PHISH-Nets:
Planning Heuristically In Situated Hybrid Networks

by
Bradley James Rhodes

Bachelor of Science EECS
Massachusetts Institute of Technology
June 1992

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
MEDIA ARTS AND SCIENCES
at the
Massachusetts Institute of Technology
September 1996

© Massachusetts Institute of Technology, 1996
All Rights Reserved

Signature of Author

Program in Media Arts and Sciences
August 9, 1996

Certified By

Pattie Maes
Associate Professor of Media Technology
Program in Media Arts and Sciences

Accepted by

Stephen A. Benton
Chairperson
Departmental Committee on Graduate Students
Program in Media Arts and Sciences
PHISH-Nets:
Planning Heuristically In Situated Hybrid Networks

by
Bradley James Rhodes

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on August 9th, 1996
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
in
MEDIA ARTS AND SCIENCES
at the
Massachusetts Institute of Technology

Abstract

Autonomous characters in interactive story systems are faced with challenges similar to those faced by both symbolic planners and situated-agent architectures. This thesis presents a new architecture based on the Agent Network Architecture developed by Pattie Maes. This new architecture, called PHISH-Nets, is especially well suited for the creation of characters with personality, who must satisfy multiple interacting goals in real-time in an only partially predictable dynamic environment. The algorithm has been used to implement The Big Bad Wolf, an autonomous character in a 3D graphical animated world based on The Three Little Pigs. The PHISH-Nets algorithm is explained in detail, and is informally verified in a series of experiments constructed within the graphical world. The results of the experiments are discussed, and limitations of the algorithms are presented along with possible future directions for this work.

Thesis Supervisor: Pattie Maes
Title: Associate Professor of Media Technology
PHISH-Nets: Planning Heuristically In Situated Hybrid Networks
by
Bradley James Rhodes

The following people served as readers for this thesis:

Reader

[Signature]

Joseph Bates
Professor of Computer Science
Carnegie Mellon University School of Computer Science

Reader

[Signature]

Janet H. Murray, Ph.D.
Senior Research Scientist, MIT Humanities Department
Director, Laboratory for Advanced Technology in the Humanities
Acknowledgments

First and foremost, I would like to thank my advisor Pattie Maes, for her inspiration and advice in the past four and a half years. Most importantly, I would like to thank her for her trust and willingness to let me run with a crazy idea, while still giving guidance. I feel quite lucky to have her as an advisor.

I would also like to thank Bruce Blumberg for the use of his Hamsterdam system, his brainstorming sessions, and for his countless hours of pouring over my buggy code, explaining and re-explaining his system until I finally groked it. Without his help this work would have never had a face.

In addition to being a reader, Joe Bates has done much to further the merger between AI and story-telling, and his thoughts on this synthesis have been a great influence.

Janet Murray has been a good source of ideas both as a reader and a teacher, and has continued to be a reminder that this work is about autonomous characters first, and autonomous agents second.

Jang Kim programmed most of the motor functions for the wolf and pig, and has been a great help. His “alternative death scenes” has pushed this project from rated-G to rated PG-13.

Elizabeth Manicatide designed the 3D renditions of The Big Bad Wolf and three little pigs, and her artistic skill has added much to this work.

Barbara Hayes-Roth has had infinite patience with this project, which started three years ago as a three-month independent research project under her direction. Sorry I’m so late handing it in — I guess the problem was harder than I expected.

Alan Wexelblat provided extensive editing of this document, for which I am very grateful. Alan has a way of eliminating every third word of a document, leaving the finished product much more readable than before.

Marvin Minsky’s ideas have been a great influence on this work, and his Society of Mind theory is what turned me on to this field.

Thanks to my brother, who over the course of a three-hour train-trip not only managed to understand my entire thesis but also to solve the my two remaining difficulties with the algorithm.

Thanks to my Mom, who has supported me in everything I’ve done and made me believe I could do anything.

Finally thanks to my Dad, who not only suffered through the first draft of this document, but whose support (financial and otherwise) has brought me where I am today. His advice, personal and professional, continues to be a guide. Two down, one to go!
# Table of Contents

1: Introduction ......................................................... 13

1.1 Motivation ....................................................... 13
1.2 Characters, not agents ............................................ 14
1.3 The domain addressed by this thesis .......................... 15
1.4 Roadmap to the thesis ............................................ 18

2: Related Work ....................................................... 20

2.1 Agent Network Architecture .................................... 20
2.2 Hamsterdam ...................................................... 22
2.3 Hap .......................................................... 23
2.4 The Virtual Theater Project .................................... 24
2.5 Real Time Responsive Animation ............................... 24
2.6 Terzopolous' Artificial Fishes .................................. 25

3: The PHISH-Nets Architecture ................................. 26

3.1 High-level specification language ............................ 26
  3.1.1 Behavior Specification Modules ......................... 27
  3.1.2 Goals .................................................... 30
  3.1.3 Variables vs. indexical functional aspects ............. 31

3.2 Compiled PHISH-Nets ........................................... 32
  3.2.1 Modules and links ....................................... 32
  3.2.2 An example link ........................................ 41
  3.2.3 Tagging suggestions with source information ........... 42

3.3 Adjustable Parameters ........................................ 44

3.4 The action-selection loop ..................................... 46
  3.4.1 Initialization phase ..................................... 46
  3.4.2 Module-passing phase ................................... 47
  3.4.3 Conflictor link phase ................................... 49
  3.4.4 Resolution phase ....................................... 50
  3.4.5 Special cases phase .................................... 51

3.5 A simple example .............................................. 53
  3.5.1 Example initialization phase ............................ 53
  3.5.2 Example module-passing phase ........................... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.3</td>
<td>Example conflictor-link phase</td>
<td>56</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Example resolution phase</td>
<td>58</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Example special cases phase</td>
<td>59</td>
</tr>
<tr>
<td>3.6</td>
<td>Avoiding thrashing caused by failed action</td>
<td>60</td>
</tr>
<tr>
<td>3.7</td>
<td>What to do when things still go wrong</td>
<td>61</td>
</tr>
<tr>
<td>4:</td>
<td>Experiments and Results</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Experiment 1: single goal and single method</td>
<td>64</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Setup</td>
<td>64</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Result</td>
<td>65</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Variations</td>
<td>65</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Discussion</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>Experiment 2: Multiple methods / complimentary goals</td>
<td>66</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Setup</td>
<td>66</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Result</td>
<td>66</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Variations</td>
<td>66</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Discussion</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Experiment 3: Multiple Inhibitory Goals.</td>
<td>67</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Setup</td>
<td>67</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Result</td>
<td>67</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Variations</td>
<td>68</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Discussion</td>
<td>69</td>
</tr>
<tr>
<td>4.4</td>
<td>Experiment 4: Undoing one goal to accomplish another</td>
<td>70</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Setup</td>
<td>70</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Results</td>
<td>70</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Variations</td>
<td>71</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Discussion</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Experiment 5: Mutual inhibition</td>
<td>71</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Setup</td>
<td>71</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Results</td>
<td>72</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Discussion</td>
<td>72</td>
</tr>
<tr>
<td>4.6</td>
<td>Experiment 6: Opportunism</td>
<td>73</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Setup</td>
<td>73</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Results</td>
<td>73</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Variations</td>
<td>73</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Discussion</td>
<td>74</td>
</tr>
<tr>
<td>4.7</td>
<td>Experiment 7: Failure of actions</td>
<td>74</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Setup</td>
<td>74</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Results</td>
<td>74</td>
</tr>
</tbody>
</table>

**Table of Contents**

3.5.3 Example conflictor-link phase ................................ 56
3.5.4 Example resolution phase ........................................ 58
3.5.5 Example special cases phase ..................................... 59
3.6 Avoiding thrashing caused by failed action ..................... 60
3.7 What to do when things still go wrong .......................... 61

4: Experiments and Results .............................................. 63
4.1 Experiment 1: single goal and single method .................... 64
4.1.1 Setup .................................................................. 64
4.1.2 Result ................................................................ 65
4.1.3 Variations ................................................................ 65
4.1.4 Discussion ................................................................ 66
4.2 Experiment 2: Multiple methods / complimentary goals .......... 66
4.2.1 Setup .................................................................. 66
4.2.2 Result ................................................................ 66
4.2.3 Variations ................................................................ 66
4.2.4 Discussion ................................................................ 67
4.3 Experiment 3: Multiple Inhibitory Goals ......................... 67
4.3.1 Setup .................................................................. 67
4.3.2 Result ................................................................ 67
4.3.3 Variations ................................................................ 68
4.3.4 Discussion ................................................................ 69
4.4 Experiment 4: Undoing one goal to accomplish another ........ 70
4.4.1 Setup .................................................................. 70
4.4.2 Results ................................................................ 70
4.4.3 Variations ................................................................ 71
4.4.4 Discussion ................................................................ 71
4.5 Experiment 5: Mutual inhibition .................................... 71
4.5.1 Setup .................................................................. 71
4.5.2 Results ................................................................ 72
4.5.3 Discussion ................................................................ 72
4.6 Experiment 6: Opportunism .......................................... 73
4.6.1 Setup .................................................................. 73
4.6.2 Results ................................................................ 73
4.6.3 Variations ................................................................ 73
4.6.4 Discussion ................................................................ 74
4.7 Experiment 7: Failure of actions .................................... 74
4.7.1 Setup .................................................................. 74
4.7.2 Results ................................................................ 74
### Table of Contents

4.7.3 Discussion ........................................... 75  
4.8 Experiment 8: Giving up on the impossible ................................. 76  
4.8.1 Setup .................................................. 76  
4.8.2 Results ................................................ 76  
4.8.3 Variations ............................................. 76  
4.8.4 Discussion ............................................ 77  

5: Discussion ............................................... 79  
5.1 Producing Personalities using PHISH-Nets ................................. 79  
5.2 Extensibility ............................................... 80  
5.3 Speed of the algorithm .......................................... 80  
5.4 Comparison with the Agent Network Architecture ....................... 83  
5.5 Limitations and future work .................................. 84  
5.5.1 Lack of global knowledge ................................ 84  
5.5.2 Limitations on types of actions ............................... 86  
5.5.3 Deadlines and other hidden constraints on actions .................. 87  
5.5.4 Performing multiple actions simultaneously ....................... 87  
5.5.5 Detecting when to give up only at the last minute .................. 88  
5.5.6 Detecting failure and breakdowns ................................ 89  
5.5.7 Limitations of the current implementation ......................... 91  

6: Conclusion ............................................. 92  

A: The full specification for the wolf ................................ 93  

Bibliography ............................................... 103
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>The behavior module for Eat-object</td>
<td>28</td>
</tr>
<tr>
<td>Table 2</td>
<td>The behavior module for Pickup-object</td>
<td>28</td>
</tr>
<tr>
<td>Table 3</td>
<td>The behavior module for Walk-to-location</td>
<td>29</td>
</tr>
<tr>
<td>Table 4</td>
<td>The behavior module for Drop-object</td>
<td>29</td>
</tr>
<tr>
<td>Table 5</td>
<td>The specification for the goal Hunger</td>
<td>30</td>
</tr>
<tr>
<td>Table 6</td>
<td>Example energies of instantiated behaviors</td>
<td>58</td>
</tr>
<tr>
<td>Table 7</td>
<td>Example energies of executable instantiated behaviors</td>
<td>59</td>
</tr>
<tr>
<td>Table 8</td>
<td>Variables</td>
<td>81</td>
</tr>
<tr>
<td>Table 9</td>
<td>Assumed constants</td>
<td>82</td>
</tr>
<tr>
<td>Table 10</td>
<td>Intermediary values</td>
<td>82</td>
</tr>
<tr>
<td>Table 11</td>
<td>Order of speed for each part of the algorithm</td>
<td>83</td>
</tr>
<tr>
<td>Table 12</td>
<td>The behavior module for Eat-object</td>
<td>93</td>
</tr>
<tr>
<td>Table 13</td>
<td>The behavior module for Pickup-object</td>
<td>93</td>
</tr>
<tr>
<td>Table 14</td>
<td>The behavior module for Explode-object</td>
<td>94</td>
</tr>
<tr>
<td>Table 15</td>
<td>The behavior module for Walk-to-location</td>
<td>94</td>
</tr>
<tr>
<td>Table 16</td>
<td>The behavior module for Pogo-to-location</td>
<td>94</td>
</tr>
<tr>
<td>Table 17</td>
<td>The behavior module for Climb-down-chimney-to-location</td>
<td>95</td>
</tr>
<tr>
<td>Table 18</td>
<td>The behavior module for Drop-object</td>
<td>95</td>
</tr>
<tr>
<td>Table 19</td>
<td>The behavior module for Blow-down-object</td>
<td>95</td>
</tr>
<tr>
<td>Table 20</td>
<td>The behavior module for Pack-TNT-at-location</td>
<td>96</td>
</tr>
<tr>
<td>Table 21</td>
<td>The behavior module for Explode-object</td>
<td>96</td>
</tr>
<tr>
<td>Table 22</td>
<td>Links of the full wolf, with prominent filters and transformations</td>
<td>98</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: The Big Bad Wolf, a PHISH-Nets Character ......................... 16
Figure 2: The goal to be carrying a pogo-stick ............................. 32
Figure 3: The behavior module Eat-Object ................................. 34
Figure 4: The binding for a pogo-stick .................................... 35
Figure 5: The suggestion to eat the pig .................................. 35
Figure 6: The predecessor link between Pickup-Object and Walk-to-Location ... 36
Figure 7: The simple wolf as a compiled PHISH-Net ....................... 40
Figure 8: ANA when activation-energy is divided evenly among recipients ... 43
Figure 9: ANA when full activation-energy is given to each recipient ........ 43
Figure 10: Default values for adjustable parameters ........................ 46
Figure 11: Initial conditions for Experiment 1 ............................. 65
Figure 12: The wolf about to destroy the house .......................... 68
Figure 13: The wolf climbing down the brick house chimney ............ 75
Figure 14: The wolf's predecessor links .................................... 101
Figure 15: The wolf's conflictor links ................................. 102
Chapter 1: Introduction

1.1 Motivation

There are two primary approaches to creating action-selection architectures for autonomous agents. The classical planning approach assumes that an agent has an internal representation of actions, goals, and the state of the world, and this representation is used to plan a sequence of actions to achieve these goals. The plan is handled by an execution process that more or less blindly performs the specified actions. Classical planners can produce near-optimal sequences of actions to satisfy multiple goals in complex environments. However, they tend to fail in dynamic environments or when the state of the world is not knowable [Brooks, 1990; Agre & Chapman, 1987]. For this reason, they are best suited for bounded environments, such as chess. Situated-agent architectures, on the other hand, emphasize direct coupling of perception to action, decentralization, and dynamic interaction with the environment [Maes, 1990]. Situated agents are designed to perform “good enough” actions in an unpredictable environment with incomplete knowledge. However, most situated architectures can not satisfy multiple goals nor perform complex sequences of actions to achieve a goal. They tend to be best suited for lower-level problems in robotics, where the complexity and unknowability of the real world prevents planning a sequence of actions based on an internal model [Brooks, 1991].

There are many environments that fall into neither of these categories but share features of each. In interactive story systems, for example, autonomous agents and human players both play characters in a narrative, and the story progresses partially as a result of character
actions [Kelso, 1992]. Agents operating in such an environment must have abilities of both reactive robots and chess-players. Like robots, virtual characters must act in real-time in a rapidly changing and unpredictable world. However, to be believable and interesting a character must be able to perform complex sequences of actions and must at least appear to be motivated by internal goals.

1.2 Characters, not agents

Most autonomous agent systems are designed to satisfy one or more goals in a given environment. How these goals are accomplished is usually secondary to getting the job done. Virtual characters, on the other hand, have more than goals and plans: they have biases, fears, styles, hidden motivations, and a host of other traits that make up an individual personality. These traits affect how a character acts — and it is through these actions that an audience gains insight into a character’s personality.

