflect this.

This is very similar to a notion called *articulated variables*[21] (or **avars**) used in Pixar's modeling language **ML**[22]. ML is a C-like[23] procedural language in which animation is effected by allowing certain variables to change over time.

In eve, though, there is no need to predeclare variables, so instead of having variables which are defined as articulated, eve allows *articulated commands* (or **ACmds**). ACmds are any valid eve command that is composed of elements that may change over time. ACmds allow a model builder to leave animation hooks inside a model. For example, consider the following example (inspired by one in[24]):

```
defineShape SquishySphere
    Scale[expr 1/sqrt($s)] [expr 1/sqrt($s)] $s
    Sphere 1 -1 1 360
endShape
```

The shape defined here is scaled in X, Y, and Z by some function based on the value of the variable **s** when the model is evaluated. When the model is compiled, the values of the parameters passed to the Scale command is evaluated and assigned. Consider, however, this model:

```
defineShape SquishySphere
    ACmd {Scale[expr 1/sqrt($s)] [expr 1/sqrt($s)] $s}
    Sphere 1 -1 1 360
endShape
```

In this shape, the addition of the ACmd statement means that the expression:

```
Scale[expr 1/sqrt($s)] [expr 1/sqrt($s)] $s
```

is evaluated, just as in the previous example, but a dependency is set up in the run-time system whereby any changes to the variable component of that command (in this case, the variable **s**) will result in the immediate re-evaluation of this command, which will be reflected in the model the next time it is rendered. In this particular example, by manipulating s over time, the sphere undergoes a volume preserving scale which makes it look like it is being compressed.

4.2.1: Animating Bodies

Recall that WavesWorld is a testbed for autonomous animation. So far I have talked little, if any, about actually animating the characters built in WavesWorld. WavesWorld is designed to support a wide variety of animation techniques: key-framing, forward kinematics, inverse kinematics, forward dynamics, inverse dynamics, etc.

Time

Eve allows a programmer to write routines which manipulate a virtual actor in its environment in a way which has reference to the real time of the virtual environment. In WavesWorld, seconds are divided into ticks.

If a particular motor program is being written with the knowledge that it will be used in an environment where it is certain to be able to

```
set u 0
set howManySteps [expr 3 * ticksPerSecond]
set uIncr [expr 1.0/[expr {$howManySteps - 1}]]
for {set i 0} {$i < $howManySteps} {incr i} \
    {  performTheAction $u; \
       set ticksPassed [synchWithScene]; \
       set u [expr $u + ($uIncr * $ticksPassed)]; \
    }
```

18. Hayes-Roth, B. *A Blackboard Architecture for Control,* in "Readings in Distributed Artificial Intelligence", Ed. by Bond, A. and Gasser, L., Morgan Kaufmann, 1988.
19. Ousterhout, J. "Tcl and the Tk Toolkit", Addison-Wesley, Reading, MA, 1994 in preparation).
20. Upstill, S. "The RenderMan™ Companion".
21. Reeves, William T., Eben F. Ostby, Samuel J. Leffler, *"The Menv Modelling and Animation Environment"*, The Journal of Visualization and Computer Animation, Vol 1:33-40 (1990).
22. ibid.
23. Kernighan, B.W. and D. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.
24. Reeves, W.T. et.al., *"The Menv Modelling and Animation Environment"*,, page 35.

DRAFT

Sampling: Scene vs. Shot (Simulation vs. Visualization)

4.3 So What's a Virtual Actor in WavesWorld?

Recall that in Build-a-Dude I gave the following definition of a dude: *a collection of goal agents, sensor agents, and skill agents, connected together in an action selection network*. In WavesWorld, I've expanded this to include both my new perceptual mechanisms (receptors), internal state (emotional state, short term memory, etc.) and the actual animatable body parts of the character (it's shape and shading info).

So in WavesWorld, what is this "character" or "virtual actor" I've been talking about? Here's my current working definition: *some set of state, including internal state, geometric shapes, and shading information, along with a collection of receptors, goal agents, sensor agents, and skill agents, connected together in an action selection network.*

V: An Example: sanderWorld

Now that you have some notion of what can be modeled in WavesWorld, let's look at a simple example to explain how the whole system works. The world that I'll discuss is called sanderWorld and is based on the example given in Maes[25], which in turn was based on one in Charniak & McDermott[26]. The scenario consists of a single character, a two-handed robot which has the dual tasks of spray-painting itself and sanding a board.There are a few props in the world: a sander, a sprayer, a board, and a vise. As Maes points out, this is a toy example, but one with enough complexity to give a glimpse of what the system can do.

5.1: The Robot Sander/Sprayer — A Minimally Defined Character

Our character, the sander, consists of some geometry, shading parameters, internal state, agents and receptors.

