Believable Agents: Building Interactive Personalities

A. Bryan Loyall

May 1997

CMU-CS-97-123

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by Fujitsu Laboratories, Mitsubishi Electric Research Laboratories, and Justsystem Corporation.

Keywords: artificial intelligence, believable agents, interactive characters, personality, active language, embodied language and action, situated natural language generation, art, entertainment, Oz

Abstract

In the traditional character-based arts of film, books, animation, or theater, the audience is presented with vivid, powerful, personality-rich characters such as Rick from *Casablanca* or the Genie or Flying Carpet from Walt Disney's *Aladdin*. Unless one can afford a troop of improvisational actors (and not even then for characters such as the Flying Carpet), one is only able to watch these rich characters. *Believable agents* allow people to not just watch, but also interact with such powerful, personality-rich characters.

Believable agents are a combination of autonomous agents and believable characters from the traditional character-based arts, such as animation, film or literature. They are accurately described both as autonomous agents with the same powerful properties as characters from the arts, and as computer-based, interactive versions of personality-rich characters. Believable agents could be powerfully used for art, entertainment, as part of interactive story systems, social user interfaces, or teaching systems.

This dissertation presents my progress toward the creation of believable agents. The first result of this dissertation is a study of the problem to be solved. This study has two parts: a study of believability in traditional character-based arts, and a study of believability for agents, primarily drawing on experience in multiple attempts to construct such agents.

The second result of the dissertation is an architecture called Hap specifically designed to support the requirements and expression of believable agents. This support includes a formal language for the detailed expression of an artistically-chosen personality, automatic control of real-time interactive animation, and architectural support for many of the requirements of believable agents. The architecture is also a unified architecture for the agent in that all aspects of an agent's mind are expressed directly in Hap. By doing this the other aspects, such as arbitrary computation, computation of emotions, and recognizing patterns of activity over time for higher-level sensing, all inherit Hap's properties for believability.

The third result of this thesis is an approach for natural language generation for believable agents. This result includes an analysis of the requirements for natural language in believable agents, an architecture for supporting these requirements, and an example grammar and behaviors that use this architecture as part of an agent's behavior.

A number of agents have been built using this framework. One is described in detail to show how the technologies described in the dissertation can be used together to build complete believable agents. A public exhibit of some of the constructed agents provides evidence that this approach can achieve a certain level of believability, and that others can learn the framework and use it to build believable agents.

Acknowledgments

First and foremost I want to thank Joseph Bates. As my advisor, Joe has contributed greatly to my work and life here. Joe has shown me how to follow a dream without becoming sidetracked when the dream seems impossible. He has a broad view of research coupled with high and unwavering standards. These rare and important qualities have helped me tremendously: they have allowed me to proceed even during my cynical moments and helped evolve my view of research, which I believe is the most valuable result of the doctoral process. I don't think I could have picked a better advisor.

I also want to thank my committee, Jill Fain Lehman, Tom Mitchell and Nils Nilsson. They have each strengthened this work dramatically by their critical analysis and suggestions. Although I grumbled at the time, the improvements they suggested were well needed and worth the effort.

I want to thank the members of the Oz group for creating the special environment in which the ideas of this thesis have grown: Joseph Bates, Peter Weyhrauch, Scott Neal Reilly, Phoebe Sengers, Mark Kantrowitz, Koichi Murakami, Jon Swartz, Mike Shapiro, Alma Whitten, James Altucher, Michael Mateas, and other members over the years. In particular, I want to thank Scott Neal Reilly for our collaboration on agent architectures, and Koichi Murakami for his wide-ranging support of the ideas in this thesis and my work.

I want to also make special mention of Peter Weyhrauch. Peter and I joined the Oz group at the same time as its first two students. We have worked together since the early days when we shared a single IBM-RT, and spent many a late-night hacking session over Hal's pizza. He has the valuable trait (or curse?) of being able to stick with one of my meandering rants, whether the topic is research, philosophy or personal, long enough to get to the heart of the matter and help to solve or understand it better. My work and my time at CMU would have been much less rich without his friendship and collaboration.

Many people worked on systems with me over the years. I want to thank those who helped to create Lyotard: Mark Kantrowitz, Scott Neal Reilly, Phoebe Sengers and Peter Weyhrauch; as well as those who worked to create Edge of Intention: Joseph Bates, Andy Witkin, James Altucher, Alex Hauptmann, Mark Kantrowitz, Dirk Langer, Koichi Murakami, Drew Olbrich, Zoran Popovic, Scott Neal Reilly, Phoebe Sengers, Will Welch and Peter Weyhrauch.