For example, there is one scene in the Walt Disney movie, Cinderella, in which the two mice Jacques and Gus are trying to collect kernels of corn without being caught by the cat. Jacques, the wiser, experienced mouse, runs out, grabs a few kernels and runs back to his hole. Gus, the new-comer to the mouse-hole, has not yet learned to be fearful of the cat, and is also much fatter and greedier than Jacques. Gus runs out of the hole and starts collecting every kernel in sight. As he staggers back under his heavy load, he sees one more kernel and tries to pick it up as well, upsetting his already precarious pile and dropping his entire load. Even as the cat was pouncing towards him, Gus furiously tried again to pick up every last kernel. The two mice had the same primary goal, to collect the corn, but because of their personalities they had different ways of achieving this goal.

Gus does not act in ways that optimally achieves his primary goals, and these foibles are what make him a likable and interesting character. However, it is not enough to simply
make an error-prone agent and call it a character. If this were the case, the job would be
easy! Rather, it is the complex melding of multiple, often conflicting motivations that
make a personality. Jacques acted as he did because of the interaction between his hunger,
his fear of cats, his desire to not take risks, his desire to make the cat looks stupid, and his
desire to help people. Gus has a much higher relative desire to get every last kernel of
corn, and this difference in desires shows through his actions.

Because a character's personality shines through his actions, convincing, believable
characters for interactive story systems must be based on a good action selection architec-
ture. This architecture must be able to integrate multiple motivations in choosing a char-
acter's actions. The architecture must also be able to choose between many different
methods to achieve the same goal, because it is the choices between these different meth-
ods that best display a character's personality. These and other desirable traits in an action-
selection algorithm for interactive story systems are discussed below.

1.3 The domain addressed by this thesis

Even among action-selection architectures specifically designed for characters in inter-
active story-systems, there are many different approaches. Some architectures try to make
characters with animal-like behavior and thus look to ethology for inspiration [Terzopou-
os, 1995; Blumberg, 1994]. Other architectures, such as earlier versions of the OZ
project's HAP system [Loyall, 1991], are designed for creating more human-like charac-
ters that can handle complicated sequences of actions in environments where the effects of
actions are atomic and the geometry of the world is discrete. This type of environment is
commonly found in text-based interactive stories, where the single action walk-north will
move to a completely new room or location. These and related architectures are described
in Chapter 2.
The algorithm described in this thesis, called PHISH-Nets (Planning Heuristically In Situated Hybrid Networks), is designed to create characters that interact with human players in a graphical environment. These characters would not necessarily be human (language, for example, is completely ignored by the algorithm). However, neither would they be completely animalistic in how they act to satisfy their goals. The type of character envisioned by this algorithm is closest to Wile E. Coyote from the old Warner Brothers *Coyote / Roadrunner* cartoons, who, although he does not speak, can still make very complicated plans to catch the Roadrunner.

These characters should have a large repertoire of actions, many of which could potentially, but perhaps unreliably, satisfy the same goal. They should also be able to have multiple, possibly conflicting goals. Finally, they should be able to produce complex sequences of actions to achieve these goals. All these traits make a character more interesting over time and give it greater flexibility to act in ways that are “in character.”
Because human players are involved, a character must also be robust in an unpredictable world and handle failure gracefully.

While PHISH-Nets are designed with the application of interactive story systems in mind, they can be more generally applied to create characters that operate in environments with the following traits:

- The world is continuous. In such a world, character actions are usually fine-grained, where a single instantiation of walk might only take a single step forward.
- Because actions take time to accomplish, the environment might change before an action is completed, forcing the character to change his plan.
- The world is dynamic and unpredictable. Character actions might not have the effects intended, or they may fail entirely.
- Characters have multiple, possibly conflicting goals.
- Characters have multiple means to accomplish the same thing.
- Characters must often perform long sequences of actions in order to accomplish a goal. Sequences of five to ten actions should not pose a difficulty.
- Characters should be easily extendable. New actions, objects, and situations should easily be added to a character or the world without forcing an author to hand-code how each new element relates to the old.

There are two things this thesis is specifically not trying to address. First, PHISH-Nets is not proposed as an explanation for the internals of real animals or humans. Second, the graphical characters created should not be mistaken for simulations of physical robots which might be built. Although the algorithm may be used for physical robots if the domain matches the above traits, the graphical characters created are an end themselves rather than a simulation.
1.4 Roadmap to the thesis

Chapter two gives a brief introduction to the related work in the field of autonomous virtual characters. It includes a lengthy description of Maes' *How To Do The Right Thing* algorithm, which was the starting point for this work. It also briefly describes the behavior systems in *Hamsterdam*, *HAP*, and Minsky's *Society of Mind*, as well as Perlin's real-time graphical puppets and Terzopoulos' artificial fishes, which focus more on motion and realistic graphics than on behavior.

Chapter three describes in detail the PHISH-Nets algorithm. It first describes the high-level character-specification language, which is used by an author to encode a character's actions and goals. It then describes the structure of a PHISH-Net, the network that is compiled from this high-level specification. The action-selection loop is described along with an example. Finally, the rather extensive system for detecting and handling failure conditions is explained.

Chapter four describes test scenarios that have been run using the implementation of PHISH-Nets. These test-runs all take place in a 3D graphical interactive story based on *The Three Little Pigs* story. In them, the Big Bad Wolf tries to satisfy his hunger and urge for wanton destruction through a series of scenarios that demonstrate the algorithm's ability to handle multiple methods for solving a problem, multiple interacting goals, opportunistic behavior, and repeated failure of actions leading to eventually giving up on a goal entirely.

Chapter five discusses the creation of characters with personality, the extensibility of PHISH-Nets characters, and speed of the algorithm. It also compares the algorithm with its closest kin, Maes' Agent Network Architecture (ANA). The chapter closes with a dis-
cussion of the limitations currently inherent in the algorithm and future work yet to be performed.

Chapter six concludes with a brief summary of this thesis.
Chapter 2: Related Work

A few things can be said in general about agent architectures designed for real-time execution in dynamic environments. Because agent architectures must act quickly with partial or inaccurate knowledge of the world, none of them necessarily produces optimal solutions to a problem. Instead, they are designed to produce “good enough” actions quickly and to change plans whenever necessary. To achieve the requisite speed, almost all have a reactive component that relies heavily on precomputed mappings of situations onto appropriate actions. The remainder of this chapter discusses agent architectures that have been used or could be used to produce characters in interactive story systems. It focuses on these architecture’s advantages and limitations.

2.1 Agent Network Architecture

Maes [Maes, 1989; Maes, 1991] describes a hybrid architecture called an “Agent Network Architecture” (ANA), in which the selection of a particular behavior is an emergent property of a parallel process. Her architecture was the starting point for the PHISH-Nets architecture, so there are many similarities between them.

ANA was inspired by Minsky’s Society of Mind theory [Minsky, 1986], which states that the mind is made up of a society of interacting parts. Each of these parts, on its own, is mindless, but intelligent behavior emerges from the interaction among these parts. In an agent network, a character is described in terms of the actions that character can perform. These actions are low-level behaviors such as “walk towards food” and do not on their own exhibit intelligent behavior. Actions are selected by a spreading activation mechanism
through a network where each action is a node. These nodes are linked together at compile time with links that represent causal relations among the actions in a character’s repertoire.

As an agent operates in the world, so-called “activation energy” comes into the network from both internal goals and from the state of the world. This energy is then spread through the network by a process that makes activation energy accumulate in the most relevant behaviors. When an executable behavior acquires activation energy beyond a certain threshold, it is enacted.

The architecture has been shown to be fast, robust, and flexible in a dynamic and unpredictable environment, and it can also handle multiple goals gracefully [Maes, 1991]. ANA networks are also compiled from a relatively straightforward description of actions, and are thus both easy to create and to extend. However, they do not support parameterized actions such as \textit{Walk-to <location>}, and as a consequence have trouble acting in environments where there are complex interactions among possible behaviors. They can also get stuck in loops where the same action is performed over and over, especially if there are no methods to satisfy an action’s preconditions, or if all such methods have failed.

Tyrrell [Tyrrell, 1994] also points out several shortcomings in ANA. Most notably, he points out that the architecture favors appetitive (preparatory) actions over consummatory (achieving) actions. He also demonstrates a flaw in division of energy among behavior-nodes that causes actions that accomplish many goals to be starved out. A similar flaw penalizes actions whose preconditions can be solved in many ways. These last two flaws will be discussed in more detail in Section 3.2.3.

The PHISH-Nets architecture is inspired by ANA, but addresses the above shortcomings so it can be used to create virtual characters for interactive story-systems.
2.2 Hamsterdam

The Hamsterdam behavior system [Blumberg, 1995] is similar to ANA in that activation energy is passed between nodes representing behaviors, but these nodes are arranged in a hierarchy rather than in a flat network. The algorithm is based strongly on ethology\(^1\), and is therefore extremely well suited to the creation of virtual animals. The Hamsterdam architecture is used in the ALIVE [Maes et al., 1995] project, which situates a human in a virtual 3D environment with animated characters, and has also been used in the interactive story “Dogmatic” [Galyean, 1995].

Hamsterdam was designed to operate in an uncertain world with incomplete knowledge, handle multiple goals, and to act appropriately when actions do not have the desired effect. It is also designed to work in a continuous world where actions are not atomic and take time to accomplish. However, to add a new behavior an author must hand-code different situations in which the action can be used. Furthermore, unlike ANA, Hamsterdam is primarily a reactive system, and is not designed to handle complex sequences of actions to achieve a goal. This characteristic makes it a less attractive architecture for designing characters that must engage in sophisticated planning-style tasks.

Blumberg’s Hamsterdam system also implements a sophisticated motor system for controlling graphical characters that incorporates blending of simultaneous actions, animation of focus-of-attention cues, and incorporation of emotional state into the performance of actions. This motor system is being used to implement the graphics and low-level actions of characters in the test runs of this thesis, even though the Hamsterdam behavior system is not used.

---

1. The study of animal behavior

22
Chapter 2  

2.3 Hap

The Hap reactive agent architecture [Loyall, 1991] is a part of the Oz project at CMU and was designed specifically to create characters in interactive story environments. Hap creates an “active plan tree” which consists of a sequence of actions necessary to achieve a goal. Each time step, the active plan tree is first pruned of all plans that are nonsensical and goals that have been spontaneously fulfilled. Then one of the goals from the tree is chosen to be executed, on the basis of hand-coded goal priorities. Finally, if the chosen goal is a primitive action, it is executed; otherwise subgoals are indexed from a plan memory and placed into the active plan tree for subsequent processing. The HAP architecture has been modified to provide real-time control of three graphical characters, called woggles, in an animated world titled Edge of Intention [Loyall, 1993]. This real-time system can handle continuous spaces, and also blends actions together such that the animation for an action changes depending on what action is coming next. It also incorporates a model of emotion that affects both the actions a character takes and the manner in which those actions are performed.

Hap has several of the features described in Section 1.3. The architecture is designed to be opportunistic, handle multiple goals, specify sequential or parallel plans for accomplishing a goal, and to act appropriately when actions do not have the desired effect. It is also designed such that new plans can be easily added without affecting existing plans. However, there are a few limitations to the architecture, especially with respect to goal arbitration. Hap uses hand-coded priorities to arbitrate between multiple top-level goals, but does not take into account how actions may affect less important goals. Opportunistic behavior is also limited to taking advantage of situations where a high-priority goal is spontaneously fulfilled; Hap will not take advantage when a lower-priority subgoal is sat-
isfied. For example, if a character came across a water fountain when it was more hungry than thirsty, it would still look for food rather than drink. These are some of the limitations that the PHISH-Nets algorithm is designed to overcome.

2.4 The Virtual Theater Project

Stanford’s Virtual Theater Project has applied the BB1 blackboard architecture [Hayes-Roth, 1995] to creating virtual characters for interactive story systems. Unlike HAP, ANA, or the PHISH-Nets architecture, this architecture’s primary focus is the production of actions that are in character and appropriate for a character’s mood, rather than producing actions that achieve goals. Interaction with a character is structured around the model of a virtual stage, in which a human director gives general guidelines to a character in terms of a loose script. For example, a character might be given the script to “Stand at the wall for five seconds, then play until a certain event happens, then sit down” [Hayes-Roth, 1996]. These scripts can either be written out in advance (virtual actors) or produced in real-time by a human director (virtual puppets). Characters take the general human-provided instruction such as “play,” and within those constraints act in character, e.g., by either playing energetically or quietly. The system has a simple model of social interaction that characters use to infer the mental state of other characters. It also has a simple emotional model that is used to color a character’s actions.

2.5 Real Time Responsive Animation

Ken Perlin at NYU has developed what he calls “real time graphic puppets.” [Perlin, 1995] These animated characters can perform hand-coded actions while moving in a life-like manner. However, the characters do not encode any real intentionality: they only encode a visual impression of personality and life-like behavior. Although this work could
provide an alternate implementation for a character's low-level motor functionality, it does not provide an action selection mechanism beyond a low-level scripting system.

2.6 Terzopolous' Artificial Fishes

Demetri Terzopoulos at the University of Toronto has developed a physically based simulation of artificial fishes [Terzopolous, 1995]. His work concentrates primarily on realistic modeling of fish motion and bodies, but he has also developed a simple architecture to produce fish-like behavior. His action selection algorithm uses a few very simple hierarchical rules to choose between basic intensions such as avoid, escape, and mate, based on parameters such as fear and libido. While this behavior system works well for producing fish-like behavior, it does not scale well to handling sophisticated sequences of actions where multiple actions can accomplish a single goal.
Chapter 3: The PHISH-Nets Architecture

An autonomous agent or virtual character in the PHISH-Net architecture is divided into five basic units: a sensor-processor, a set of behaviors, a set of goals and their relative importance, a control unit, and actuators. The sensor-processor converts raw sensory data into high-level boolean state information such as *I'm-at-location <location-arg>*. This state information is updated every time-step. The behaviors in a system are the actions the agent can take. They are domain-dependent programs that perform relatively simple low-level actions on the environment (e.g., *Walk-to <location-arg>*). These behavior primitives are treated as black boxes and can be implemented in hardware, hand coded, or even implemented as another PHISH-Net. While behaviors in ANA take no arguments, PHISH-Net behaviors may take one argument. The control unit is the part that, given the agent's goals and the state of the sensor-predicates, decides in which of the behaviors to engage. Finally, actuators implement the chosen behavior. The PHISH-Net architecture is concerned primarily with the control unit and to some extent with the set of behaviors and goals made available to a character. The sensor-processor and the actuators are assumed, but are not dealt with by the algorithm.

3.1 High-level specification language

A character is written in a high-level behavior specification language that is later converted into a compiled representation. In this high-level specification, each action available to a character is represented by a *behavior module* which specifies when the action can be executed and what its expected results are. The character's possible high-level goals are
also encoded. These behavior modules and goals are then connected into a network at compile time by links that represent causal relations among the behaviors.

3.1.1 Behavior Specification Modules

A behavior module contains the code that executes a behavior's particular action and characterizes the effects of that action. It may take one argument. Four lists of predicates and two other fields specify a behavior:

- A *precondition list* contains predicates that must be true for the behavior to be executable.
- An *add list* contains predicates expected to become true by execution of the behavior. These are only the expected positive effects of an action, used for action selection. This list is not used by the simulated world, and the expected and actual effects of an action may differ.
- A *delete list* contains predicates which are expected to become false by execution of the behavior. These are only the expected negative effects of an action, used for action selection. This list is not used by the simulated world, and the expected and actual effects of an action may differ.
- A *relevant-when list* contains predicates which must be true for the behavior to be relevant. This indicates under what conditions a module will act as expected, as opposed to what conditions must be met for an action to be taken at all. For example, the action *eat-object* will only satisfy hunger when the object is food.\(^1\) All conditions in a relevant-when list must be true for a behavior to be relevant.

- The *argument type* of a module specifies what type of argument the module can take. If the module takes no argument, the dummy variable “blank” is used.

---

\(^1\) This is the equivalent of the *context-condition* in the HAP architecture.
The precondition-ordered field indicates whether it matters what order preconditions are achieved. If this boolean is true, a module will only attempt to satisfy later preconditions when earlier ones are already satisfied (i.e., preconditions will be satisfied in order). 2

An example specification for a simple version of The Big Bad Wolf character from *The Three Little Pigs.* is shown in Tables 1-4. The complete specification for The Big Bad Wolf is given in Appendix A.