As far as geometry goes, it will need at least two arms, and some way of moving around. At this stage, you don't really care about his shading parameters. You might imagine that the robot has two arms and can magically fly around, i.e.:

With regard to internal state, the requirements for that will emerge as you start to design the character's behavior. To implement the behavior, it needs the following agents:

two goal agents:

- **boardSanded**
- **selfPainted**

ten skill agents:

- **pickUpSprayer**
- **pickUpSander**
- **pickUpBoard**
- **putDownSprayer**
- **putDownSander**
- **putDownBoard**
- **sandBoardInHand**
- **sandBoardInVise**
- **sprayPaintSelf**
- **placeBoardInVise**

ten sensor agents:

- **operational**
- **boardSomewhere**
- **sanderSomewhere**

25. Maes, P. *How to Do the Right Thing*
26. Charniak, E. & McDermott, D. *Introduction to Artificial Intelligence,* Reading, MA, Addison-Wesley, 1985.

DRAFT

- **sprayerSomewhere**
- **boardInHand**
- **sanderInHand**
- **sprayerInHand**
- **handIsEmpty**
- **boardSanded**
- **selfPainted**

The above sensor agents will need the following eleven receptors, which will be embedded either in the virtual actor (VA) itself or the virtual environment (VE):

- **robotMobility (VA)**
- **robotLocation (VA)**
- **leftHandLocation (VA)**
- **rightHandLocation (VA)**
- **boardLocation (VE)**
- **sanderLocation (VE)**
- **sprayerLocation (VE)**
- **iAmHoldingSomethingLeft (VA)**
- **iAmHoldingSomethingRight (VA)**
- **iHaveBeenPainted (VA)**
- **boardHasBeenSanded(VE)**

5.2: A Digression on Designing Sensor Agents & Their Receptors

It is important to realize that the preceding list of receptors are somewhat arbitrary. They depend on both the information that we can get from the place they are embedded (i.e., the virtual actor or the virtual environment) and what kind of computation the sensor agent wants to perform (i.e., very general to very specific). Since you are the one implementing the sensor agents, it comes down to how you want to construct them.

For example, we have a sensor agent **boardSomewhere**. How does it work? Well, I might have a virtual environment that can answer that question ("is the board somewhere?") directly. If so, it would only need a single receptor to ask the virtual environment that question every once in a while. Unfortunately, the board might be located miles away from the robot, which really doesn't follow the spirit of what the sensor agent is supposed to measure. Let's assume the virtual environment can answer the question "where is the board?" I'll define a receptor **boardLocation** and inject it into the virtual environment.

For the sensor agent, you would like to take that info and compare it with the location of the robot and see if they are within a few meters of each other. If so, the sensor agent will return True. To do this, the sensor will need to know the robot's location, which involves defining another receptor, **robotLocation**, and "injecting" it (see **Appendix B**) in the character. You should realize that the selection of the criteria "within a few meters" is again arbitrary; if it works, great, if it doesn't, change it.

5.2.1: Receptors & Agents: A Programmer's Perspective

At this point, you might be asking yourself "what does a receptor and an agent actually look like?"

Well, from a programmer's perspective, a receptor is a code fragment which is executed at some sampling frequency. It returns a single, arbitrarily typed value. If its value changes from one sample to the next, it sends a message to each sensor agent which has registered interest with the receptor, apprising the sensor agent of the receptor's new value. A receptor can only communicate directly with the environment that it is embedded in and indirectly with the sensor agents dependent on it. Here's an example of a receptor:

```
proc aReceptor {variableName oldValue sensorList} {
    global $variableName
    set v [set $variableName]
    if {![string compare $v $oldValue]}
       {foreach s $sensorList {ask $s "updateRV $v"}}
       {}
    return $v
}
```

One of the driving assumptions behind the receptors/sensor agents split is that a receptor will be evaluated many times relative to the sensor agent, so a given receptor should be computationally simpler than the sensor agents using them.

A sensor agent is a function which takes as its argument a set of receptor values, corresponding to the current sample values of the corresponding receptors. It only can communicate to the virtual actor (to store state information), and returns a single boolean value. It can be as simple as a single line of code to a very complex function.

```
proc aSensorAgent {rv1 rv2} {
    # do some calculations involving the passed in values
    # of the receptors this sensor value depends on
    if {$calculationResult} {return T} {return F}
}
```

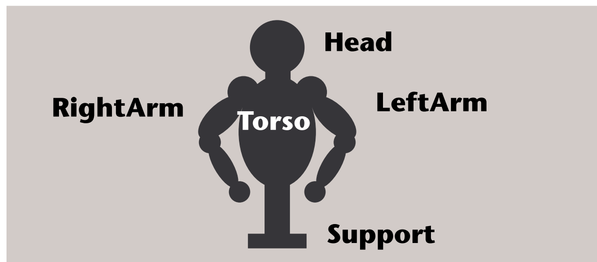
A skill agent is defined by a set of lists of sensor agents' names which it depends on, as well as an function to be called when it is executed. Inside the body of the skill agent's function it can communicate directly to the virtual actor's shared state and the virtual environment.

5.3: The Robot Sander/Sprayer — Iteration 2

You now have a set of behavioral components which comprise a minimal but complete set of parts to build our virtual actor. You might now start specifying some more interesting body parts (both shape and shading)

DRAFT

for our actor. You can start off with a pretty simple robot



with a head, two arms, a torso, and a single leg for support. As we continue, you can refine it more, but this is better than the two arms and magic flying powers it had before.