I thank Charles Forgy and Production Systems Technologies for the use of RAL, Rule-extended Algorithmic Language, as the Rete implementation and target language of the Hap compiler.

For comments on early drafts of this thesis, I thank Joseph Bates, Phoebe Sengers, and Stan Wood. Where this document is readable, it owes much to their careful attention.

Even though this simple advice is surprisingly hard to follow, it repeatedly aided me in the writing process. I want to thank Avron Barr for saying to start writing early and to make sure that I write something every day. Michael Redhill provided the key insight from his years of writing that one shouldn't let worries about the blank pages of tomorrow keep today's page blank. He also provided the reassurance that it is normal to turn into a zombie when deep in the writing process.

Carnegie Mellon computer science is an amazing place to do graduate work. I want to thank those who make it that way: Sharon Burks, Catherine Copetas, Jim Skees, Karen Olack, and the rest of the administration who made red tape practically non-existent, facilities who keep the computing environment running more smoothly than any other place I have known, the visionaries that created the department and instilled it with its spirit, and those that are trying to keep this spirit in the face of changing times (it's worth it).

I want to thank those whose friendship and support helped keep me sane and happy during this process. First I want to thank my family for being a deep source of support, encouragement and inspiration: Mom, Brenda, Herb, Jake, Annette, Mike, Mark, Joe and Janel. I want to thank my office mates: Alan, Jay, Shirley, Andrew, Karl, Bill, Morgan, Eric, and Kamal. I want to particularly thank Jay and Andrew for being fountains of information on everything from Latex to "does this argument fly?". I want to thank those who have been with me since the early years: Peter, Ric, Kevin, Mike, Jim, Cathy, Will, Shirley, Lisa, Fritz, Maria, David, Tom, Koz, James, Anwar, Ali, Juan, csgrad88; and those who shared in the surly years: Peter, Andrew and Chris. By also going through the thesis process, they provided day-to-day reassurance that I wasn't going insane (or they were too), as well as general support. I want to also thank those who only knew me during the surly years and put up with me anyway: Steve, Stan, Belinda, Alma, Matt, Linda, Patty, Terri, Charmaine, Lisa, Corey, Paul and Jose.

And finally, I want to thank Debbie, for everything.

Contents

List of Figures

Chapter 1

Introduction

Imagine not just watching, but being able to interact with your favorite character from the arts. You might choose to walk into Rick's Café Américain from *Casablanca* and invite Rick to join you for a drink, knowing all along that he might refuse, but enjoying the interaction all the same. Or you might choose to be a rabbit in the woods with Elmer Fudd during rabbit season. You could have the experience of evading Elmer's gun while confusing him at every opportunity.

Enabling such interactions is the dream of this thesis. I want to build and enable others to build computer-based interactive versions of such personality-rich characters. This is the dream of creating believable agents.

1.1 What Are Believable Agents?

In this section I define believable agents in broad terms. For a deeper and more detailed understanding of believable agents, please read the analysis of requirements for believable agents in Chapter 2.

Believable agents are personality-rich autonomous agents with the powerful properties of characters from the arts. They are an outgrowth of both autonomous agent research [Agents 1997] in computer science and the notion of believable characters from traditional stories. In the traditional story arts — film, literature, drama, animation, etc. — a character is considered believable if it allows the audience to suspend their disbelief and if it provides a convincing portrayal of the personality they expect or come to expect. Believable agents are autonomous agent versions of such characters. This goal is different from the bulk of autonomous agent work in that the focus is on building agents that have distinct, specific personalities. Many aspects traditionally focused on in autonomous agent work are less important for this goal, for example, the ability of the agent to accomplish a useful task or be an effective problem solver. Other little studied qualities are crucial, like self motivation, emotion, and social facility. Nevertheless there is much overlap in the two areas. Both believable agent research and much of autonomous agent research are concerned with reactivity to the environment, general responsiveness, and existing in unpredictable worlds with people and other agents.