**Table 1: The behavior module for Eat-object**

<table>
<thead>
<tr>
<th>Name</th>
<th>Eat-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Eat-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>hunger-satisfied (interesting-objects-from-blank &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Delete-list</td>
<td>hands-free (carried-object-from-blank &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Relevant-when</td>
<td>object-edible &lt;object-arg&gt;</td>
</tr>
</tbody>
</table>

a. This transformation produces every object in the world, given the argument “blank.”

b. This transformation produces the object currently being carried given the argument “blank.”

**Table 2: The behavior module for Pickup-object**

<table>
<thead>
<tr>
<th>Name</th>
<th>Pickup-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Pickup-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>I'm-at-location (location-of &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>Yes</td>
</tr>
<tr>
<td>Add-list</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td>hands-free (interesting-objects-from-blank &lt;object-arg&gt;)</td>
</tr>
</tbody>
</table>

a. This transformation produces the argument “blank” given any object.

2. This distinction between ordered and unordered subgoals is the same as the distinction between sequential and parallel plan nodes in the HAP architecture.
Chapter 3  The PHISH-Nets Architecture

<table>
<thead>
<tr>
<th>Name</th>
<th>Walk-to-location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Walk-to-location]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>location</td>
</tr>
<tr>
<td>Preconditions</td>
<td>[none for the simple version]</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>I'm-at-location &lt;location-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td></td>
</tr>
<tr>
<td>Relevant-when</td>
<td>location-on-ground &lt;location-arg&gt;</td>
</tr>
</tbody>
</table>

Table 3: The behavior module for Walk-to-location

<table>
<thead>
<tr>
<th>Name</th>
<th>Drop-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Drop-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>hands-free (carried-object-from-blank &lt;blank-arg&gt;)</td>
</tr>
<tr>
<td>Delete-list</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Relevant-when</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: The behavior module for Drop-object

A predicate in a behavior module can consist of lone predicate, such as object-in-hand <object-arg>, or a predicate plus a transformation, such as I'm-at-location (location-of <object-arg>). A transformation is a domain dependant function that converts an argument binding into one or more different argument bindings. For example, a transformation could convert an object into the locations of that object, convert a location into the set of all objects blocking the path to that location, or convert an object into itself (i.e. no change). Lone predicates and transformations in the precondition-list must take the same argument type as their module. A precondition-predicate with a transformation must take the same input type as the output type of the transformation. Predictes that do not need arguments (such as hands-are-empty) take the dummy variable “blank” as an argument. Transformations for predicates in the add or delete lists of a module are the reverse of precondition-predicate transformations: add and delete list transformations must output the same type as
the module and accept as input the same type as the transformations associated predicate.

For example, the predicate hands-free takes type “blank” as an argument. When it appears in the precondition-list of Pickup-object, hands-free is paired with a transformation that turns any object into a blank. On the other hand, when it appears in the add-list of Drop-object, hands-free is paired with a transformation that turns a blank argument into the object currently being carried.

### 3.1.2 Goals

Goals consist of a strength value and predicate to make true. For example, the goal “hunger” might be represented by the precondition hunger-is-satisfied with a strength of 50. Goals always have a base argument type of “blank,” though their predicate can have transformations just like preconditions. For example, carrying-object (blank-to-pogo-stick-transformation <blank-arg>) would represent the desire to be holding a pogo-stick.

The strength of a goal can change over time as an agent’s internal state and environment change. At runtime, goals feed activation energy to modules that best satisfy them by a mechanism explained in Section 3.4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Hunger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument Type</td>
<td>blank</td>
</tr>
<tr>
<td>Predicate-to-make-true</td>
<td>hunger-satisfied</td>
</tr>
<tr>
<td>Goal Value</td>
<td>22</td>
</tr>
</tbody>
</table>

**Table 5: The specification for the goal Hunger**

a. Goals never take arguments, so this field is always of type “blank”

---

3. For example, a character’s hunger might decrease because he has just eaten (a change in the environment) or because he has gotten distracted (a change in the character’s internal state). Changes due to the environment would be performed by the world’s physics layer; changes due to the character’s internal state would be performed by the implementation of individual actions.
3.1.3 Variables vs. indexical functional aspects

ANA does not incorporate classical variables and variable-passing, and Maes [Maes, 1989] argues that many of the algorithm's advantages would disappear if variable-passing was introduced. Maes suggests that *indexical-functional aspects* [Agre, 1987] be used instead of classical variables. The idea is that behaviors are specified relative to the current situation of the agent rather than to absolute identities of objects. For example, instead of specifying a behavior as *Goto-location (x,y)*, one would specify the location in terms of the functional aspects desired, as in *Go-to-the-nearest-exit-from-the-room*.

Indexical-functional aspects reduce the problem space by differentiating only those objects, locations, and states that have unique and relevant functions. However, in complex environments the number of unique, relevant functions is usually large. For example, a tool-using agent needs more differentiation than *Get-the-nearest-tool*, because the action *Pound-a-nail* requires a hammer, whereas *Drive-a-screw* requires a screwdriver. Furthermore, even if several real-world objects could be conflated into a single function, extensions to the agent could later require a reworking of these groupings [Rhodes, 1995].

Behaviors in PHISH-Nets use a mixture of indexical-functional aspects and classical variable-passing. Each behavior may take a single argument of a given type, such as "object" or "location." The set of all possible arguments is kept manageable by only specifying those that are "interesting." For example, in the current implementation of The Big Bad Wolf, the only interesting locations are special locations, the locations of objects, and the locations of other characters. However, after a location is declared interesting it is still encoded by (X, Y, Z) coordinates, rather than by a functional aspect such as "location of screwdriver."
3.2 Compiled PHISH-Nets

3.2.1 Modules and links

The high-level specification for a character is compiled into a PHISH-Net, which is the actual representation used at runtime to determine a character’s actions. Compiled PHISH-Nets are made up of the following data types:

- **Goals**: Like goals in the high-level specification, goals in a PHISH-Net indicate the conditions a character is trying to satisfy in the world. For example, the goal to be carrying a pogo-stick encourages the character to act in ways that will make the condition \( I'm-carrying-pogo-stick \) become true. Each goal has a value indicating how important the goal is. Each goal also has its own list of predecessor-links from that goal, and a list of conflictor-links from that goal. Predecessor links connect a goal to behavior modules that might satisfy that goal. Conflictor links connect a goal to behavior modules that might undo that goal once it is satisfied. Both kinds of links are described in detail below.

- **Behavior Modules**: Behavior modules (often referred to simply as "modules") represent an uninstantiated action such as *Pickup-object*. Compiled behavior modules consist of a list of suggestions for and against particular

![Diagram](image)

**Figure 2**: The goal to be carrying a pogo-stick

- **Goal Name**: *Carrying Pogo-stick*
  - **Value**: 22
  - **Predecessor-link** to *Pickup-Object*
  - **Conflictor-link** to *Drop-object*
bindings, a list of predecessor-links and conflictor-links, and the effector for this action (the code that is executed when the action is chosen). Modules also contain three blacklists indicating which bindings are currently considered to be “failures,” “given-up on,” and “inhibited.” Blacklists are simply lists of bindings that should be allowed to pass to a module (in the case of the failure-blacklist and giveup-blacklist) or be allowed to pass from a module (in the case of the inhibited-blacklist). When a module’s action is taken and fails to achieve the effect expected, the binding for that action is added to the failure-blacklist for that module, along with a number indicating how many time-steps before the action can be tried again. A similar mechanism for adding bindings to the giveup-blacklist and inhibited-blacklist is described in Section 3.6.
Module Name: Eat-Object
Effector: [Implementation for Eat-Object]

Suggestion-list:
- Suggestion to eat Pig
- Suggestion to eat Bacon
- Suggestion to eat Spam

Failure-blacklist:
- Don’t eat Twinkies for 242 timesteps
- Don’t eat Moon-pies for 122 timesteps

Giveup-blacklist:
- Don’t bother eating caviar for 42 timesteps

Inhibited-blacklist:
- Don’t try to eat Tofu next timestep

Figure 3: The behavior module Eat-Object

- **Bindings**: A binding is an argument with which a behavior module can be instantiated. For example, *pickup-object* can be instantiated with the binding <*pogo-stick*>. A behavior module paired with a binding is called an instantiated behavior, or just a behavior. Bindings are tagged by type, but there is no typechecking in the current implementation.
Suggestions: Suggestions consist of a behavior module and binding that is being suggested, the goal that is the original source of this suggestion, a "hop-count" counter indicating the suggestion's distance from the original suggesting goal and a field indicating whether the suggestion is in favor of or against the instantiated behavior being executed. If a suggestion is in favor of the behavior being taken, it is called a "supportive" suggestion. If it is opposed to the behavior, it is called "inhibitory." As a rule, supportive suggestions are passed down predecessor-links and inhibitory suggestions are passed down conflictor-links. During each initialization phase of the algorithm, suggestions are created by goals and passed down links to behavior modules. When a goal creates a suggestion the suggested-module and suggested-binding fields are both labeled "blank," but these are set later by transformations in the links that pass suggestions to behavior modules.

Suggestion:
- Suggested module: Eat-Object
- Suggested binding: pig
- Source goal: Hunger
- Hop-count counter: 1
- Supportive or inhibitory?: supportive

Figure 5: The suggestion to eat the pig
Links: Links are the conduit through which suggestions are passed among goals and behavior modules. Links have two ends. The front end can be either a goal or a behavior module; the back end is a behavior module. Between the two ends is a series of filters and transformations that convert input suggestions to output suggestions. For example, if a character wanted to pick up a pogo-stick, the filter-series for the link from Pickup-object to Walk-to-location would take the pogo-stick suggestion as input and would output all the locations in which pogo-sticks can be found. The filter-series is explained in more detail later.

Figure 6: The predecessor link between Pickup-Object and Walk-to-Location
At compile-time, predecessor links are created from every goal to every behavior module that can satisfy it, and from every behavior module to every behavior module that can make true one of the from-module’s preconditions. Conflictor links are created from every goal to every behavior module that would undo that goal once it was satisfied, and from every behavior module to every module that can undo one of the from-module’s preconditions. These links represent causal relationships among the behaviors. For example, a predecessor link would exist from Pickup-object to Walk-to-location because the latter satisfies I’m-at-location, which is a precondition of the former. There would also be a link between the goal Hunger and Eat-Object, because the latter satisfies the former, and a conflictor-link between Eat-Object and Drop-Object, because dropping an object undoes a precondition necessary for eating that object.

More formally, there is a predecessor-link between a goal and a module when the condition associated with the goal is in the behavior module’s add-list. There is a predecessor-link between two modules every time the from-module contains a predicate in its precondition-list which matches a predicate in the to-module’s add-list. Similarly, there exists a conflictor-link between a goal and a module when the condition associated with the goal is in the to-module’s delete-list, and between two modules every time a predicate in the from-module’s precondition-list matches one in the to-module’s delete-list. Conflictor links and predecessor-links differ only in their conflictor-transform and condition-filter, described below.

Suggestions are passed from one module (the “from-module”) to another module (the “to-module”) through the following series of filters and transformations. When a filter receives a suggestion, it either passes that suggestion on unchanged to the next filter or transformation in the series, or it “blocks” that suggestion and does not let it pass. For example, a filter might block all suggestions to act with bindings that had failed recently.
Transformations, on the other hand, modify suggestions they receive into different suggestions. Transformations can also convert a single suggestion into multiple suggestions. For example, a transformation might convert a suggestion to act with a particular object into several suggestions, one for each location in which that object might be found. Transformations and filters pass their processed suggestions to the next filter or transformation in the series, in a bucket-brigade fashion. The first filter in the series receives all suggestions contained in the from-module's suggestion-list. The last filter in the series passes all the suggestions it doesn't block to the to-module, which adds those suggestions to its suggestion-list. Each filter and transformation in the filter series modifies suggestions such that only suggestions that are relevant in the character's current situation are passed through a link.

A filter series consists of the following filters and transformation, in order:

- **condition-filter:** The condition-filter blocks all suggestions for which the from-module's preconditions are already satisfied. Blocked suggestions are not passed to filters later in the series and do not affect the suggestion-list of the to-module. This filter insures that a module not try to make true a precondition that is already true. In conflictor-links, this filter instead blocks all suggestions that are *not* satisfied, because only satisfied conditions should be able to inhibit.

- **order-filter:** The order-filter blocks all suggestions for which earlier preconditions in the suggesting module's ordered precondition-list are not yet satisfied. This filter prevents a module from attempting to satisfy its preconditions out of order. If order does not matter for this particular module, the order-filter passes all suggestions through unchanged.
• **inhibition-filter:** This blocks removes all suggestions for bindings that were inhibited in the previous time-step. As described later, inhibited bindings are not passed.

• **counter-update-transform:** This transformation increments the hop-counter: the counter in a suggestion that indicates how many steps away from the original goal the suggestion is. As is explained later, the hop-counter is used to slightly bias the algorithm towards shorter action sequences.

• **precondition-transformation:** This transformation changes a suggestion from one type (e.g., “object”) to another (e.g., “location”). The transformation to use (if any) is specified in the suggesting module’s precondition. For example, the transformation for the first precondition in `pickup-object` takes an object and returns the location of that object. Some transformations produce several outputs for the same input. For example, the `location-of <object-arg>` transformation might return several locations if the object is especially large, or if there are several copies of the same object.

• **add-transformation:** This is the same as the precondition-transformation, except that it is specified in the receiving-module’s add-list (delete-list for conflictor-links). The add-transformation also changes the suggested-module field of each suggestion from the “from-module” to the “to-module” for this link.

• **conflictor-transform:** If in a predecessor-link, this filter simply passes a binding on, unchanged. In conflictor-links it changes a suggestion in favor of executing with an argument into a suggestion opposed to executing with that argument. In this way, goals can inhibit actions with which they clash, and modules can inhibit actions that would make their true preconditions become false.
- **failed-module-filter**: This filter removes all bindings for which the receiving module has previously failed or otherwise given up. Failure of and giving up on modules is discussed in Section 3.6.

- **relevant-when-filter**: This filter removes all bindings for which the receiving module is not applicable. Applicability is determined from the receiving module's relevant-when field. For example, *eat-object* is only relevant when the object in question is a kind of food.

---

**Figure 7: The simple wolf as a compiled PHISH-Net**

Chapter 3  
The PHISH-Nets Architecture
3.2.2 An example link

Assume *Pickup-object* already has several suggestions of objects to pick up in its suggestion-list. These suggestions would either have come from other behavior modules (via those module’s own predecessor links) or directly from goals. *Pickup-object* passes all these suggestions to its own predecessor and conflictor-links, including the link from *Pickup-object* to *Walk-to-location*. In this particular link, the condition-filter blocks suggestions to pick up objects already in the current location of the agent, and passes on the rest of the suggestions. The order in which *Pickup-object*’s preconditions are satisfied matters, but because *I’m-at-location (location-of <object>)* is the first precondition of *Pickup-object*, the order-filter passes all the suggestions through. Then the transformation-filter changes all the suggestion’s bindings from objects (the object to be picked up) into the (x,y,z) coordinates of the location that is to be walked to (the location of that object). If an object is in multiple locations (either because there are several copies or because the object is especially large), this transformation will pass on suggestions for several different locations for every suggestion for an object the transformation receives. After this point in the series, all subsequent filters and transforms will receive suggestions for locations that should be walked to, not suggestions for objects that should be picked up. The add-list for *Walk-to-location* contains no transformations, so it passes suggestions unchanged. The link between *Pickup-object* and *Walk-to-location* is not a conflictor-link, so the conflictor-transform also passes suggestions unchanged. The failed-module-filter then blocks all suggestions for locations that *Walk-to-location* has previously failed (that is, those bindings still on *Walk-to-location*’s blacklist), and passes on the rest. Finally, the relevant-when-filter blocks all locations not at ground-level, since *Walk-to-location* does not work above ground. The remaining suggestions are added to *Walk-to-location*’s suggestion-list. As
will be discussed in Section 3.4, *Walk-to-location* will then repeat the process by passing these new suggestions down its own predecessor and conflictor-links.