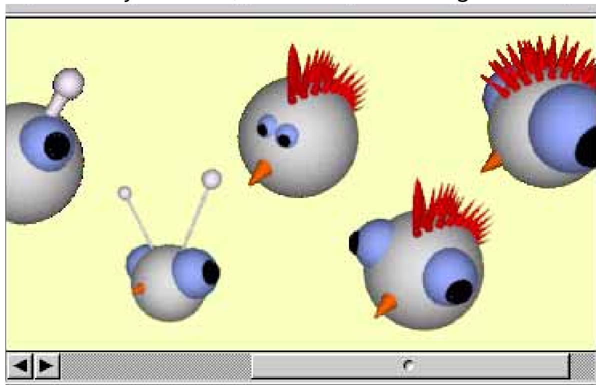
You now need to implement the various skill agents. To start with, you can make them magically successful. For example, the **pickUpSander** skill agent will consist of one message, telling the sander to move itself to the robot's nearest hand:

```
proc pickUpSanderExecuteMethod {} {
    set location [askBodyManager “set freeHandLocation”]
    ask EnvironmentManager “set sanderLocation $location”
    return
}
```

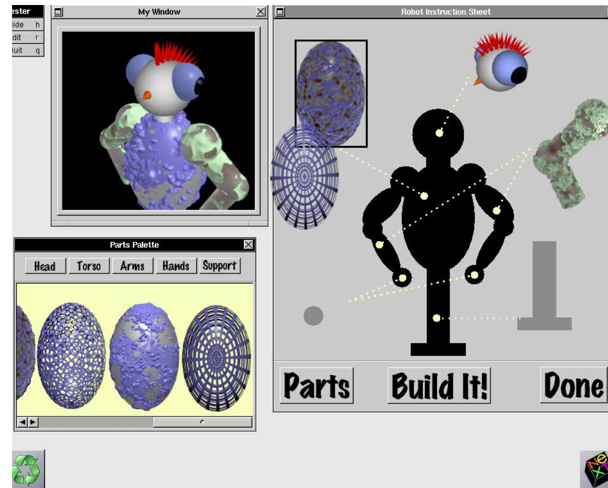
The rest of the skill agents are then implemented similarly.

5.4: Making Choices Map to Other Choices

Now that you have a basic, functioning implementation of our robot and its environment, you can now iterate over specific portions and improve them. The first thing we might want to do is allow for more variety of shape. You might decide to design several different kinds of heads. Using the tools in WavesWorld, you can quickly and easily build a user interface to allow manipulation of some set of parameters in a head model written in eve. After a bit of experimentation, you might settle on a small variety of heads, such as the following:



Continuing on that tack, you might decide to allow a variety of shape and shading for the various body parts of the virtual actor. Certain combinations of shapes and shading parameters might imply particular values of the shared state of the virtual actor, and might actually change which of several implementations of the skill or sensor agents are used for that particular character. For example, here's a user interface I designed from scratch and implemented in a few hours one evening:



The window on the lower left was a parts palette, containing samples of various configurations of body parts. By dragging the images of the various body parts on to the diagram on the right, the user made certain choices concerning the character under construction (visible in the upper left window). What's especially interesting here is the fact that although each particular choice had obvious mappings to shape and shading (“choose this head; get this kind of shape, choose this torso, get it shaded that way”), it can also be used to control other aspects of the character. For example, if you choose one of the heads with the eyes in “prey position” (i.e., on the side of the head, as opposed to the front), the virtual actor might act in a more skittish or nervous way as it goes about its activities. On the other hand, the fact that you choose one of the peeling, rusted torsos might cause the virtual actor that is built to be somewhat brutish and clumsy, intent on its task with little regard for what it bumped into along the way.

5.5: Iterative Refinement of Agents

One of the underlying themes in building a testbed is allowing many solutions to the same problem. Depending on your particular criteria of the moment, one solution might be better than another, but viewed from a certain perspective, they both solve the original problem. The approach to perception I take in WavesWorld ties in with this theme nicely. A given implementation of a sensor agent might initially report on some aspect of the environment in a rather naive way, but one that

DRAFT

works for the given situations the character finds itself in. In trying make the sensor agent more general, a developer might build some more receptors for the sensor agent which allow it to be more sensitive to the original conditions it was measuring or to allow it to take other situations into account.

I want to equally support two different approaches: very specific and very general. If a developer just wants to bang out a minimal agent implementation that consists of a single line of code, great. If they want to labor over an agent that needs a dedicated CM-5 to run, that should be fine too. One of the difficulties with building an autonomous animation testbed is the difficulty in allowing scalable, iteratively developed behavior. For example, say you'd like to build a simple character that can pick up and put down a variety of items in a particular setting. What you want to do is sketch out a set of skill and sensor agents which form a framework to do this. You'd like to be able to implement each with simply a line or two of code: the **pickUpCup** skill agent merely translates the cup to the hand in one frame, the **cupIsNearby** sensor agent always returns True, etc. After the character selects and performs the appropriate activity, you can then begin to refine the agents.

Recall that WavesWorld is a testbed; by its nature it should not only *allow* experimentation, it should p*romote* it.