Although believable agents draw heavily from believable characters, they are different in that they are both computer based and interactive. The fact that they are computer based means that an author cannot rely on a human actor to bring many of the necessary qualities of believability to the agent, as one can, for example, in film or theater. This difficulty is shared by two areas of the arts, literature and animation, in that the illusion of life and personality must be constructed from simple components such as words or line drawings. Frank Thomas and Ollie Johnston, two of Disney's original influential animators, describe this difficulty eloquently when they say, "Out of nowhere we have to come up with characters that are real, that live, that interrelate. We have to work up the chemistry between them (if any is to exist), find ways to create the counterpart of charisma, have the characters move in a believable manner, and do it all with mere pencil drawings" [Thomas and Johnston 1981, p. 18]. This difficulty is shared in the task of constructing believable agents.

The fact that believable agents are interactive also separates them from characters in most traditional arts, and makes their creation even more difficult. The author of a believable agent cannot control or know with certainty the situations the agent will encounter. In a linear story, the author can bring all of his or her humanity to bear to decide how the character reacts to each situation in turn. For a believable agent some version of a "whole"[1] personality must be created. This agent must react to situations as they arise without being able to draw directly on the wealth of its creator's knowledge.

In defining the notion of believable agents, it is useful to consider some related but different goals.

1.1.1 Believable Agents and Realistic Agents

The goal of creating realistic human or animal agents has been pursued by many researchers [Badler *et al.* 1990; Thalmann and Volino 1997; Thalmann *et al.* 1997; Blumberg 1994]. This goal clearly shares much with the goal of creating believable agents, but it is a different goal. In order to create a believable agent it is not necessary to create a realistic agent. For evidence we can look to artists' creation of believable characters. They always abstract from reality, retaining only those elements of realism that contribute to conveying the personality. Thomas and Johnston say, "The more an animator goes toward caricaturing the animal, the more he seems to be capturing the essence of that animal .... If we had drawn real deer in *Bambi* there would have been so little acting potential that no one would have believed the deer really existed as characters. But because we drew what people imagine a deer looks like, with a personality to match, the audience accepted our drawings as being completely real" [Thomas and Johnston 1981, p. 332].

[1]How complete the personality has to be is limited by a number of factors including the expectations of the human interactor, the world in which the agent exists, the other agents present, and the possible length of the interaction.

This does not mean that the abstraction or caricature of realism must be obvious or extreme, as it is in much comic art, such as Roger Rabbit's movements and expressions in the movie *Who Framed Roger Rabbit?* or many of Lucy's expressions in the popular television show *I Love Lucy*. These obvious caricatures can be very effective in such comic expression, but for other settings, choosing the appropriate abstraction is key. This is an area where a wealth of knowledge from existing art can be applied. Consider how audiences understand and relate to characters such as Snow White, Aladdin, and Bambi. These characters seem real to the audience. They don't seem to be extreme caricatures at all. Yet if one examines them, they are still far removed from realism.

Believability does not require human form. This is suggested by the existence of characters such as Bambi. It is shown even more strongly by the character of the Flying Carpet in the Disney animated film *Aladdin* [Clements and Musker 1992]. It has no way of being realistic, it is a totally fantastic creature. In addition, it does not have many of the normal avenues of expression: it has no eyes, limbs nor even a head. It is only a carpet that can move. And yet, it has a definite personality with its own goals, motivations and emotions.

Writing in the arts on this subject is somewhat confusing on the surface. Walt Disney would often ask for more "realism" when critiquing a piece of animation, and Stanislavski was a champion of a movement in acting called "Realism". Yet, they both also criticize realism in their art forms. This confusion has more to do with the limitations of English (or Russian) as a medium of expression than it does with any disagreement. Thomas and Johnston repeatedly emphasize what Disney was asking for:

> There was some confusion among the animators when Walt first asked for more realism and then criticized the result because it was not exaggerated enough . . . . When Walt asked for realism, he wanted a caricature of realism. One artist analyzed it correctly when he said, "I don't think he meant 'realism,' I think he meant something that was more convincing, that made a bigger contact with people . . . " [Thomas and Johnston 1981, pp. 65–66].