### 3.2.3 Tagging suggestions with source information

Suggestions need more information than just argument bindings: they also need to communicate the importance of a suggestion. Maes' Agent Networks Architecture communicates the importance of an action by passing activation energy from goals and state variables to modules and from module to module. However, as Tyrrell points out in [Tyrrell, 1994], there is not enough information maintained by ANA to manipulate energy properly.

Briefly stated, the problem is whether to divide activation energy evenly among all modules that can satisfy a precondition, or to give the full amount to each. If activation energy is divided evenly among all modules that satisfy a precondition, modules with preconditions that many other modules can satisfy are unfairly disadvantaged. For example, if a character was very hungry and slightly thirsty but had 50 different behaviors that could satisfy that hunger, each of those behaviors would get only 1/50th of the energy from hunger. This would cause the character to try to get a drink, even though thirst is only a secondary goal. If, on the other hand, energy is given to each module without dividing by the number of modules receiving that energy, modules that are unique in satisfying a goal are unfairly advantaged. Take, for example, a character that is more hungry than thirsty and has the choice of 50 different drinks from a vending machine. With this method for dividing energy, each behavior representing buying a different drink would get the full amount supplied by thirst. However, if using the vending machine to buy any of these drinks required exact change then all 50 modules would give the full amount of energy to “get change for vending machine.” The act of getting change for the vending machine would
become an all-consuming passion for the character, even though thirst is still only a secondary goal to hunger.

![Diagram of energy distribution for hunger and thirst]

**Figure 8: ANA when activation-energy is divided evenly among recipients**

![Diagram of energy distribution for full activation-energy]

**Figure 9: ANA when full activation-energy is given to each recipient**

The desirable result is for energy to be divided such that actions are favored relative to the number and strength of the goals they help satisfy, regardless of the paths that connect...
actions to goals. For these reasons, in PHISH-Nets every suggestion includes a tag indicating the goal that originally spawned it, rather than an explicit energy value. Energy is only computed when necessary based on these tags, rather than stored as a number. The base energy of a suggestion is given by:

\[
\text{base energy} = \begin{cases} 
G \times S^H, & \text{when suggestion is supportive} \\
-G \times C \times S^H, & \text{when suggestion is inhibitory}
\end{cases}
\]

where \(G\) is the source goal's strength value, \(H\) is the number of hops between the goal and this suggestion, \(S\) is the sequence-length parameter (default equals 0.99), and \(C\) is the conflictor-multiplier parameter (default equals 0.90). These last two parameters are described in Section 3.3.

When \(S\) is significantly less than 1.0, the multiplication by \(H^S\) causes shorter sequences of actions to be favored. When the sequence-length parameter is close to 1.0 the bias caused by this multiplication is completely overpowered whenever competing suggestions have different goal strengths (\(G\)), but still has an effect when two suggestions both have the same source goal, or when their goal sources have the same strength. This bias helps avoid situations where the character chooses a much longer sequence of actions to achieve a goal than necessary.

### 3.3 Adjustable Parameters

The following parameters are available to adjust the functioning of the algorithm. The use of these parameters will be further described in Section 3.4.

- **Sequence-length-multiplier** (default = 0.99): A suggestion’s base energy is multiplied by the sequence-length-multiplier raised to the power of the
number of hops away that suggestion is from its source goal. The closer to 0.0, the more bias there is towards shorter action sequences.

- **Nearness-multiplier** (default = 10): When a behavior module receives an extra boost of energy this nearness-energy ranges between 0.0 and 1.0, times this parameter. A high value of the nearness-multiplier makes the algorithm favor actions that are closer to being accomplished; e.g., going to closer locations.

- **Maximum-number-of-hops** (default = 20): This is the maximum number of steps a suggestion can be away from its goal source before being deleted. This limit puts an upper-bound on the runtime of the algorithm. The default is 20, a value far longer than the length of the longest action sequence in any of the experiments performed.

- **Conflictor-multiplier** (default = 0.9): This parameter indicates the amount by which inhibitory suggestions are multiplied. If it is set to 1.0, one action can keep another of equal energy from acting through inhibition. Lower values give a character more freedom to undo a satisfied goal or precondition if it might mean achieving another. In environments where undoing a satisfied precondition may mean never satisfying it again, this value should be high; lower values allow more dithering and experimentation. However, a value less than 1.0 also opens the possibility of dithering between two near-equal goals. Examples of both possibilities are shown in Section 4.3 and Section 4.4.

- **Failure-penalty-time** (default = 400): The number of timesteps a binding stays on a behavior module’s failure blacklist before it can be tried again. If the world is especially dynamic, this parameter should be lower, in order that the character will try again more quickly. The appropriate value for this parameter also depends on how long a typical action takes in the domain world.
• **Giveup-penalty-time** (default = 80): The number of timesteps a binding stays on a behavior module’s giveup blacklist before that binding can be tried again. This parameter should be a high enough value that the character will change his environment before retrying, to avoid getting into action loops.

• **Inhibition-penalty-time** (default = 1): The number of timesteps a binding stays on a behavior module’s inhibition blacklist before it can be tried again. This should always be set to 1, and is included here simply for completeness.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence-length-parameter</td>
<td>0.99</td>
</tr>
<tr>
<td>Nearness-multiplier</td>
<td>10</td>
</tr>
<tr>
<td>Maximum-number-of-hops</td>
<td>20</td>
</tr>
<tr>
<td>Conflictor-multiplier</td>
<td>0.90</td>
</tr>
<tr>
<td>Failure-penalty-time</td>
<td>400</td>
</tr>
<tr>
<td>Giveup-penalty-time</td>
<td>80</td>
</tr>
<tr>
<td>Inhibition-penalty-time</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 10: Default values for adjustable parameters

### 3.4 The action-selection loop

Every time-step the following actions are taken to determine what a character should do:

#### 3.4.1 Initialization phase

1. All suggestions from the previous time-step are erased.\(^4\)

---

\(^4\) The only state carried over from one time-step to the next is the state of each module’s failure-blacklist, giveup-blacklist, and inhibited-blacklist. Every timestep, a character will re-evaluate what action to take based entirely on the values in these blacklists and the state of the world without any other memory of previous history.
2. The failure-blacklist, giveup-blacklist, and inhibited-blacklist for every module is checked, and bindings that have been on the blacklist for their allotted “penalty time” are removed from the list so they can be tried again.

3. Goals generate blank suggestions that are passed down the predecessor-links that connecting the goal to behavior modules. The suggestions generated by goals do not specify a suggested-binding or suggested-module; these fields in the suggestion are set to “blank,” and will be set later by the precondition-transformation and add-transformation filter of the link from the goal. For example, the goal Hunger would create a blank suggestion to pass down its predecessor link to Eat-Object. The add-transformation of this link puts Eat-object into the module-suggested field of the suggestion. It also passes the suggestion through the transformation “object-from-blank,” which is the transformation specified in the add-list for Eat-Object. This transformation converts the suggestion for a blank binding into several suggestions, one for each interesting object currently in the environment. At the end of the filter series for this link, the relevant-when-filter will block all but suggestions for edible objects. In this way Eat-object’s suggestion-list becomes populated with suggestions to eat every edible object in the environment.

4. At the end of the initialization phase, modules that received new suggestions are marked as “dirty.”

### 3.4.2 Module-passing phase

The module-passing phase is as a loop in which modules pass suggestions among themselves down their predecessor-links:

5. Until all modules in the network are marked “clean”:

---

5. Suggestions are not passed down conflictor-links until the conflictor-link phase, described below.
Chapter 3

The PHISH-Nets Architecture

- For each dirty module pass all suggestions in the module’s suggestion-list down all the module’s predecessor links. The filter-series in each link will insure that only suggestions for actions that are appropriate to the current situation are passed on to the to-module of a link. Specifically, the condition-filter of a link blocks suggestions for bindings for which the from-module is already executable. The order-filter keeps a link from passing suggestions if preconditions before this link are not yet satisfied and order matters. The inhibition-filter blocks bindings for which the from-module is being inhibited. The precondition and add-transforms convert a suggestion into a type the to-module can understand—e.g., from type “object” to type “location.” Finally, the failed-module-filter and relevant-when-filter insure that only suggestions for relevant bindings that haven’t failed recently are passed to the to-module.

- Modules that have received suggestions from links add these suggestions to their suggestion-list, with the following exceptions. If a module receives a suggestion that is a duplicate to one already in the module’s suggestion-list, and if the new copy has a lower hop-count than the old, then the new copy is added to the suggestion-list and the old copy is deleted from the list. Otherwise, the new copy is not added to the suggestion-list. For efficiency, suggestions with a base energy between \(0.1\) and \(-0.1\) are also not added, nor are suggestions that have a hop-count greater than the value of the parameter \(\text{maximum-number-of-hops}\) (default equals 20 hops).

- When a receiving module adds a suggestion to its suggestion-list, the module is marked “dirty.”

---

6. Two suggestions are defined as identical if they both have the same goal, suggested-module, and suggested-binding. Their hop-counts and whether the two suggestion are supportive or inhibitory are ignored.

7. Note that an inhibitory suggestion can replace a non-inhibitory if the former has a lower hop-count than the latter. However, an instantiated behavior cannot support or inhibit itself because any self-supporting or self-inhibiting suggestion will be a copy with a larger hop-count, and thus will be removed.
Once a module passes all its suggestions down its predecessor-links, the module is marked “clean.” Dirty modules continue to be serviced until all modules are clean.

### 3.4.3 Conflictor link phase

In the conflictor link phase, all modules and goals pass their suggestions down their conflictor links. Unlike the module-passing phase, there is no loop because conflictor-links do not propagate more than one hop. The following events happen in the conflictor link phase:

6. Each goal creates a new blank suggestion and passes it down all the goal’s conflictor-links. Note that all suggestions passed down conflictor-links are set to be inhibitory suggestions by the conflictor-transform of the link. These suggestions therefore have a negative base energy.

7. Each module passes all supportive suggestions in the module’s suggestion-list down all the module’s conflictor links. Like the suggestions passed down conflictor-links from goals, these suggestions are turned into inhibitory suggestions and as a result have a negative base energy.

8. The base energies of all suggestions for and against each particular instantiated behavior are summed. This sum is the base energy of the entire instantiated behavior. Two subtotals are also maintained: the sum of the base energy of all supportive suggestions for an instantiated behavior is called the supportive base energy of the behavior. Similarly, the sum of the base energy of all inhibitory suggestions for an instantiated behavior is called the inhibitory base energy of the behavior.

9. Any instantiated behavior whose base energy is negative is declared inhibited and
is added to the inhibited-blacklist for the module associated with the behavior. Instantiated behaviors added to an inhibited-blacklist will remain on the list for the number of timesteps indicated in the \textit{inhibition-penalty-time} parameter, which defaults to one timestep. Any instantiated-behavior on the inhibited-blacklist is marked as not executable this timestep. Furthermore, the inhibited-filter in every link’s filter-series will block any suggestion that contains a binding on the from-module’s inhibited-blacklist, and therefore suggestions for inhibited bindings cannot be passed to other modules.\(^9\)

\section*{3.4.4 Resolution phase}

In the resolution phase the action the character should take this timestep is determined. The following steps occur in this phase:

10. All instantiated behaviors that are not executable are removed from consideration.

   By definition, an instantiated behavior is executable if all its preconditions are true for the binding with which it is instantiated, and if that behavior’s binding is also not on the module’s inhibited-blacklist.

11. Each executable instantiated behavior is given an extra boost of energy associated with how close to completion the action is. For example, each executable binding in \textit{Walk-to-location} would be given extra energy proportional to how close the location is. Actions that are always close to completion, such as \textit{Drop-object}, always receive the maximum boost when executable. This extra energy is called the nearness-

---

8. I.e., more important goals or a larger number of goals are inhibiting the instantiated behavior than are supporting it.

9. Note the difference in the handling of the inhibited blacklist from the failure-blacklist and the giveup-blacklist. If an instantiated behavior is on the inhibited-blacklist of a module, suggestions for that behavior cannot be passed from that module. On the other hand, if an instantiated behavior is on the failure or giveup-blacklists of a module, suggestions for that behavior may not be received by that module. These limitations are handled automatically by the inhibition-filter and the failed-module-filter, respectively.
energy of an instantiated behavior.

12. The total energy of each executable instantiated behavior is computed by summing the behavior's base energy and nearness-energy.

13. The instantiated behavior with the highest total energy is chosen as the action to take, and the effector for the behavior module is executed with the binding indicated in the instantiated behavior.

3.4.5 Special cases phase

There are several kinds of situations a character can get into that require special handling. The steps taken to handle these situations are described in detail in Section 3.6 and Section 3.7, but briefly these steps are as follows:

14. Executable instantiated behaviors are called “passed-over” if their supportive base energy plus nearness energy is higher than the supportive base energy plus nearness energy of the instantiated behavior that was chosen for execution. Passed-over behaviors would have been chosen as the action to execute, were it not for inhibition bringing their total base energy down. Of all the passed-over executable instantiated behaviors, the behavior with the largest amount of inhibition is placed on the inhibited-blacklist for its associated module. Because only one behavior per time-step added to the inhibited-blacklist in this fashion, this step circumvents the situation where two high-energy behaviors mutually inhibit each other, thereby allowing a lower energy behavior to be chosen. An example of this situation is given in Section 4.5.10.

10. A more subtle effect of this step is to insure that a character will give up on taking an action whose preconditions cannot be satisfied due to inhibition. Passed-over behaviors are likely to never get executed, and therefore non-executable behaviors need to know to stop relying on the passed over behaviors for making preconditions become true. This is handled by the give up step, described below and in Section 3.6.
15. If the instantiated behavior that just executed is judged to be a “failure,” the binding of that behavior is added to the failed module’s failed-blacklist. The binding will stay on the failure-blacklist for the number of timesteps specified in the failure-time-penalty, which defaults to 400 timesteps. This step of the algorithm forces a creature to avoid behaviors known to fail. Currently failures are detected on an ad-hoc bases by the physics layer of the implementation rather than by the creature himself. Extensions that would give a character greater flexibility in detecting failures are discussed in Section 5.5.6.

16. If a non-executable instantiated behavior cannot satisfy any of its remaining false preconditions, it is called an unsatisfiable behavior. A behavior might not be able to satisfy its preconditions because the modules that would satisfy them are unavailable, have failed, or are being inhibited to the extent that they will never be chosen for execution. By definition, an instantiated behavior cannot satisfy a precondition if a suggestion for the behavior’s associated binding cannot pass through the filter-series of any predecessor-link from the associated module. Bindings associated with an unsatisfiable instantiated behavior are added to the associated module’s giveup-blacklist. This step of the algorithm forces a creature to avoid trying to satisfy preconditions for behaviors that are hopeless. This kind of situation is discussed further in Section 3.6.

17. In some cases the resolution phase will result in no executable instantiated behavior being available. This condition may exist simply because no behaviors are currently executable at all, or it may be indicative of a failure in the algorithm. To avoid the latter condition, if no action is selected by the resolution phase then the highest energy instantiated behavior that is actively inhibiting another instantiated behavior is added to the giveup-blacklist of the inhibiting behavior’s associated module. If no
such instantiated behavior exists (i.e., no instantiated behaviors are currently inhibiting) then the giveup-blacklists for all modules are cleared. This step of the algorithm is designed to get a character out of a few special degenerate cases described in Section 3.7.

3.5 A simple example

As a simple example consider the network implementing The Big Bad Wolf, as shown in Figure 7 on page 40. The wolf is created from the four behavior modules: Eat-object, Pickup-object, Walk-to-location, and Drop-object; and from the one goal Hunger which has a value of 22. In this example, the world contains the wolf's favorite food, the little pigs Moe, Larry, and Curly, and also contains a pogo-stick. For this example, Moe is at the (x,y,z) location [10, 10, 0], Larry is at [20, 20, 0], and Curly is at [30, 30, 0]. The wolf is lurking nearest to Curly at [35, 35, 0], and he's hungry.11

The main algorithm runs as a continuous loop, producing the "best choice action" every time-step. Following the steps outlined in Section 3.4, we have the following phases and steps of the algorithm:

3.5.1 Example initialization phase

1. All suggestions from previous time steps are erased.

2. All blacklists are checked for bindings that should be removed. Since in this example no bindings are currently on blacklists, no change is made.