VI: Conclusions

I've shown what problems WavesWorld is trying to tackle, and how it has evolved into the system that it is today. WavesWorld successfully brings together disparate ideas from diverse fields such as AI, computer graphics & animation, parallel and distributed programming. It raises questions about how to build autonomous animated characters that exist in networked virtual environments and provides tools for exploring those questions.

WavesWorld is a still evolving system. I'm just now at the point where I've brought all the important components up to parity with each other and can begin experimentation in earnest. I hope to enlarge my base of collaborators this year to include the diverse set of people that I envision using task-level animation and synthetic cinema systems in the future.

DRAFT

Appendix A: The Spreading Activation Planning Algorithm

This appendix details the planning algorithm used in WavesWorld. This appendix is based on Chapter 3 of my SMVS thesis[27], although it has been updated to reflect my current research results and implementation. The original planning algorithm was developed and described first in Maes' paper[28], and this particular implementation and extensions (for parallel skill execution and distributed execution) was first described in[29] and in more detail in[30]. The extensions with regard to sampling behaviors and perceptual sampling controls have not been discussed elsewhere.

The notion of using a network of inter-connected motor skills to control the behavior of a virtual actor was first described by Henry Jappinen[31] in 1979. My advisor David Zeltzer discussed this notion in the context of computer animation[32] in1983. This was later independently elaborated and first implemented by Pattie Maes[33] in1989 for physical robots. 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 in use on a wide variety of platforms.

This appendix begins with an algorithm for the problem of action selection for an autonomous agent as presented by Maes[34], 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 two errors I found while implementing it, as first noted in my SMVS thesis.[35]

A: I: 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 *t*; *G* being implemented by an independent process
- a function **R(t)** returning the propositions that are a goal of the agent that have 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 was 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 $\mathbf{x = (c_x, a_x, d_x, \alpha_x)}$, the input of activation to module *x* from the state at time *t* is:

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

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:

$$inputFromGoals(x, t) = {}_j\, \gamma \frac{1}{\#A(j)} \frac{1}{\#a_x}$$

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

The removal of activation from competence module *x* by

27. Johnson, M.B. *Build-a-Dude*.
28. Maes, P. *How to Do the Right Thing*, Connection Science, Vol. 1, No. 3, 1989.
29. Zeltzer, D. and M.B. Johnson, *Motor Planning: An Architecture for Specifying and Controlling the Behavior of Virtual Actors,* The Journal of Visualization and Computer Animation, Vol 2:2 (1991).
30. Johnson, M.B. *Build-a-Dude*.
31. Jappinen, H.J, *A Perception-Based Developmental Skill Acquisition System,* Ph.D. thesis, Ohio State University, 1979.
32. Zeltzer, D.L. *Knowledge-Based Animation,* Proc ACM SIGGRAPH/SIGART Workshop on Motion, 1983.
33. Maes, P. *How to Do the Right Thing*.
34. ibid
35. Johnson, M.B. *Build-a-Dude*.

DRAFT

the goals that are protected at time *t* is:

takenAwayByProtectedGoals(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 $y = (c_y, a_y, d_y, \alpha_y)$:

spreadsBW(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:

spreadsFW(x, y, t) =

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

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

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

takesAway(x, y, t) =

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

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:

$$\alpha(y, 0) = 0$$

$$\alpha(y,t) = decay \begin{pmatrix} \alpha(y, t-1)(1 - active(y, t-1)) \\ + inputFromState(y, t) \\ + inputFromGoals(y, t) \\ - takenAwayByProtectedGoals(y, t) \\ + \sum_{x,z} \begin{pmatrix} spreadsBW(x, y, t) \\ + spreadsFW(x, y, t) \\ + takesAway(z, y, t) \end{pmatrix} \end{pmatrix}$$

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 if

$$\alpha(i, t) >= \theta \quad (1)$$
$$executable(i, t) = 1 \quad (2)$$
$$\forall_j fulfilling(1) \Lambda (2): \alpha(i, t) >= \alpha(j, t) \quad (3)$$

active(t, i) = 0 otherwise

A: II: Maes' Algorithm: Pros and Cons

2.1: 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 animal 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[36,37]. 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.

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."* [38]

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.

36. Sherrington, C.S. "The Integrative Action of the Nervous System," Yale University Press, 1906.
37. McFarland, D.J. *The Behavioral Common Path,* Phil. Trans. Roy. Soc., 270:265-293, London, 1975.
38. Maes, P. *How to Do the Right Thing.*

DRAFT

2.2: 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"[39] vs. efficiently implementing it as such*
- *the lack of a method of parallel skill execution*

39. ibid.

A: III: Some Proposed Solutions

3.1: 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 which I originally thought would be too limiting for anything more than toy networks built by hand. I felt that any implementor would soon tire 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.

My original solution was to posit a sort of generic competence module, which I called 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.

The template idea is a reasonable start, but it helps to think about this algorithm in the context of a larger system, where given a particular situation, a piece of software might analyze the scenario and generate a action selection network, with all the requisite receptors and agents from a set of high level, general template skill and sensor agents.