Similarly, when one looks closely at Stanislavski's writing, it is clear that the confusion is one of terminology, as is shown by this summary of his views comparing Realism with Naturalism, a form of acting he seemed to think little of. "Naturalism, for him, implied the indiscriminate reproduction of the surface of life. Realism, on the other hand, while taking its material from the real world and from direct observation, selected only those elements which revealed the relationships and tendencies lying under the surface. The rest was discarded." [Benedetti 1985, p. 11]

1.1.2 Believable Agents and Intelligent Agents

One of the primary focuses of artificial intelligence is the construction of intelligent agents, and the creation of intelligence in other, non-agent, forms. Although this goal has some commonalities with believable agents, they are not the same goal. Intelligence is not a central requirement for believable agents.

This is not to say that general competence is not needed for believable agents. Competence is needed in a number of areas, for example, routine physical activities in which the agent engages, social knowledge for interactions with other agents and people, etc. But the primary focus for believable agents is not *intelligence* in these activities or in general. Instead, the focus for believable agents lies elsewhere.

Consider characters from the arts. What are the properties that makes the characters portrayed by Humphrey Bogart, Meryl Streep, and Charlie Chaplin powerful and believable? These characters appear to be aware, to have their own internal drives, and to have emotions and social relationships. Each of these properties is *artistically chosen* and *combined* by both the author and actor to express a particular personality. Consider a character such as Elmer Fudd from the classic Warner Brothers cartoons. For his personality, intelligence is not only not central, it would actually undermine his character.

Character-based artists in many media have struggled to capture what is needed for believability. They describe these qualities and many more. (A detailed analysis of these requirements for believability is the subject of Chapter 2.) Their experience suggests a different focus than traditionally taken in pursuit of artificial intelligence, and if one wants to create believable agents it is this new focus that must be understood and pursued.

1.2 Areas of Contribution

The goal of this dissertation is to make progress toward the construction of believable agents. When I started this work, I thought it would primarily involve work in an agent architecture and theory of mind. But a fundamental goal from the beginning was to actually *build* complete believable agents[2]. This goal to build a complete agent has led this work to touch on many areas that I hadn't initially intended. It still is primarily a work in believable agents, and contains a strong agent architecture component, but it now relates to work in computer graphics, theories of emotion in an agent, theories of activity, natural language processing and even programming language design. Many of these have only been touched upon by this research as necessary for the work. Had there been time, increased effort in all of these areas would have been useful for advancing the work. Some specific areas of advancement are addressed in the discussion of future directions in Section 11.3.

Of the areas this work to some degree includes, there are three main areas of contribution: believability, agent architecture, and natural language generation.

1.2.1 Believability

The main contribution of this thesis is in believability itself. This thesis presents the results of a study of what believability means for agents. This study draws both on what is known about believability in the character-based arts, as well as what has been learned by

[2]Although at that point we didn't call them believable agents. Bates, Neal Reilly and I coined that term a few years later for our internal discussions. It and related terms such as life-like computer characters became more widely used after the 1994 AAAI Spring Symposium on Believable Agents.

attempting to build interactive, autonomous versions of such characters. The results of this study have been used to guide the technology built and the use of that technology for the construction of agents.

1.2.2 Agent Architecture

In agent architectures, this thesis presents an architecture called Hap specifically designed to support the requirements and expression of believable agents. This support includes a language for the detailed expression of an artistically-chosen personality, automatic control of real-time interactive animation, and architectural support for many of the requirements of believable agents. The agent architecture is a unified architecture for the agent in that all aspects of an agent's mind are expressed directly in Hap. By doing this the other aspects, such as arbitrary computation, computation of emotions, and recognizing patterns of activity over time for higher-level sensing, all inherit Hap's properties for believability.

1.2.3 Natural Language Generation

In natural language generation, this thesis presents steps toward natural language generation for believable agents. This work includes an analysis of requirements for NLG for believable agents, technology for the direct expression of natural language generation in Hap as an approach to addressing those requirements, and examples of uses of the system in an agent.

1.3 A Few Words About This Research and Ideologies

Research in AI is often marked by strong ideologies: connectionism, behavior-based AI, artificial life, etc. The next section describes some of these in a survey of previous work. Often such ideologies take rather extreme stances with respect to previous approaches.

Although it may not appear so at first glance, this work attempts to be ideology free. The central methodological goal in my work is to make progress toward believable agents. This sometimes gives rise to apparent ideological stances, but these positions arise solely from the goal to make progress toward believable agents. I freely embrace approaches from any ideology or combination of ideologies that allows the best progress toward believable agents.

In pursuit of this work, I began with a traditional planning and learning approach. Through increasingly understanding the problem of building believable agents, the approach has shifted to the one presented in this thesis.