3. The goal Hunger creates a suggestion with the suggested-module and suggested-binding fields both set to "blank," the source-goal field set to "Hunger," the hop-count counter set to zero, and the supportive-or-inhibitory tag set to "supportive."

11. Suspenseful, isn't it?
Chapter 3  The PHISH-Nets Architecture

This suggestion gets passed down *Hunger*’s only predecessor-link, to *Eat-object*. Hunger is not currently satisfied, so the link’s condition-filter lets the suggestion through. The order-filter and inhibition-filter don’t matter for links from goals, so these filters also let the suggestion through unchanged. Then the counter-update-transform increments the suggestion’s hop-count counter from zero to one. Because the specification for *Hunger* contains no transformation, the precondition-transformation of the filter-series passes the suggestion through unchanged. However, the add-transformation makes two changes to the suggestion. First, it changes the suggestion’s suggested-module field from blank to *Eat-Object*. Second, it passes the suggestion through the transformation described in the add-list for the high-level specification of *Eat-Object*. In this case, that transformation is *Interesting-objects-from-blank*. Given the one suggestion with the blank suggested-binding field, this transformation produces a suggestion for each object in the world: one suggestion with the suggested binding field set to “Moe,” one set to “Larry,” one set to “Curly,” and one set to “pogo-stick.” All four new suggestions are passed on to the next filter in the filter-series, the failed-module-filter. Because no actions have failed yet in this example, the failed-module-filter passes all four suggestions on unchanged. Finally, the relevant-when-filter blocks the suggestion to eat the pogo-stick, because pogo-sticks aren’t edible. The other three suggestions are allowed to pass out of the filter-series, and all three are added to the suggestion-list of *Eat-object*.

4. New suggestions have been added to *Eat-object*, so the module is marked “dirty.”

3.5.2 Example module-passing phase

5. The following actions occur in the module-passing phase loop:
Since *Eat-object* is dirty, it passes the three suggestions in its suggestion-list down its one predecessor-link to *Pickup-object*. The filter-series increment the three suggestions’ hop-count counter and changes their suggested-module to *Pickup-object*, but otherwise allow the suggestions to pass through unchanged.

- Since *Pickup-object* currently has no suggestions in its suggestion-list and none of the suggestions’ hop-counter is greater than 20, all three suggestions are added to *Pickup-object*’s suggestion-list.

- *Pickup-object* is marked “dirty.”
- *Eat-object* is marked “clean.”

The loop repeats, and this time *Pickup-object* is dirty. All three suggestions are passed down all three predecessor-links leading from *Pickup-object*: the link to *Walk-to-location*, the link to *Drop-object*, and the link to *Eat-object*. In the links to *Drop-object* and to *Eat-object*, all three suggestions are blocked by the condition-filter, because the wolf’s hands are already empty. However, none of the suggestions are blocked in the link from *Pickup-object* to *Walk-to-location*. In this link, the filter-series converts the three suggestions with the object bindings “Moe,” “Larry,” and “Curly” into suggestions with the location bindings “[10, 10, 0],” “[20, 20, 0],” and “[30, 30, 0],” the locations of the three pigs. It also changes their suggested-module field from *Eat-object* to *Pickup-object*, and increments their hop-counters from one to two.

- *Walk-to-location* is marked “dirty.”
- *Pickup-object* is marked “clean.”

12. Were the wolf’s hands full, the suggestions would still be blocked by the order-filter, because the wolf is not yet at the location of any of the objects, and the order in which *Pickup-object*’s preconditions are satisfied matters.
Again the loop repeats, and this time Walk-to-location is dirty. However, in this simple example Walk-to-location has no predecessor-links. The module is therefore is automatically marked “clean.” Since no module is still dirty, the loop terminates and the module-passing phase is complete.

3.5.3 Example conflictor-link phase

6. Hunger creates a new blank suggestion, but because it has no conflictor-links the suggestion does not get passed.

7. Eat-object, Pickup-object, and Walk-to-location all have supportive suggestions in their suggestion-lists, and pass these suggestions down their conflictor-links. Walk-to-location has no conflictor-links, and therefore does nothing. Eat-object has two conflictor-links, one to Drop-object and one back to Eat-object. Because none of the three pigs are currently in-hand, all three suggestions are blocked by the condition-filter of both these links. Pickup-object passes the three suggestions in its suggestion-list down its single conflictor-link, which leads back to Pickup-object. The wolf’s hands are free, so this links conflictor-link passes each suggestion unhindered, and the counter-update-transform increments their hop-count from two to three. The precondition-transformation for this link changes the suggested-binding of each of the three suggestions from a pig into “blank.” The add-transformation then converts each of the three “blank” suggestions into four separate suggestions, one for each “interesting” object in the world. At this point, there are

---

13. Note that no module can ever pass a suggestion to itself without changing the suggestion’s suggested-binding field, because the suggestion will automatically be a duplicate with a longer hop-count, and will therefore be deleted. For example, no suggestion will ever make it down the conflictor link from Eat-object to itself, because the suggested-binding is not transformed. It is possible, however, for a suggestion supporting picking up one object in Pickup-object to lead to an inhibitory suggestion to not pick up any other objects.

14. The add filter also converts the suggested-module field from the from-module to the to-module, but in this case these are both the same module.
12 suggestions being fed into the conflictor-transformation: three supporting the picking up of “Moe,” three supporting “Larry,” three supporting “Curly,” and three supporting picking up the pogo-stick. The conflictor-transformation turns all 12 of these supportive suggestions into inhibitory suggestions, and the failed-module-filter and relevant-when-filter of this link allow these 12 inhibitory suggestions to pass unhindered. When the suggestions are received by Pickup-object, all nine suggestions inhibiting the picking up of Moe, Larry, and Curly are ignored, because the suggestions have identical suggested-modules and suggested-bindings as suggestions already on Pickup-object’s suggestion-list, and these inhibitory suggestions have a hop-count one higher than the hop-count of the old, non-inhibitory suggestions. Of the remaining three inhibitory suggestions that inhibit the picking up of the pogo-stick, the first is added to Pickup-object’s suggestion-list. The other two are ignored, because once the first is added, the remaining two are duplicates.

8. Using the equation: base energy = \( \begin{cases} 
G \times S^H, & \text{when suggestion is supportive} \\
-G \times C \times S^H, & \text{when suggestion is inhibitory}
\end{cases} \)

and the values \( G = 22 \), \( C = 0.90 \), and \( S = 0.99 \), the energies for each instantiated
behavior are calculated and are found to have the following values:

<table>
<thead>
<tr>
<th>Instantiated behavior</th>
<th>Supportive base energy</th>
<th>Inhibitory base energy</th>
<th>Total base energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat-object &lt;Moe&gt;</td>
<td>22</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>Eat-object &lt;Larry&gt;</td>
<td>22</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>Eat-object &lt;Curly&gt;</td>
<td>22</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>Pickup-object &lt;Moe&gt;</td>
<td>21.56</td>
<td>0</td>
<td>21.56</td>
</tr>
<tr>
<td>Pickup-object &lt;Larry&gt;</td>
<td>21.56</td>
<td>0</td>
<td>21.56</td>
</tr>
<tr>
<td>Pickup-object &lt;Curly&gt;</td>
<td>21.56</td>
<td>0</td>
<td>21.56</td>
</tr>
<tr>
<td>Pickup-object &lt;pogo-stick&gt;</td>
<td>0</td>
<td>-19.21</td>
<td>-19.21</td>
</tr>
<tr>
<td>Walk-to-location [10, 10, 0]</td>
<td>21.34</td>
<td>0</td>
<td>21.34</td>
</tr>
<tr>
<td>Walk-to-location [20, 20, 0]</td>
<td>21.34</td>
<td>0</td>
<td>21.34</td>
</tr>
<tr>
<td>Walk-to-location [30, 30, 0]</td>
<td>21.34</td>
<td>0</td>
<td>21.34</td>
</tr>
</tbody>
</table>

Table 6: Example energies of instantiated behaviors

9. Pickup-object <pogo-stick> has a negative total base energy, so it is declared to be inhibited. The binding “pogo-stick” is added to the inhibited-blacklist for Pickup-object, with a penalty time of one timestep.

3.5.4 Example resolution phase

10. The only instantiated behaviors that are executable are Walk-to-location [10, 10, 0], Walk-to-location [20, 20, 0], and Walk-to-location [30, 30, 0]. All other instantiated behaviors are removed from consideration.

11. The nearness energy for each executable behavior is calculated and added to the base energy. In the wolf’s current implementation the nearness energy for a location a given distance from the wolf is given by the following equation:

\[
\text{nearness-energy} = \begin{cases} 
\text{nearness-multiplier} \times \frac{100 - \text{distance}}{100}, & \text{distance} < 100 \\
0, & \text{otherwise}
\end{cases}
\]

With the wolf at [35, 35, 0], the nearness energy calculates to the following values

---

15. In general, the nearness equation is domain dependent. In this implementation, if a behavior represents an action that is always close to completion, such as Drop-object, the nearness energy is simply equal to the nearness-multiplier times 1.0.
for the three suggestions:

<table>
<thead>
<tr>
<th>Instantiated behavior</th>
<th>total base energy</th>
<th>nearness energy</th>
<th>total energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk-to-location &lt;[10, 10, 0]&gt;</td>
<td>21.34</td>
<td>6.46</td>
<td>27.80</td>
</tr>
<tr>
<td>Walk-to-location &lt;[20, 20, 0]&gt;</td>
<td>21.34</td>
<td>7.87</td>
<td>29.21</td>
</tr>
<tr>
<td>Walk-to-location &lt;[30, 30, 0]&gt;</td>
<td>21.34</td>
<td>8.58</td>
<td>29.92</td>
</tr>
</tbody>
</table>

Table 7: Example energies of executable instantiated behaviors

12. The total energy of all three instantiated behaviors is computed by adding each behavior’s total base energy to its nearness energy. These add up to be 27.80, 29.21, and 29.92, respectively.

13. The instantiated behavior with the highest total energy is chosen as the action to execute. In this case, the effector of Walk-to-location is called with the argument [30, 30, 0]. This causes the wolf to take a single step towards Curly, slowly but surely sealing his doom.

3.5.5 Example special cases phase

14. Of the two executable instantiated behaviors that did not get chosen for execution, neither was passed over only because of inhibition, so no behavior is added to an inhibited-blacklist in this step.

15. If the physics layer reports an error in executing the action Walk-to-location [30, 30, 0], the binding [30, 30, 0] will be added to Walk-to-location’s failure-list. Otherwise, it won’t.

16. There are no instantiated behaviors that are currently unsatisfiable, so no bindings are added to any module’s giveup-blacklist this timestep.

17. An action was chosen this timestep, so no corrective measures need to be taken.
3.6 Avoiding thrashing caused by failed action

ANA will occasionally get into loops if an action fires but fails to accomplish what is expected. This problem can cause a character to thrash, blindly performing the same failed action over and over again without any benefit. For example, a character try forever to pickup an object that is too heavy to lift. Maes argues that in a rapidly changing world these action loops will soon be broken because the environment will change [Maes, 1989], but loops become more of a problem in less dynamic environments. In the PHISH-Nets architecture, the failure-blacklist in each module prevents this simple form of thrashing.

A more subtle form of thrashing occurs when an action has several preconditions, one or more of which are unsolvable. For example, take the simple Big Bad Wolf specification, but assume Drop-object is unavailable (either because it failed previously, is being inhibited, or simply is not implemented). If the wolf is carrying a non-food item, he will first walk over to the food (i.e., a pig). Since his hands are full the wolf cannot pick up the pig, but will still religiously stay next to the pig to satisfy the other precondition for Pickup-object, even if staying near the pig means not satisfying less important but satisfiable goals. This situation is an example of a “dog in the manger” problem, where a module keeps less important modules from acting even though it cannot become active itself.

The PHISH-Nets architecture detects dog-in-the-manger problems by noting instantiated behaviors that are not executable but cannot create output from any of their predecessor-links. These behaviors are added to the giveup-blacklist for its associated module, and this blacklist is used the same as the failure-blacklists to prevent thrashing. Because a module cannot accept a binding on a giveup-blacklist, it is possible to have trickle-up effects where the failure of a single instantiated-behavior causes another behavior to be added to a giveup-blacklist, which in turn causes a third behavior to be added to a giveup-
blacklist. For example, in the above case where Drop-object is not available the pig would be added to Pickup-object's giveup-blacklist. On the following timestep, the pig would be added to Eat-object's giveup-blacklist, because giving up on picking up the pig made Eat-object's preconditions unsatisfiable.

There are a few degenerate cases where a behavior is impossible to execute (i.e., not all its preconditions can be satisfied), but where this condition is not detected and therefore the instantiated behavior is not put into the giveup-blacklist. For example, looping preconditions (e.g., where a character needs to fix the hole in the bucket to fetch the water needed to fix the hole in the bucket) are not detected in the above procedure. This particular form of the dog-in-the-manger problem is discussed in Section 5.5.1.

3.7 What to do when things still go wrong

Rather than trying to design the algorithm to handle all situations a character might ever encounter, the PHISH-Nets architecture is instead designed to handle as many common situations as is reasonably possible and to detect and remedy a breakdown should it occur. The simplest form of breakdown occurs when a creature has not come up with an action by the end of the action-selection cycle. One possible cause is that there is no executable action that helps a goal. In this case taking no action is appropriate. However, a second possible cause is that an impossible-to-satisfy (but not given up on) high-energy goal is inhibiting lower-energy but satisfiable goals. Yet another possibility is that a behavior could not be executed earlier and was added to a giveup-blacklist, but since that time the environment has changed such that the behavior can now become executable. If no action is taken, the following actions are taken to remedy the situation. First, the highest energy module/binding pair that is actively inhibiting another module/binding is placed in
the giveup-blacklist. If after this action the creature still can take no action, all giveup-blacklists are cleared.

The methods detailed above handle most common failure-modes, but the PHISH-Nets algorithm can still get into loops in certain circumstances. These situations, and how future extensions might handle them, are discussed in Section 5.5.6
Chapter 4: Experiments and Results

The algorithm has been implemented in Scheme [Abelson et al, 1985], a LISP-like language, and has been tested in a 3D graphical interactive story system based on The Three Little Pigs. In this story system an autonomous wolf inhabits a cartoon world similar to the Warner Brothers' Coyote/Road-Runner cartoons. The wolf tries to catch little pigs through a variety of methods, including climbing down chimneys, blowing down houses, or simply planting bombs in front of the pigs' houses. A human controller can "play God" to the wolf, controlling when bombs go off, whether houses blow down, and whether character actions succeed or fail. The controller can also move objects around or remove them completely from the world. In the test scenarios only the wolf is an autonomous character, and a single pig is controlled by hand. The physics of the world, although still controllable by a human player, does not yet have an easy-to-use interface. Future versions will have three autonomous pigs in addition to the wolf, and will provide a GUI interface for real-time control of the virtual world.

Locations in the 3D world are continuous (x,y,z) coordinates. Objects in the world consist of the wolf, a pig, a house that can be built or destroyed and through which the wolf cannot pass, a pogo-stick that can be used to jump to the roof of the house, and a chimney that can give the wolf access to the house. There is also a bundle of dynamite (TNT) and a detonator plunger to make it explode. In addition to the four behavior modules specified earlier the wolf has Blow-down-object, Pogo-to-location, Plant-tnt-at-location, and Explode-object behaviors. The specification for the behaviors of the more advanced wolf is shown in Appendix A.
The house can be made of three different materials, straw, twig, or brick. The straw house can be destroyed either by huffing-and-puffing-and-blowing-it-down, or by exploding it with the TNT. Blowing on the twig house will fail, but the TNT will still work. Both methods will fail to destroy the brick house. The wolf does not know what material the house is, and must find out by trial and error what methods work. The house is a large structure, and (location-of <house>) returns four values: the locations of the front door, the back wall, and the two side walls\(^1\). It is possible to walk inside the house from one location to another, and from anywhere inside the house to the outside, but the wolf cannot walk from a location outside the house to a location inside except by climbing down the chimney to the fireplace. The wolf can walk to any location on the ground so long as there is a path to it\(^2\), but he cannot walk to locations above ground-level. He can, however, pogo-stick to locations above the ground as long as he has the pogo-stick.