My current solution is to allow the sensor agents to communicate to skill agents by manipulating shared state in the virtual actor. For example, suppose you have a character with a sensor agent **aCupIsNearby** and a skill agent **pickUpCup**. The skill agent keys off (among other things) that sensor agent. When the sensor agent computes itself, it has access to the virtual environment via its receptors. If it computes itself to be True, it has, *at that time*, access to information concerning the particu-

DRAFT

lar cup that it has decided "is nearby." It stores this information by sending a message to the virtual actor and then returns True. That message is stored in the character's shared state as short term memory. Some time later, after the appropriate activation has flowed around the action selection network, the skill agent **pickUpCup** is called. The first thing the skill agent does is retrieve the necessary information about the particular cup it is supposed to be picking up from shared state of the virtual actor. This information might be stored as some static piece of data regarding some aspect of the object (i.e., where it was when it was sensed by the sensor that put it there) or it might be a piece of active data, like a pointer or handle to the actual object in the environment. Either way, this allows agents to communicate via the shared state of the virtual actor, using it as a blackboard.[40]

3.2: Loops

The second major difficulty with the original 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 ($\theta, \phi, \gamma, \delta$) to effect change in that network's behavior. This is an idea much in the spirit of Minsky's B-brains[41], 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. 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.

A more interesting solution, and one that has its basis in the ethological literature, is to introduce a notion of fatigue to competence module execution. By introducing some notion of either skill-specific or general fatigue levels, the repeated selection of a given competence module (or sequence of competence modules) can be inhibited.

40. Hayes-Roth, B. *A Blackboard Architecture for Control,* in "Readings in Distributed Artificial Intelligence", Ed. by Bond, A. and Gasser, L., Morgan Kaufmann, 1988.

41. Minsky, M. *The Society of Mind.*

3.3: How to Select $\theta, \phi, \gamma, \delta$

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, 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 competence modules called *follower* links, we could modulate the 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[42] and Schank's scripts and plans[43], 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.

3.4: 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."*[44]

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

42. Minsky, M., "The Society of Mind."

43. Schank, R. and Abelson, R. "Scripts, Plans, Goals and Understanding," Lawrence Erlbaum Associates, 1977.

44. Maes, P. *How to Do the Right Thing.*

DRAFT

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 traditional AI search, and that it does not suffer from combinatorial explosion.[45] 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 $\mathbf{O(n^2)}$, 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 $\mathbf{c_x}$, $\mathbf{a_x}$, $\mathbf{d_x}$, $\mathbf{c_y}$, $\mathbf{a_y}$, and $\mathbf{d_y}$ must also be available to calculate the activation flow. This implies either a global database/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.

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)**, $\mathbf{c_x}$, $\mathbf{a_x}$, $\mathbf{d_x}$, $\mathbf{c_y}$, $\mathbf{a_y}$, and $\mathbf{d_y}$) 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 $\mathbf{a_x}$, $\mathbf{d_x}$, $\mathbf{c_y}$, $\mathbf{a_y}$, and $\mathbf{d_y}$). 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.

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 manage 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 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 $\mathbf{a_x}$, $\mathbf{d_x}$, $\mathbf{c_y}$, $\mathbf{a_y}$, and $\mathbf{d_y}$. 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.

3.5: 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 **chewGum** 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.

45. ibid.

DRAFT

A: IV: Summary of My Algorithm Extensions

4.1: Asynchronous Action Selection

A crucial difference between my algorithm and Maes' original is that (in mine) once a skill has been selected, an execute message is dispatched to it, and the action selection process continues actively trying to select the next action. Since the process actually executing the action is distinct from the process in which action selection is taking place, there is no need to enforce synchronicity at this point (i.e. we don't have to wait for that function call to return). At some point in the future, a finished message will probably be received by the registry/dispatcher, signifying that the skill agent has finished attempting to achieve the task.

4.2: Parallel Skill Execution

An executing skill agent will tie up some of the network's resources in the world, thereby inhibiting other similar skill agents from executing, while still allowing skill agents which don't need access to the same resources the ability to be selected. For example, if the walk-to-door skill is selected, it will engage the legs resources, and other skill agents (run-to-door, sit-down, etc.) will be inhibited from executing because one of their preconditions (legs-are-available) is no longer true. Other skill agents, such as close-window, or wave-good-bye, can still be selected, because the resources they require are available, and they are (possibly) receiving enough activation energy to be selected.

4.3 Adding Fatigue

At the beginning of each step of the action selection algorithm's loop, the registry/dispatcher checks for messages from any currently executing skill agents. When a given skill agent has finished executing (either believing it accomplished its task or giving up for some reason), it sends a message back to the registry/dispatcher. At this point, and not before, the skill agent's activation level is reset. This is analogous to the last section of Maes' mathematical model in which the selected competence module's activation level is reset to zero. In our implementation, the skill agent's activation level is reset taking into account how successful it was in affecting the world in the way it predicted it would, combined with some hysteresis to avoid continually executing a skill agent which always fails to do what it promised.

If a skill agent has been called (i.e. sent an execute message) consecutively for its maximum number of times (this can be network dependent, skill agent dependent, time dependent, or some combination of all three), its activation level is reset to zero. If however, the skill agent has not been called consecutively more than its maximum, its new activation level is calculated thus:

$$\alpha_{current}\left(1-\left(\frac{\text{consecutiveCalls}}{\text{maximumCalls}}\right)\right)\left(1-\left(\frac{\text{correctPredictionsAboutWorldState}}{\text{totalPredictionsAboutWorldState}}\right)\right)$$

The intuitive idea here is that if a skill agent has seemingly done what it said it would, its goal has been achieved and its activation level should be reset to zero (just as in Maes' original algorithm). If however, the world state as measured differs significantly from what the skill agent predicted, the obvious course of action is to allow that skill agent a high probability of being selected again (by not reducing its activation level much). The likelihood of calling a given skill agent consecutively should decrease over time, thereby building some hysteresis into the action selection. For example, if you dial a phone number and get a wrong number, you usually attempt to dial the phone number again. If the second attempt fails, you start looking around for some other way to accomplish your goal-rechecking the number in your address book, calling directory assistance, etc.

4.4 Perceptual Sampling Controls

In Maes' original system, while provisions are made for the fact that a given skill may or may not be successful, there is no discussion of sensor fallibility. Two situations may lead to a sensor reporting incorrect information: incorrect assumptions by the sensor designer, or undersampling of the phenomena that the sensor is built to perceive. While there is little we can explicitly do about a badly designed sensor, we can discuss sampling.

The solution I developed was inspired by a notion from Rasmussen who talked about the signals, signs, and symbols to which a virtual actor attends:

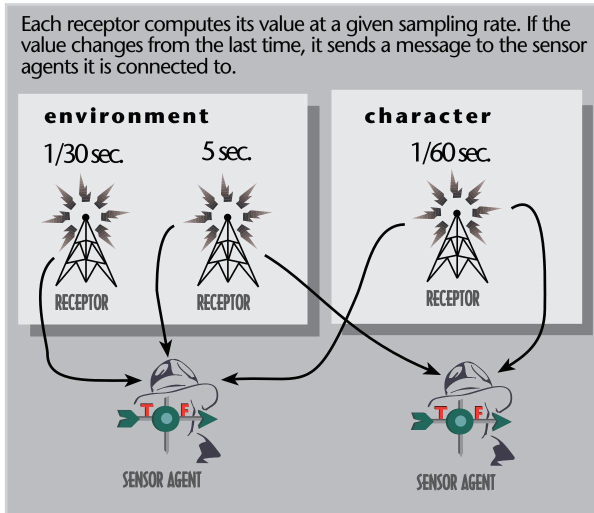
> *"Signals represent sensor data — e.g., heat, pressure, light — that can be processed as continuous variables. Signs are facts and features of the environment or the organism."*[46]

I realized that sensor agents corresponded directly to **signs**, but I needed some sort of representation for **signals**. In WavesWorld, these would be something that digitally sampled continuous signals in either the virtual actor or the virtual environment. I termed these perceptual samplers **receptors**. Receptors are code fragments that are "injected" by sensor agents into either the virtual actor or the virtual environment. Each receptor has a sampling frequency associated with it that can be modified by the sensor agent which injected it. These probes sample at some rate (say, every 1/30 of a second or every 1/2 hour) and if their value changes from one sample to the next they send a message to the sensor agent, which causes the sensor agent to recalculate

46. 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).

DRAFT

itself.



Each receptor computes its value at a given sampling rate. If the value changes from the last time, it sends a message to the sensor agents it is connected to.

Receptors are shared among sensor agents, and their information is also available to skill agents. Because their sampling frequency can be modulated, it is possible to trade off the cost of sampling the world at a high frequency vs. the possibility of aliased signals.

Appendix B: A Brief Discussion of Active Objects

I: Introduction

Recall that in Build-a-Dude I realized that agents didn't necessarily map well to a single autonomous process, and I decided that I needed some orthogonal notion of what a process was. Ideally, a given process could run anywhere, on any appropriate computational resource. Secondly, since we are taking an object-oriented/message-passing perspective, the messages that the processes send to each other should be rich enough to comprise a language. Ideally, the messages would be in some dynamic language which would allow the processes to send complete programs to each other and potentially embed code inside of other processes. Finally, these processes should be flexible enough to understand that they were one process among many, and be able to understand how to trade computational power for communications bandwidth. In other words, a process should be able to determine, at a given point in time, if it is cheaper to send another process a message, polling for some information, or if it is cheaper to download a piece of code (i.e., a set of messages) which will run locally in that remote process and apprise the sending process that the information is available. The process should be able to determine this (fallibly, of course) and be willing and able to act on this knowledge.

1.1: Criteria

So to reiterate, my criteria for a useful process are:

- *portable, i.e., run on many kinds of computers*
- *allow trading of computational power for communications bandwidth, and vice versa*
- *communicate with other processes in a reasonably expressive dynamic language*

This notion of a process I've come to call an active object, which, in contrast to most traditional notions of a software object, is actually an active process, with its own protected name space and inner loop. I have several different ways of explaining this; let's start with an intuitive one before we jump in to the hoary details.

DRAFT

II: An Intuitive Explanation

An active object starts up and plugs in its phone. Unfortunately, you don't have a phone book — you can take calls, and you could make calls, if you only knew who to call, but you don't. You might have a short list of phone numbers of people that you call, but in general, people call you, you don't call them.