Each experiment described below is designed to test a different aspect or design goal of the algorithm. Specifically, the experiments test the proper handling of multiple goals, multiple ways of performing an action, multiple conflicting goals, opportunistic behavior, and graceful handling of the failure of an action.

### 4.1 Experiment 1: single goal and single method

#### 4.1.1 Setup

Goal: *Hunger-satisfied* = 22

Actions available to the wolf: All

Objects in world:

---

1. As explained in Section 3.2.1, transformations are one-to-many, and thus can produce multiple outputs for a single input. All four locations are passed as separate suggestions to the modules that use the location of the house.
2. i.e., so long as the location is not inside the house when he is outside
House is straw
Pig starts in the house
Wolf starts a short distance away from the house, nearest the front door.

Figure 11: Initial conditions for Experiment 1

4.1.2 Result

The wolf walked to the front door of the house, huffed-and-puffed-and-blew-it down, walked over to the pig, picked it up, and ate it.

4.1.3 Variations

Variation 18. If the wolf instead starts nearest to the left-hand wall of the house, he will walk to that wall to do his huffing and puffing.

Variation 19. If, while the wolf is walking towards the front-door, the pig suddenly runs out the back, the wolf will walk around the house rather than blow it down (i.e., adapting to the changing situation).
4.1.4 Discussion

The main experiment shows the standard handling of a plan-like sequence of actions to satisfy a goal. Variation (1) shows how the wolf prefers closer locations when given otherwise equal choices. Variation (2) shows how he can smoothly handle changes in the environment.

4.2 Experiment 2: Multiple methods / complimentary goals

4.2.1 Setup

Goal: \textit{Hunger-satisfied} = 22, \textit{Desire-to-destroy-things} = 10

Actions available to the wolf: All

Objects in world:

- House is straw
- Pig starts in the house
- Wolf starts a short distance away from the house
- TNT lies a short distance away from the house in a unique location
- Detonator lies a short distance away from the house in a unique location
- Pogo-stick lies a short distance away from the house in a unique location

4.2.2 Result

The wolf walked to the house, huffed-and-puffed-and-blew-it down, walked over to the pig, picked it up, and ate it.

4.2.3 Variations

Variation 1. When the wolf's goals were changed to \textit{Hunger-satisfied} = 22, \textit{Desire-to-be-in-high-places} = 10, instead of blowing down the house the wolf went to the pogo-stick, picked it up, bounced to the chimney of the house, climbed down
the chimney, dropped the pogo-stick, picked up the pig, ate the pig, picked up the pogo-stick, and bounced back up to the roof of the house.

4.2.4 Discussion

This experiment shows how multiple goals can contribute to determining the next action. In environments where there are several methods for accomplishing the same action, often it is possible to satisfy lower priority goals at the same time a primary goal is being satisfied. Note that Hunger-satisfied was deliberately made higher than the secondary goal to avoid introducing conflicting goals into the experiment. Conflicting goals are addressed in the next experiment.

4.3 Experiment 3: Multiple Inhibitory Goals

4.3.1 Setup

Goals: Hunger-satisfied = 30, Destruction-urge = 20, Carrying-pogo = 10

Actions available to the wolf: All except Blow-down-object

Objects in world:

- House is straw
- Pig starts in the house
- Wolf starts a short distance away from the house
- TNT lies a short distance away from the house in a unique location
- Detonator lies a short distance away from the house in a unique location
- Pogo-stick lies a short distance away from the house in a unique location

4.3.2 Result

The wolf walks to the pogo-stick, picks it up, and bounces to the TNT. He then drops the pogo-stick to pick up the TNT, brings the TNT to the house and places it there. He
walks to the detonator and explodes the house. He then walks to the pogo-stick, bounces
to the pig, drops the pogo-stick, eats the pig, and then picks up the pogo-stick again. The
same result occurs regardless of the distance between the pogo-stick and the wolf.

Figure 12: The wolf about to destroy the house

4.3.3 Variations

Variation 1. With the goals set to $\text{Destruction-urge} = 30$, $\text{Carrying-pogo} = 20$, $\text{Hunger-satisfied} = 10$, everything happens the same as before until the wolf gets to
the pig. At this point he simply stands on top of the pig and refuses to
release the pogo-stick, because it is a more important goal than satisfying his hunger.

Variation 2. With the goals $\text{Carrying-pogo} = 30$, $\text{Hunger} = 20$, $\text{Destruction-urge} = 10$,
the wolf gets the pogo-stick and bounces to the TNT. At this point he refuses
to drop the pogo-stick, and gives up on exploding the house. He bounces to
the chimney of the house and climbs down. Again he refuses to drop the
pogo-stick, and gives up on eating the pig. At this point no recommended
action is executable, so all the giveup blacklists are cleared and he bounces
to the TNT. The cycle repeats ad infinitum, with the wolf going between
TNT, chimney, and pig.

Variation 3. With all three goals set equal to 30 and the conflictor-multiplier parameter
set to 1.0, the wolf gets the pogo-stick, bounces to the TNT, drops the pogo-
stick, picks up the TNT, walks to the house and plants the TNT, and
explodes the house. He then walks back to the pogo-stick and bounces to the
pig. At this point he refuses to drop the pogo-stick, because this action
would undo his goal of carrying the pogo-stick. Having no other actions to
take, the wolf does nothing from this point on.

Variation 4. With all three goals set equal to 30 and the conflictor-multiplier parameter
set to its default of 0.9, the story begins the same as in variation 3, above, but
once the wolf gets to the pig he instead gets into a loop: he first drops the
pogo-stick, then picks it back up, then drops it again, ad infinitum.

4.3.4 Discussion

The results of this experiment and variations 1 and 2 happened as should be expected:
the wolf always tried to satisfy lower priority goals but would sacrifice them to satisfy a
higher priority goal. The last two variations show what happens when two goals of the
same priority conflict. Recall that the conflictor-multiplier parameter is a kind of gauge on
how much dithering should be allowed. A value of 1.0 means conflictor-links carry their
full energy, so an even slightly higher-energy suggestion will always win out. In
variation 3, Carrying-pogo has slightly more inhibitory power than Drop-object <pogo-
stick> has energy, because Drop-object is one hop further removed from its energy source,
Hunger-satisfied. In variation 4, the energy from the conflictor link is reduced just enough
to allow Drop-object <pogo-stick> to act, but then the wolf is in a situation where he mustchoose between picking up the pig and picking up the pogo-stick again. As in variation 3,
the pogo-stick wins because of the slight bias towards shorter action-paths. The specific failures in variations three and four are due to general problems the PHISH-Nets algorithm has with situations where the ordering of actions is especially important. This point will be discussed further in Section 5.5.1.

4.4 Experiment 4: Undoing one goal to accomplish another

4.4.1 Setup

Goals: *Destruction-Urge* = 30, *Hands-are-free* = 30

Actions available to the wolf: All except *Blow-down-object*

Objects in world:

- House is straw
- Pig starts in the house
- Wolf starts a short distance away from the house
- TNT lies directly in front of the house
- Detonator lies a short distance away from the house
- Pogo-stick lies a short distance away from the house in a unique location

Special: *Explode-object* is modified such that the wolf must not only be standing next to the detonator to explode an object but must actually be holding the detonator. The *conflictor-multiplier* is set to its default of 0.9.

4.4.2 Results

At the start of this scenario the wolf has already satisfied the goal of having nothing in its hands. It goes to the detonator and undoes this goal by picking up the detonator. It explodes the house, then drops the detonator.
4.4.3 Variations

Variation 1. Changing the conflictor-multiplier from its default to 1.0 causes the wolf to walk towards the detonator but he takes no action once he gets to it.

4.4.4 Discussion

In this experiment, Hands-are-free is satisfied at the start, but needs to be undone to achieve the wolf’s other goal. Hands-are-free inhibits Pickup-object, but because the conflictor-multiplier parameter is less than 1.0 the inhibition is slightly less than its total energy and the wolf picks up the detonator. At this point there are two unsatisfied goals, one that would be satisfied by dropping the detonator and one that would be satisfied by exploding the house. Explode-object <house> inhibits Drop-object <detonator>, but there is no inhibition in the reverse direction. This condition gives Explode-object <house> the clear advantage. After the detonation is complete, the wolf drops the detonator, thereby achieving both goals. Note that this experiment is different from the situation in Section 4.3, variations 3 and 4, where there was mutual inhibition instead of inhibition on only one side.

It should be noted that the environment in this experiment is equivalent to the A-on-B, B-on-C blocks-world problem described in [Sussman, 1975]. In this classical problem, a robot must satisfy the two goals of having block A stacked on top of block B and having block B stacked on block C. At the start of the experiment, A is already stacked on B and must be taken off if B is to be moved.

4.5 Experiment 5: Mutual inhibition

4.5.1 Setup

Goals: Carrying-pig = 22, Carrying-detonator = 22, Destruction-urge = 10
Chapter 4 Experiments and Results

Actions available to the wolf: All

Objects in world:

House is straw
Pig starts away from the house in the same location as the detonator
Wolf starts a short distance away from the house
detonator lies in the same location as the pig

4.5.2 Results

The wolf approaches the location where both the detonator and pig are. When he gets there, for a single time-step he starts to move towards the house to blow it down, but on the next timestep he picks up one of the two objects at random. He then walks to the house and blows it down.

4.5.3 Discussion

Both Carrying-pig and Carrying-detonator need use of the wolf’s hands and therefore inhibit each other. In such a situation it is important to insure that the two behaviors do not eliminate each other, thereby allowing a third, lower priority behavior to take effect. In the example above, as soon as the wolf gets within reach of both objects the behaviors Pickup-object <pig> and Pickup-object <detonator> inhibit each other. Due to the conflictor-multiplier parameter, each brings the other’s energy value to slightly above zero, and the uninhibited Destruction-urge can take over. However, if the inhibition is not taken into account then both Pickup-object <detonator> and Pickup-object <pig> have energies higher than the action that was finally chosen, so at the end of the timestep the action with the largest amount of inhibition is placed on the inhibited blacklist, as described in Section 3.4. In this case, both have equal amounts of inhibition, so one of them is placed on the list. In the next timestep, the action that was placed on the inhibition blacklist can-
not pass inhibition, so the other action remains uninhibited and becomes active. Thus, after one timestep of performing a less relevant action, the wolf will either pick up the pig or pick up the detonator, and then go to satisfy his lesser goal.

4.6  Experiment 6: Opportunism

4.6.1  Setup

Goals: \textit{Destruction-urge} = 22

Actions available to the wolf: All

Objects in world:

- House is straw
- Wolf starts a large distance away from the house
- TNT lies between the wolf and the house, but slightly to one side
- Detonator lies a short distance away from the house in a unique location
- Pogo-stick lies a short distance away from the house in a unique location

4.6.2  Results

The wolf begins by walking towards the house. As he nears the TNT, he modifies his course, picks up the TNT, and then continues towards the house. Once at the house, he huff-and-puffs and blows the house down.

4.6.3  Variations

Variation 1. Same as above except the TNT is placed some distance further away from the wolf’s path to the house. In this case, the wolf is not diverted and proceeded to blow down the house, ignoring the TNT.
Chapter 4  Experiments and Results

4.6.4 Discussion

In an unpredictable world it is often wise to be opportunistic, favoring actions that are immediately available even if they satisfy less important goals. The extra boost of energy given to modules especially close to completion give characters a degree of opportunism. In this experiment, grabbing the TNT was less important than going to the house, because exploding the house takes more steps than blowing it down, but the wolf still picked up the TNT “just in case” if it was close.

4.7 Experiment 7: Failure of actions

4.7.1 Setup

Goals: Hunger-satisfied = 22, Destruction-urge = 22

Actions available to the wolf: All

Objects in world:
- House is brick
- Pig starts in the house
- Wolf starts a short distance away from the house
- TNT lies a short distance away from the house in a unique location
- Detonator starts a short distance away from the house in a unique location
- Pogo-stick lies a short distance away from the house in a unique location

4.7.2 Results

The wolf goes to the house and huffs-and-puffs to try and blow it down. Because the house is made of brick the action fails, so he goes to the TNT, picks it up, places it by the house, goes to the detonator and tries to explode the house. This also fails. He then goes
to the pogo-stick, bounces to the roof of the house, climbs down chimney, drops the pogo-stick, picks up the pig, and eats it.

Figure 13: The wolf climbing down the brick house chimney

4.7.3 Discussion

This experiment shows how PHISH-Nets handle failure of actions robustly by using multiple methods to achieve a goal. When blowing the house down failed, the binding <house> was added to the failure blacklist for Blow-down-object and not tried again. The wolf then tried to explode the house, but this action also failed and the binding<house> was added to the failure blacklist for Explode-object. With no other methods available to destroy the house, the wolf used the pogo-stick to catch the pig, thereby satisfying his remaining goal. Note that not only did the wolf not pathologically continue to try blowing the house down, but he also went on to try other methods of simultaneously satisfying both goals before finally giving up and satisfying only his hunger.
4.8 Experiment 8: Giving up on the impossible

4.8.1 Setup

Goals: Destruction-urge = 30, Carrying-pig = 10

Actions available to the wolf: All except Blow-down-object

Objects in world:
- House is straw
- Pig starts a short distance away from the house
- Wolf starts a short distance away from the house in a unique location
- TNT lies a short distance away from the house in a unique location

Special: After the wolf places the TNT by the house, but before he grabs the pig, the TNT is moved away from the house.

4.8.2 Results

The wolf starts by going to get the TNT and placing it next to the house. He then decides that there is no way to explode the house because there is no detonator, and proceeds to satisfy his secondary goal of carrying the pig. While he is walking to the pig the TNT is moved away from the house, but the wolf has already given up on that action so he does not pathologically move the TNT back to its starting location. Instead, he goes and picks up the pig.3

4.8.3 Variations

Variation 1. Add the detonator to the world, but place it in the house. This time the wolf drops the TNT, but does not add exploding the house to his giveup blacklist. However, because there is no pogo-stick in this variation, he cannot get to

---

3. At this point, there are no actions selected so the wolf actually clears his giveup-buffer, carries the pig over to the TNT, drops the pig, and takes the TNT to the house, continuing in this loop ad infinitum.
the roof to climb down the chimney to get to the detonator. He therefore
goes to satisfy his secondary goal to pick up the pig. Because exploding the
house was not added to the wolf's giveup-blacklist, when the TNT is moved
away from the house, he goes to replace it before going to pick up the pig. If
a pogo-stick is added to the world, the wolf then uses the pogo-stick to get to
get to the chimney, climbs down, and explodes the house before going back
to the pig.

Variation 2. This variation is radically different from the standard version of the
experiment. In this variation, the wolf has three goals: \textit{Carrying-pig} = 30,
\textit{Carrying-detoner} = 20, and \textit{Destruction-urge} = 10. The pig, wolf, and
detonator all start some distance away from the straw house in unique
locations. The wolf has all abilities, including \textit{Blow-down-object}. The wolf
first walks to the pig and picks it up. He then walks to the detonator, but
when he gets there he cannot drop the pig because \textit{Carrying-pig} inhibits
\textit{Drop-object <pig>}. The wolf then gives up on picking up the detonator, and
walks to the house to blow it down. Even if the detonator is moved away
from the wolf, he will not pathologically walk to the detonator, but will
instead service his lesser achievable goals. After blowing down the house,
no action is selected, so all the giveup blacklists are cleared. The wolf then
goes back to the detonator and stands over it, but still refuses to drop the pig.