After installing your phone (with integrated fax machine) and checking the line, you then drop into your daily routine, which is an endless loop. In this loop, the first thing you do is check your original phone and see if anybody is trying to call you. If they are, you pick up the phone, say hello, and put them on hold. You only have one phone number, but you have a couple of hundred lines, so you can do that. You then check any of the lines of people that you've already put on hold. If any of them want to talk to you, you pick up the phone for a short time and speak to them or read the fax they sent you. You give each line you've put on hold a short opportunity to chat, and then you put them back on hold. You then move on to do any work that you particularly do (this is where each active object can be slightly different). You then move on to a pile of Post-It notes that the folks on the phone have faxed to you. The Post-It notes have questions on them. The Post-It notes are actually in neat little stacks which correspond to how often the folks that sent them expect you to look at them. Some are supposed to be looked at every opportunity, and some are just supposed to be looked at every hour; it varies. Anyway, when looking over the question on a given Post-It note, if your answer to it differs from the last answer you gave to this same question, you fax back this new answer to the person who faxed you the question. The next thing you do is check your fax line for messages from anybody else that *you* have faxed questions to and see if any of the questions you sent them have new answers. Finally, you look at your itemized TODO list sitting on the legal pad next to you and, if it's the appropriate time, you perform the various tasks on the list.

Two important points about the Post-It notes: unless people explicitly tell you not to, if two people fax you the same Post-It note, you actually throw the second Post-It note away and scribble the second sender's name at the bottom of the first Post-It note. Then when you make the call to update the sender with the new answer to his question, you also call all the names scribbled at the bottom of the Post-It note. The other important side note is that anybody who's name is on a Post-It note can call you up and ask you to put the Post-It note on a different priority pile, so that you will look at it more or less often.

III: A More Precise Description

An active object is a single, sequential process running on some computational resource. It comes on-line somewhere on the network, and makes itself available for communication with others that speak its language. All active objects can understand some subset of messages that a given active object sends, but some respond to certain messages, or **protocols**, that other don't. After initializing itself and making itself available for communication, it drops into an endless loop, checking for new connections, evaluating messages from old connections, doing whatever activities that it specifically does, and talking to itself (more about this later).

```
do forever
    checkEmbeddedReceptors()
    checkForDirtyReceptors()
    if (newConnectionPending())
        hookUpNewConnection()
    if (msgPendingFromConnection())
        readEvaluatePrintOnConnection()
    doWhatThisParticularActiveObjectDoes()
    evaluateCommandList()
```

3.1: Attaching, Asking, Telling, Waiting

There are a few important activities relating to active objects that we haven't really discussed yet. How does it communicate with other active objects? All communication in WavesWorld is point-to-point. This breaks down in to 4 activities:

- *making itself available to receive messages*
- *attaching to another active object so it can send that active object messages*
- *sending messages to another active object*
- *getting immediate replies to messages it sent to another active object*

When an active object starts up, it makes itself available to the world to be able to receive messages. Once an active object knows where another active object lives, it can send itself an **attachTo** to attach to that active object. One of the arguments that the **attachTo** msg takes is a unique (to that active object) name by which it will refer to the active object it is attaching to.

If an active object wants to tell another active object something (i.e., send it a msg), it sends itself a **tell** msg with the name of the active object it wants the msg sent to and the contents of the msg. A **tell** returns as soon as it can, as it is assumed that the sending active object

DRAFT

doesn't care about hearing the reply. If, on the other hand, an active object wants to ask another active object a question, or if it just cares what the reply is to a given msg, it sends itself an **ask** msg with the name of the other active object and the contents of the msg. The **ask** msg returns the msg that is the reply from the other active object.

3.2: A Brief Note on Time

In WavesWorld, time is measure in two ways: **ticks** and **real-time**. Real-time is exactly that; the amount of time that really passes, as measured on the computer that the active object is running on. Real-time in active objects has a resolution limited to a maximum of 2^{31} seconds and a minimum of 1 millisecond. A "tick" is the time it takes a given active object to go once through its main loop. This means that the absolute length of a tick (measured in real-time) can vary over time in a given active object. It also means that ticks aren't equivalent over

As we'll see later, certain active objects maintain still other notions of time; such as **frame number**. Ticks and real-time are common to all active objects though, and are important to understand.

3.3: Periodic Behavior — the Command List

An active object is, by definition, always active; communicating or computing. Sometimes you want an active object to engage in some periodic behavior which is different than what it would do if left to its own devices. Each active object maintains a command list, which is a list of commands that it sends itself periodically. A command is a sequential list of msgs. Each command has a frequency and duration associated with it, which tell the active object how often to send the msg (i.e., every time through the main loop, every second, every hour, etc.) and how long the active object should keep the msg on the command list (ten, infinitely, until some expression is true, etc.).

Periodic behavior is a very useful concept, and it lets an active object time slice its own activity. Since both the period and life-span of a command can be specified in either ticks or real-time, the command list allows very flexible periodic behavior. An example of a tick based command might for debugging an active object, where each time through the main loop some variable was checked. A real-time based command could be used to prototype some behavior which might eventually get implemented as a subclass of some active object.