4.8.4 Discussion

This experiment shows how PHISH-Nets give up on behaviors with preconditions that
cannot be satisfied. In the first example, the wolf gives up on going to the location of the
detonator, and in turn gives up on exploding the house. Variation number two is a similar
case, except that the wolf gives up on holding the detonator not because the action is
unavailable, but because the action is being inhibited such that it cannot be executed.
Chapter 4

Experiments and Results

Variation number one shows a case where the wolf does not realize that getting the detonator is impossible. Since the wolf could satisfy the preconditions for *Explode-object* `<house>` by walking to the detonator, he does not add the binding to his giveup blacklist. With only local knowledge there is no good way to detect that to get to the detonator he must explode the house, and to explode the house he has to get to the detonator. This problem is discussed further in Section 5.5.1.
Chapter 5: Discussion

5.1 Producing Personalities using PHISH-Nets

As stated in the introduction, one of the goals of this research is to produce an algorithm that can create characters with personality. A complete character needs more than an action selection algorithm; it also needs at least a complex model of emotions, a model of communications and social rules, and at sophisticated animation systems to show the subtleties of mood, focus of attention, and thought processes. However, at the action-selection level much can still be done to endow characters with unique and rich personalities.

In PHISH-Nets, many personality traits can be represented by creating goals that a character generally wants to accomplish. For example, a lazy character might have a goal to minimize the amount of work he has to do. This would bias all his actions towards low-effort. An energetic character, on the other hand, might instead have the goal to be reliable, which would bias his actions towards those with the least chance of failure. Although character traits are usually persistent, goals could selectively change over time, giving the illusion of learning. For example, if a character is burned his fear of fire might increase, biasing him to avoid getting too near fires in the future.

Another way to control a character’s personality is through the global parameters of the algorithm. For example, a hedonistic character might have a high valued nearness-parameter. This would cause him to “live for the moment,” and take advantage of situations as they arise rather than plan ahead. A fastidious character, on the other hand, would have a
Chapter 5

Discussion

high confractor-multiplier, causing him to hold on to what he has rather than take a chance by letting it go.

5.2 Extensibility

One of the primary advantages of the PHISH-Nets architecture is that creatures can be extended by creating new behaviors, without the need to specify in what way the new behaviors must interact with the old. For example, adding Pogo-to-location only required adding the one new behavior module, plus the graphics for the action. The author of the character does not have to specify that in order to grab the new pogo-stick object the character must use the module Pickup-object, nor that getting within reach of the pogo-stick would require Walk-to-location. Furthermore, any old or new actions will automatically use Pogo-to-location as an alternative to walking.

Such extensibility comes about because the compiler automatically links all possible causal links. Adding a new module requires a recompile but no more. Extensibility also comes because variable bindings are always encoded in a common base language. For example, to get to the location of the pogo-stick, Pickup-object <pogo-stick> passes the (x,y,z) coordinates of the location of the pogo-stick to Walk-to-location. Walk-to-location need not know about pogo-sticks at all, so long as it knows about (x,y,z) locations.

5.3 Speed of the algorithm

The speed of the algorithm depends a great deal on the kind of environment a character is in, but the order of the speed the algorithm can be determined with the help of a few assumptions. First, assume that as a network grows the number of goals, modules, and conditionals (predicates computed from sensor-values) in the network also grow but that the length of modules’ precondition-lists, add-lists, and delete-lists remain constant.
Assume also that the number of precondition-lists, add-lists, and delete-lists in which a
given conditional appears remains roughly constant; that is, when modules are added they
tend to have new effects rather than duplicate already existing modules. Together, these
two assumptions let us state that the number of links in a network grows in proportion to
the number of behavior modules. Assuming that the variables in Table 6 remain constant
as the network grows and given the variables shown in Table 5, one can easily show that
the speed of each step of the algorithm is shown in Table 8. It follows that the overall speed
of the algorithm is \( O(S \times M \times B) \), where \( S \) is the length of the longest sequence of actions
possible, \( M \) is the number of behavior modules in the network, and \( B \) is the maximum num-
ber of bindings (usually objects or locations) in the world. The value of \( B \) increases with
the size of the world, whereas \( M \) and \( S \) increase with the complexity of the character.

Although it is dangerous to make claims of parallelizability without going through the
rigors of implementation, this algorithm certainly lends itself to parallelization. Each link
in the network acts independently, and could be implemented on a separate processor.
Such an implementation would reduce the speed of the algorithm to \( O(S \times B) \). However,
as is noted in [Johnson, 1991], to create a parallel version of this algorithm one would need
a method by which each link could access the sensor-data and state of the world. Design-
ing a parallel implementation of the PHISH-Nets architecture is outside the scope of this
thesis.

<table>
<thead>
<tr>
<th>C</th>
<th>number of conditionals (predicates)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>number of behavior modules</td>
</tr>
<tr>
<td>G</td>
<td>number of goals (max = C)</td>
</tr>
<tr>
<td>B</td>
<td>max number of bindings (objects, locations, etc.)</td>
</tr>
<tr>
<td>S</td>
<td>length of the longest sequence of actions possible</td>
</tr>
</tbody>
</table>

Table 8: Variables

81
Table 9: Assumed constants

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>maximum number of preconditions for a module</td>
</tr>
<tr>
<td>A</td>
<td>maximum number of add-list entries for a module</td>
</tr>
<tr>
<td>D</td>
<td>maximum number of delete-list entries for a module</td>
</tr>
<tr>
<td>Cp</td>
<td>maximum number of preconditions a conditional appears in</td>
</tr>
<tr>
<td>Ca</td>
<td>maximum number of add-lists a conditional appears in</td>
</tr>
<tr>
<td>Cd</td>
<td>maximum number of delete-lists a conditional appears in</td>
</tr>
</tbody>
</table>

Table 10: Intermediary values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_{gm} = Ca \times G )</td>
<td>links from goals to modules</td>
</tr>
<tr>
<td>( L_{mm} = Cp \times Ca \times M )</td>
<td>inter-module links</td>
</tr>
<tr>
<td>( L_{mc} = Cp \times Cd \times M )</td>
<td>inter-module conflictor links</td>
</tr>
<tr>
<td>Phase</td>
<td>Complexity</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Initialization phase</td>
<td>$O(M)$</td>
</tr>
<tr>
<td>Spreading from goals to modules</td>
<td>$O(Lgm \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(Ca \times G \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(G \times B)$</td>
</tr>
<tr>
<td>Spreading between modules</td>
<td>$O(Lmm \times S \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(Cp \times Ca \times M \times S \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(M \times B)$</td>
</tr>
<tr>
<td>Conflictor-link phase</td>
<td>$O(Lmc \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(Cp \times Cd \times M \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(M \times B)$</td>
</tr>
<tr>
<td>Marking for inhibition</td>
<td>$O(M \times B)$</td>
</tr>
<tr>
<td>Giving extra state boost</td>
<td>$O(M \times B)$</td>
</tr>
<tr>
<td>Resolution phase</td>
<td>$O(M \times B)$</td>
</tr>
<tr>
<td>Special cases phase</td>
<td>$O(Lmm \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(Cp \times Ca \times M \times B)$</td>
</tr>
<tr>
<td></td>
<td>$= O(M \times B)$</td>
</tr>
</tbody>
</table>

Table 11: Order of speed for each part of the algorithm

5.4 Comparison with the Agent Network Architecture

A few words should be said about the differences between PHISH-Nets and its progenitor, Maes' Agent Network Architecture. First, the PHISH-Nets algorithm addresses several of the criticisms voiced by Tyrrell that were mentioned earlier. Specifically, because it tags energy sources it does not run into problems dividing up energy. The problem of dividing energy becomes especially significant when goals can be solved in multiple ways. The PHISH-Nets architecture also has no equivalent of ANA's successor links, and therefore does not fall into the problem of attending to appetitive behaviors before consumptive
ones. Another advantage over ANA is that there are very few parameters that require tweaking, and the PHISH-Nets architecture is thus easier to evaluate. In general, the parameters that do exist can be kept at the default values except for special cases. The architecture can handle much more complex sequences of actions than can Agent Networks. Finally, because behavior modules can take variables PHISH-Nets characters can easily be extended to handle new environments with existing behaviors.

The architecture also maintains many of the advantages of ANA. Because of their distributed architecture, redundancy of methods, and opportunistic behavior both systems are very robust in uncertain conditions. They can also handle multiple goals gracefully. Finally, it is easy to add new behaviors to an agent and have them integrate smoothly with old ones.

5.5 Limitations and future work

5.5.1 Lack of global knowledge

Like all behavior-based systems, the PHISH-Nets algorithm is based on the idea of producing globally reasonable actions through the local interactions of behaviors [Maes, 1993b]. These interactions should be based solely on local knowledge, without reference to a global model of the world. However, in some environments global knowledge is necessary to produce “the right” action. For example, if the wolf needs to decide whether to undo one goal in order to achieve another, it should reasonably take into account how long the goal will remain undone and whether after undoing it there will be an opportunity to redo it later. A PHISH-Nets behavior module does not have access to such information, and this limitation can cause problems such as those seen in variations 3 and 4 of experiment 3, where the wolf refuses to drop the pogo-stick to eat the pig, even though he will be
able to pick up the pogo-stick soon after. The situation is analogous to a local-maximum in a hill-climbing system.

A related problem occurs when a goal has two preconditions and the satisfaction of one of these preconditions will, somewhere down the line, undo the other. For example, pack-tnt-at-location has two preconditions: that the wolf be carrying the TNT and that he be at the location where it should be packed. It turns out that all possible methods to satisfy carrying-object <tnt> will undo the precondition im-at-location <location-arg>. This global knowledge, as well as the knowledge of how many methods might satisfy either precondition, is unavailable to Pack-tnt-at-location. In simple cases, this type of problem can be avoided by hand-coding an ordering using the “preconditions-ordered?” field of the behavior module. However, one can imagine situations where the order in which two preconditions should be executed depends on the environment. The current version of PHISH-Nets cannot handle such situations, though it might be possible to add a system similar to the giveup-blacklist that finds an appropriate ordering through trial-and-error.

A special case of the latter problem occurs when a module undoes its own precondition as a side-effect of taking action. For example, Eat-object requires that the object be in hand, and one of the effects of this action is that the object is no longer in-hand. However, if the wolf were currently carrying the TNT and wanted to have his hands free, it would make no sense to pass energy to Eat-object <pig> because there is no way for him to accomplish the feat without first dropping the TNT first. In this particular implementation of the wolf the problem is solved by the transformation on the Hands-free element of the add-list, which transforms “blank” suggestions into the object currently being carried. This transformation limits the add-list to take effect only when the current object being carried is edible.
5.5.2 Limitations on types of actions

The most obvious limitation of the high-level behavior-specification language is that modules can only take one variable argument. For example, \textit{plant-tnt-at-location} takes a location as an argument, but has to implicitly specify “TNT” as the object to place. It would be nice to be able to specify a module such as \textit{plant-object-at-location <object-arg> <location-arg>}. However, such a syntax would greatly complicate the passing of arguments between modules because the order in which arguments are passed would have to be determined.

Another limitation is that the effects of an action must be context independent. For example, it is not possible to write the action \textit{Pickup-object} such that picking up a heavy object makes the wolf tired whereas picking up a light object does not. The only context that can be encoded is in the relevant-when clause, which only states under what conditions the action will perform as expected. It would be possible to extend the algorithm to allow a relevant-when clause for each add-list and delete-list entry, or with the existing system one can create multiple actions such as \textit{Pickup-light-object} and \textit{Pickup-heavy-object}, each with its own add-lists.

Finally, the “preconditions-ordered?” field only allows an author to specify whether precondition order matters or not. It does not allow more complex situations, such as having the first two preconditions unordered while the rest ordered. Currently, to achieve this kind of ordering a behavior must be broken into two different behaviors, one with each ordering. Finer control of the ordering of preconditions would be useful, for example, in the \textit{Pogo-to-location} module. Currently \textit{Walk-to-location} specifies that both a free path exist to a desired location and that there not exist a blocker to that location. These two preconditions are different ways of phrasing the same condition — one can never be true without the other. However, \textit{Blow-down-object} satisfies eliminating a blocker, while \textit{Climb-}
down-chimney satisfies being in a location where there is a free path to an object. Both conditions should be included so both kinds of solutions are available. Pogo-to-location only has the precondition that a blocker to the destination location does not exist, but does not have the precondition that there be a free path. This is because Pogo-to-location has ordered preconditions, and there is no way to specify that either action works equally well to accomplish the end goal being able to get to a location without running into a wall. Whichever precondition was listed first would always be the one the agent tried to satisfy, and the second one listed would always be spontaneously satisfied whenever the first was satisfied. Allowing more complex kinds of orderings would allow an author to supply both preconditions to Pogo-to-location. Complex orderings would also be an easy feature to add, since the underlying architecture of the compiled PHISH-Nets can already handle such cases.

5.5.3 Deadlines and other hidden constraints on actions

In general, PHISH-Nets do not take time-related issues into account. For example, it is impossible to encode a goal that something be done “before noon” or “at 5:30pm.” It is also not currently possible to encode how long an action might take to execute, except indirectly through the bias towards shorter action sequences and the extra energy boost for an action near completion. Similarly, the cost and reliability of an action can only be coded indirectly through goals that give energy to behaviors that use little energy, or that are known to be reliable.

5.5.4 Performing multiple actions simultaneously

While the PHISH-Nets algorithm takes multiple goals into account when selecting an action to take, it does not currently support taking multiple actions simultaneously. For example, a character cannot walk and chew gum at the same time. A first approximation
Chapter 5

Discussion

of this feature would be easy to implement — a character could simply perform all actions that were not mutually exclusive, giving priority to those actions with the highest energy. However, as is pointed out in both [Loyall, 1993] and [Blumberg, 1995], it is also important to blend actions together. For example, a character walks differently when juggling than when walking alone. Furthermore, emotional states should affect how a character acts, for example by changing a character's walking gait from a slouching shuffle to a bound depending on their mood. Such extensions to the behavior algorithm will need to be integrated with the lower-level motor system. Blumberg's Hamsterdam motor system (the motor system currently used by the PHISH-Nets implementation) already supports many kinds of action-blending, so such extensions should be able to leverage off this existing work.

5.5.5 Detecting when to give up only at the last minute

One of the design decisions was to give up on an action only when all other preconditions have been satisfied. For example, in experiment number 8 where the TNT was available but the detonator was not, the wolf first placed the TNT next to the house and only then gave up on exploding the house. The underlying principle is that the world may change while other preconditions are being satisfied, making it possible to satisfy everything. More importantly, the act of satisfying one of the preconditions may make another precondition doable. However, as is seen in the above example, this heuristic can sometimes make a character look stupid if in fact the action remains impossible to accomplish: it looks as if the character only noticed that there was no detonator at the last minute. One way around this problem may be to allow the author of a character to determine whether to give up immediately or to wait until all other preconditions are satisfied, just as an author can currently mark whether the order of preconditions matter. Characters that give up
immediately would seem more foresighted, while characters that only give up when there is no hope left would seem more foolish, or at least more optimistic.

5.5.6 Detecting failure and breakdowns

Although the PHISH-Nets algorithm tries to handle many situations, it cannot handle every contingency. Beyond a certain point adding more special cases to an architecture to handle bizarre situations correctly makes the architecture more brittle rather than less. The future focus of this project will be the detection and handling of failures and breakdowns, rather than always trying to get things right the first time.

Currently, failure detection is done on an ad-hoc basis: the physics layer specifically tells the character when an action has failed. It would be desirable for a character to determine for itself whether a failure has occurred. Furthermore, a character should be able to detect subtle kinds of breakdowns, such as when it is caught in an action loop or when an action is performing as expected but is not getting the character any closer to its goal. It would also be useful to have more information about a failure or breakdown in reasoning than simply "it didn’t work."

Failures and breakdowns can be detected in at least the following ways:

- **Time-out**: Stop doing an action if it has been performed too many times in the recent past. This procedure at least guarantees that the character will try new things rather than get caught in a tight action-loop. It is the simplest of failure-handling heuristics, and therefore good when no other method can work.

- **Looping**: Stop doing an action if it always takes you back to the same state of the world. This procedure provides a better way to prevent action-loops, but it requires that the character be able to distinguish between different states of the world.
Known effector failure: Stop doing an action if the low-level physics layer of your character triggers an error while performing it.