3.4: Sampling the World

In my experience, there tend to be two schools of thought concerning an autonomous creature sensing the world. The more traditional AI approach says that when the information is desired, it is available. More sophisticated algorithms make allowances for noisy or incorrect data. (talk about hardware folks vs. off-line planners in software, not concerned with the amount of info). Real-time computer graphics systems, on the other hand, tend towards a philosophy of you-gets-what-you-gets-in-between-a-frame. If the load gets higher than that, it's not interesting, because you can't do anything with it.

Something that both approaches don't seem to discuss is a notion of differentially sampling the world.

Recall the diagram of an active object's main loop:

```
do forever
    checkEmbeddedReceptors()
    checkForDirtyReceptors()
    if (newConnectionPending())
        hookUpNewConnection()
    if (msgPendingFromConnection())
        readEvaluatePrintOnConnection()
    doWhatThisParticularActiveObjectDoes()
    evaluateCommandList()
```

We've discussed the connections, but we haven't talked about what **checkEmbeddedReceptors()**, **checkForDirtyReceptors()**, and **evaluateCommandLst()** do.

3.5: The Command List

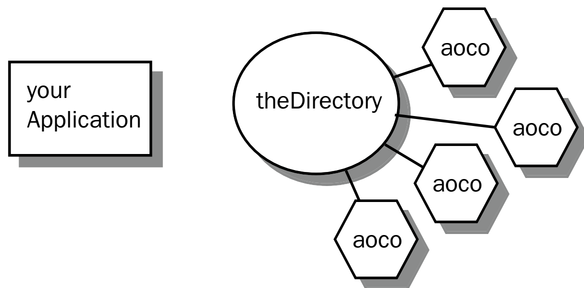
Recall that one of the ideas behind WavesWorld is to build systems that can dynamically trade off between communications bandwidth and computational power. In other words, sometimes it's more efficient to send a lot of messages to find something out (when communication is cheap compared to computation)

DRAFT

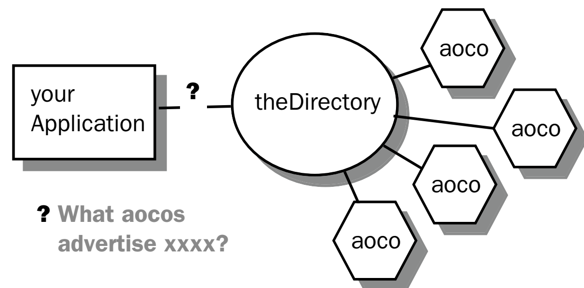
IV: Who Lives Where?

An obvious difficulty with active objects is the fact that no mention has been made of how active objects find out where other active objects are. If one writes a program that is composed of a bunch of active objects communicating with each other, how does the application do the dynamic binding of "object to iron", i.e., actually start up all of the various active objects and attach them to each other?

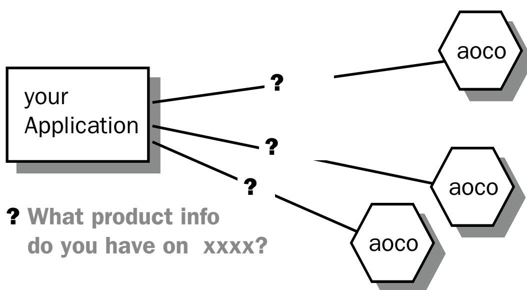
Somewhere on the network, at some location(s) known to the application code, there is an active object which is known as "theDirectory". Applications that want to use active objects come to the directory to find out product information about the active objects that they want to use. Various "active object companies", or **aocos**, advertise various products (i.e., active objects) here. TheDirec-



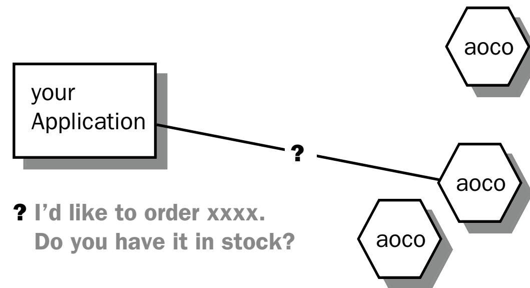
tory contains information about these various aocos and their products that the application uses to contact each of the appropriate aocos. The application asks for prod-



uct information about a given active object and finally



decides to order it from one of the vendors. Once all the



active objects have been purchased and started up, the application which actually rented them (which is now attached, as a peer, to each of the active objects) can make the attachments. This has the advantage that a given application only needs to know how to find a running copy of theDirectory — everything else is determined at run-time. Also, as one would expect, both theDirectory and the aocos are implemented as active objects themselves.

Notice that the criteria that a given application might use to decide which particular vendor to choose from is not prescribed. This allows the applications and aocos to use whatever currency they wish to base their economics of computation on. The application might base its decision on the particular product info that a given vendor supplies (i.e., this vendor offers a version of that particular active object that guarantees high quality service, etc.), or it might make its decision based on some information about the particular aoco (it's running on some resource that the application likes to do business with, or it's lightly loaded, or nearby, etc.). The infrastructure is in place, but the methodology is left open-ended.

DRAFT