Unknown effector failure: An unknown effector failure occurs when the physics layer does not report an error, but an action still does not perform as expected. The action might not have any effect at all, or it could be that no forward progress is being made. An example of the latter would be when the character walks towards a location, but does not getting any closer even though his legs are moving.

Undesirable, unexpected side-effects: An action might yield the expected result, but at the same time produce unexpected side effects in addition to the expected behavior. One problem with detecting side-effects is that it is often difficult to determine which recent action caused an effect, if indeed any action was the cause.

Preconditions get removed out from under an action: An action may simply never be able to make all its preconditions true at once. For example, if the TNT always rolls away from the house before the wolf can get to the detonator, perhaps he should give up on exploding the house at all, because it relies on too many uncertainties.

The scopes of a failure might also be determinable with enough domain knowledge. For example, the failure could be a known temporary problem (e.g., the character is in free-fall, so no action can work at this point in time), or a more permanent problem, such as his leg just fell off. It could also be a problem only with a particular binding (it is impossible to get to a certain location because the ground is too uneven), or with the entire behavior module (it is impossible for the character to pick anything up because his hands are too slippery).

Given any of these failure-modes, there are several different kinds of repairs a character might perform. If the failure is temporary, he might simply try again immediately, whereas
with a long-term failure he might wait longer, as he does in the current system. If enough
domain knowledge is present, the character may be able to set a condition that must be met
before he will try again. For example, the wolf may wait until he feels stronger before he
tries to blow down the twig house again. Finally, there may be a remedy that can be made
to the network itself. For example, if a new side effect or precondition is found for a
behavior module, that effect can be added and that segment of the network recompiled.

5.5.7 Limitations of the current implementation

The current implementation of the PHISH-Nets algorithm was designed for ease-of-
modification and speed of implementation, rather than for runtime speed. It is therefore far
slower than one would want in an actual interactive story system. Furthermore, the world
currently must be modified with scripts rather than with a GUI interface. Both of these
problems should be fixed in the next version.
Chapter 6: Conclusion

The PHISH-Nets\textsuperscript{1} architecture is an action selection architecture designed to create virtual characters for interactive story systems. These characters can hold multiple, possibly conflicting goals, and secondary goals can affect the ways in which primary goals are accomplished. Using secondary goals and appropriate setting of architecture parameters, an author can give a character unique personality traits. Characters can operate in a continuous, dynamic and unpredictable world, can perform complex sequences of actions to achieve their goals, and are easy to create and extend.

The issues involved in creating characters for interactive story systems have been discussed, and comparisons have been made between this architecture and other agent architectures designed for interactive story systems. The PHISH-Nets architecture has been described and several examples of its operation have been given. An interactive story system has been created to test and showcase the architecture. Experiments have been performed to demonstrate how different design goals of the architecture have been satisfied, and to show remaining limitations of the algorithm. Specific aspects of the architecture have been discussed, including the creation of personalities, the extensibility and modifiability of character specifications, the speed of the algorithm, and finally limitations of the algorithm and future directions for this research.

\textsuperscript{1} Planning Heuristically In Situated Hybrid Networks
Appendix A  The full specification for the wolf

<table>
<thead>
<tr>
<th>Name</th>
<th>Eat-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Eat-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>hunger-satisfied (object-from-blank &lt;object-arg&gt;) hands-free (carried-object-from-blank &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Delete-list</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Relevant-when</td>
<td>object-edible &lt;object-arg&gt;</td>
</tr>
</tbody>
</table>

Table 12: The behavior module for *Eat-object*

<table>
<thead>
<tr>
<th>Name</th>
<th>Pickup-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Pickup-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Im-at-location (location-of &lt;object-arg&gt;) hands-free (blank-from-object &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>Yes</td>
</tr>
<tr>
<td>Add-list</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td>hands-free (interesting-objects-from-blank &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Relevant-when</td>
<td></td>
</tr>
</tbody>
</table>

Table 13: The behavior module for *Pickup-object*
Appendix A

The full specification for the wolf

<table>
<thead>
<tr>
<th>Name</th>
<th>Explode-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Explode-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>TNT-at-location (location-of &lt;object-arg&gt;) Im-at-location (detonator-location-from-object &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>Yes</td>
</tr>
<tr>
<td>Add-list</td>
<td>object-doesnt-exist &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td></td>
</tr>
<tr>
<td>Relevant-when</td>
<td>object-destroyable &lt;object-arg&gt;</td>
</tr>
</tbody>
</table>

Table 14: The behavior module for Explode-object

<table>
<thead>
<tr>
<th>Name</th>
<th>Walk-to-location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Walk-to-location]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>location</td>
</tr>
<tr>
<td>Preconditions</td>
<td>object-doesnt-exist (object-blocking-path-to &lt;location-arg&gt;) Im-at-location (locations-with-a-free-path-to &lt;location-arg&gt;) Im-on-ground (blank-from-location &lt;location-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>Im-at-location &lt;location-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td></td>
</tr>
<tr>
<td>Relevant-when</td>
<td>location-on-ground &lt;location-arg&gt;</td>
</tr>
</tbody>
</table>

Table 15: The behavior module for Walk-to-location

<table>
<thead>
<tr>
<th>Name</th>
<th>Pogo-to-location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Pogo-to-location]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>location</td>
</tr>
<tr>
<td>Preconditions</td>
<td>object-doesnt-exist (object-blocking-path-to &lt;location-arg&gt;) carrying-object (pogo-stick-from-location &lt;location-arg&gt;) Im-on-ground-or-pogoing (blank-from-location &lt;location-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>Yes</td>
</tr>
<tr>
<td>Add-list</td>
<td>Im-at-location &lt;location-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td></td>
</tr>
<tr>
<td>Relevant-when</td>
<td>location-on-or-above-ground &lt;location-arg&gt;</td>
</tr>
</tbody>
</table>

Table 16: The behavior module for Pogo-to-location
### Table 17: The behavior module for *Climb-down-chimney-to-location*

<table>
<thead>
<tr>
<th>Name</th>
<th>Climb-down-chimney-to-location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Climb-down-chimney-to-location]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>location</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Im-at-location (location-of-chimney-leading-to &lt;location-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>Im-at-location &lt;location-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td></td>
</tr>
<tr>
<td>Relevant-when</td>
<td>location-at-bottom-of-a-chimney &lt;location-arg&gt;</td>
</tr>
</tbody>
</table>

### Table 18: The behavior module for *Drop-object*

<table>
<thead>
<tr>
<th>Name</th>
<th>Drop-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Drop-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>hands-free (carried-object-from-blank &lt;blank-arg&gt;)</td>
</tr>
<tr>
<td>Delete-list</td>
<td>carrying-object &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Relevant-when</td>
<td></td>
</tr>
</tbody>
</table>

### Table 19: The behavior module for *Blow-down-object*

<table>
<thead>
<tr>
<th>Name</th>
<th>Blow-down-object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>[implementation for Blow-down-object]</td>
</tr>
<tr>
<td>Argument Type</td>
<td>object</td>
</tr>
<tr>
<td>Preconditions</td>
<td>Im-on-ground (blank-from-object &lt;object-arg&gt;)</td>
</tr>
<tr>
<td>Preconditions-ordered?</td>
<td>No</td>
</tr>
<tr>
<td>Add-list</td>
<td>object-doesnt-exist &lt;object-arg&gt;</td>
</tr>
<tr>
<td>Delete-list</td>
<td></td>
</tr>
<tr>
<td>Relevant-when</td>
<td>object-is-destroyable &lt;object-arg&gt;</td>
</tr>
</tbody>
</table>
## Appendix A

The full specification for the wolf

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pack-TNT-at-location</td>
<td>[implementation for Pack-TNT-at-location]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>carrying-object (TNT-from-location &lt;location-arg&gt;)</td>
<td></td>
</tr>
<tr>
<td>Im-at-location &lt;location-arg&gt;</td>
<td></td>
</tr>
</tbody>
</table>

| Preconditions-ordered?    | Yes                                               |

| Add-list                  | TNT-at-location                                   |
| Delete-list               |                                                  |
| Relevant-when             |                                                  |

Table 20: The behavior module for Pack-TNT-at-location

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explode-object</td>
<td>[implementation for Explode-object]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TNT-at-location (location-of &lt;object-arg&gt;)</td>
<td></td>
</tr>
<tr>
<td>Im-at-location (detonator-location-from-object &lt;object-arg&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

| Preconditions-ordered?    | Yes                                               |

| Add-list                  | object-doesn't-exist <object-arg>                |
| Delete-list               |                                                  |
| Relevant-when             | object-destroyable <object-arg>                 |

Table 21: The behavior module for Explode-object
Appendix A

The full specification for the wolf
## Appendix A

### The full specification for the wolf

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Filter Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eat Object</td>
<td>Pickup Object</td>
<td>when object not in hand&lt;br&gt;Object ⇒ Object</td>
</tr>
<tr>
<td>Eat Object</td>
<td>Eat Object</td>
<td>when object is in hand&lt;br&gt;Object ⇒ Object&lt;br&gt;conflictor-filter&lt;br&gt;when object is edible</td>
</tr>
<tr>
<td>Eat Object</td>
<td>Drop Object</td>
<td>when object is in hand&lt;br&gt;Object ⇒ Object&lt;br&gt;conflictor filter</td>
</tr>
<tr>
<td>Drop Object</td>
<td>Pickup Object</td>
<td>when object is in hand&lt;br&gt;Object ⇒ Object</td>
</tr>
<tr>
<td>Drop Object</td>
<td>Eat Object</td>
<td>when object is in hand&lt;br&gt;Object ⇒ Object&lt;br&gt;conflictor-filter&lt;br&gt;when object is edible</td>
</tr>
<tr>
<td>Drop Object</td>
<td>Drop Object</td>
<td>when object is in hand&lt;br&gt;Object ⇒ Object&lt;br&gt;conflictor-filter</td>
</tr>
<tr>
<td>Pickup Object</td>
<td>Pogo to Location</td>
<td>when not at location of object&lt;br&gt;Object ⇒ Location of object&lt;br&gt;when location is on or above ground</td>
</tr>
<tr>
<td>Pickup Object</td>
<td>Walk to Location</td>
<td>when not at location of object&lt;br&gt;Object ⇒ Location of object&lt;br&gt;when location is on ground</td>
</tr>
<tr>
<td>Pickup Object</td>
<td>Climb Chimney to Location</td>
<td>when not at location of object&lt;br&gt;Object ⇒ Location of object&lt;br&gt;when location is at base of chimney</td>
</tr>
<tr>
<td>Pickup Object</td>
<td>Eat Object</td>
<td>when hands are full&lt;br&gt;when at location of object&lt;br&gt;Object ⇒ Blank&lt;br&gt;Object ⇒ Carried-object&lt;br&gt;when object is edible</td>
</tr>
<tr>
<td>Pickup Object</td>
<td>Drop Object</td>
<td>when hands are full&lt;br&gt;when at location of object&lt;br&gt;Object ⇒ Blank&lt;br&gt;Object ⇒ Carried-object</td>
</tr>
<tr>
<td>Pickup Object</td>
<td>Pickup Object</td>
<td>when hands are free&lt;br&gt;Object ⇒ Blank&lt;br&gt;Object ⇒ Interesting-objects&lt;br&gt;conflictor filter</td>
</tr>
<tr>
<td>Pogo to Location</td>
<td>Blow Down Object</td>
<td>when there is a blocker to location&lt;br&gt;Location ⇒ Object-blocking-path&lt;br&gt;when object is destroyable</td>
</tr>
</tbody>
</table>

**Table 22:** Links of the full wolf, with prominent filters and transformations
### Appendix A

#### The full specification for the wolf

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Filter Series</th>
</tr>
</thead>
</table>
| Pogo to Location      | Explode Object          | when there is a blocker to location
|                       |                         | Location ⇒ Object-blocking-path when object is destroyable                  |
| Pogo to Location      | Pickup Object           | when no blocker to location
|                       |                         | when not carrying pogo-stick Location ⇒ pogo-stick                          |
| Pogo to Location      | Drop Object             | when no blocker to location
|                       |                         | when carrying pogo-stick Location ⇒ pogo-stick conflictor-filter           |
| Pogo to Location      | Eat Object              | when no blocker to location
|                       |                         | when carrying pogo-stick Location ⇒ pogo-stick conflictor-filter
|                       |                         | when pogo-stick is edible                                                   |
| Walk to Location      | Walk to Location        | when no free path to Location
|                       |                         | Location ⇒ Location-with-free-path when free-path location on ground       |
| Walk to Location      | Pogo to Location        | when no free path to Location
|                       |                         | Location ⇒ Location-with-free-path when free-path location on or above ground|
| Walk to Location      | Climb Chimney to Location | when no free path to Location
|                       |                         | Location ⇒ Location-with-free-path when free-path location at base of chimney|
| Walk to Location      | Blow Down Object        | when there is a blocker to location
|                       |                         | Location ⇒ Object-blocking-path when object is destroyable                  |
| Walk to Location      | Explode Object          | when there is a blocker to location
|                       |                         | Location ⇒ Object-blocking-path when object is destroyable                  |
| Climb Chimney to Location | Pogo to Location      | when not at chimney-top
|                       |                         | Location ⇒ Location-of-chimney-top when chimney-top on or above ground      |
| Climb Chimney to Location | Walk to Location     | when not at chimney-top
|                       |                         | Location ⇒ Location-of-chimney-top when chimney-top on ground                |
| Climb Chimney to Location | Climb Chimney to Location | when not at chimney-top
|                       |                         | Location ⇒ Location-of-chimney-top when chimney-top at base of chimney      |
| Blow Down Object      | Pogo to Location        | when not at location of object
|                       |                         | Object ⇒ Location of object when location is on or above ground             |

*Table 22: Links of the full wolf, with prominent filters and transformations*
### Appendix A

#### The full specification for the wolf

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Filter Series</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blow Down Object</td>
<td>Walk to Location</td>
<td>when not at location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object ⇒ Location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when location is on ground</td>
</tr>
<tr>
<td>Blow Down Object</td>
<td>Climb Chimney to Location</td>
<td>when not at location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object ⇒ Location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when location at base of chimney</td>
</tr>
<tr>
<td>Explode Object</td>
<td>Pack TNT at Location</td>
<td>when TNT not at location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object ⇒ Location of object</td>
</tr>
<tr>
<td>Explode Object</td>
<td>Pogo to Location</td>
<td>when TNT is at location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when I’m not at the detonator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object ⇒ Location of Detonator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when location is on or above ground</td>
</tr>
<tr>
<td>Explode Object</td>
<td>Walk to Location</td>
<td>when TNT is at location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when I’m not at the detonator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object ⇒ Location of Detonator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when location is on ground</td>
</tr>
<tr>
<td>Explode Object</td>
<td>Climb Chimney to Location</td>
<td>when TNT is at location of object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when I’m not at the detonator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object ⇒ Location of Detonator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when location is at base of chimney</td>
</tr>
<tr>
<td>Pack TNT at Location</td>
<td>Pickup Object</td>
<td>when not carrying TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location ⇒ TNT</td>
</tr>
<tr>
<td>Pack TNT at Location</td>
<td>Pogo to Location</td>
<td>when carrying TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when not at Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location ⇒ Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when Location on or above ground</td>
</tr>
<tr>
<td>Pack TNT at Location</td>
<td>Walk to Location</td>
<td>when carrying TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when not at Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location ⇒ Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when Location on ground</td>
</tr>
<tr>
<td>Pack TNT at Location</td>
<td>Climb Chimney to Location</td>
<td>when carrying TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when not at Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location ⇒ Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when Location on ground</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when Location at base of chimney</td>
</tr>
<tr>
<td>Pack TNT at Location</td>
<td>Drop Object</td>
<td>when carrying TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location ⇒ TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>confictor-filter</td>
</tr>
<tr>
<td>Pack TNT at Location</td>
<td>Eat Object</td>
<td>when carrying TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Location ⇒ TNT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>confictor-filter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when TNT is edible</td>
</tr>
</tbody>
</table>

| **Table 22:** Links of the full wolf, with prominent filters and transformations |
Figure 14: The wolf’s predecessor links
Appendix A

The full specification for the wolf

Figure 15: The wolf’s conflictor links
Bibliography


Bibliography