1.4 Background

No work is done in a vacuum. This research draws on and is related to many earlier works in artificial intelligence and art. In this section I describe other work that has influenced mine. First, I describe the project in which my work has been done, focusing on the direct influences of the group on the work described herein.

Second, I describe the work of others in artificial intelligence and art that was done previous to or in the early stages of my work and that influenced it. I describe work in three areas: agent architectures, natural language processing and believability.

In the conclusion (Chapter 11), I describe and compare my work to more recent related works.

1.4.1 The Oz Project

The work described in this dissertation was pursued as part of the Oz project led by Joseph Bates at Carnegie Mellon. The goal of the Oz project is to enable people to create and participate in interactive stories. Toward this goal we study interactive versions of the elements that we believe make traditional non-interactive stories powerful: characters, dramatic structure (plot), and presentation. For a deeper understanding of the Oz project please see [Bates 1992].

Believable agents are necessary for many models of interactive story. (A survey of approaches to interactive story including the role of agents in the approaches is given in [Weyhrauch 1997].) In addition, believable agents have other applications. There is increasing interest recently in various models of "friendly" or social user interfaces, for example [Ball *et al.* in press] for which believable agents are an important part. The advantages of believable agents as part of tutoring or teaching systems is also recently receiving attention, for example [Lester and Stone 1997]. For me personally, the most interesting application for believable agents is to art and entertainment. These agents could be used as part of such art or entertainment systems, as they are in interactive stories for example, or used alone as the center of the art or entertainment experience.

The Oz project is a rich culture in which to develop this work because of the freedom to pursue believability purely, without the requirement that the agents also be assistants, user interfaces or even part of an interactive story. It is a culture that values the entertainment and artistic applications of the work. Also, it has many synergistic research pursuits that have influenced this work. Work in this project is described in several Ph.D. dissertations and research papers [Neal Reilly 1996; Weyhrauch 1997; Kantrowitz in press; Sengers 1996; Kelso *et al.* 1993; Loyall and Bates 1993; Bates *et al.* 1994; 1992; Kantrowitz and Bates 1992].

When work is done as part of a research group, there is always the question of what work was individually done and what was done by others in the group. Traditionally, and in my case, the most significant outside contributor is the advisor. This work has profited by many discussions with my advisor, Joseph Bates, especially during the early phases. In addition, I want to point out four inclusions of other's work in this dissertation that are included for completeness or for illustrative purposes:

- In Chapter 3, I describe an example domain. I include this description as concrete example of the types of domains the work in this dissertation is designed to support. It is a real-time, animated world with low-level actions and sensing. This domain was created by several people in the Oz group as well as the Graphics group at Carnegie Mellon.

- Chapter 6 describes the model of emotions that is an important and integral part of the agent architecture for believable agents presented in this dissertation. The emotion model itself is the work of Scott Neal Reilly, and presented in more detail in his thesis [Neal Reilly 1996]. The integration of this model with the agent architecture is joint work with Neal Reilly.
- The minds of agents described in Chapter 7 were created by myself and four members of the Oz group. They are included to illustrate how all of technology described in the rest of the dissertation can be used together to construct a complete agent.
- The approach to natural language generation for believable agents described in Chapter 8 was influenced by Mark Kantrowitz's stand-alone natural language generation system [Kantrowitz and Bates 1992; Kantrowitz 1990]. Much of the traditional grammar is directly translated from his grammar.

1.4.2 Agent Architectures

There is a tradition of work on agent architectures that stretches back to the beginnings of AI. To survey it all is beyond the scope of this section. Instead, I focus here on the agent architecture work that influenced or is most closely related to my work.

Tradition of Building Complete Agents

The problem of building agents or robots that can function reasonably in dynamic worlds or in interactions with people is not unique to believable agents or even to recent trends in artificial intelligence and robotics. Making progress on this problem has been a pursuit of work in Computer Science since its beginnings. Many of these early approaches give strong insights into how to build integrated autonomous agents, agents that can function in dynamic worlds, or agents that interact with people. These early works can also be seen as precursors of currently active research approaches and trends.

In 1963 W. Grey Walter built two machines, called M. Speculatrix and M. Docilis, [Walter 1963] that looked like armored tortoises. They each had an "eye" (a gas-filled photo-electric cell on a rotatable rod), and a touch sensor, and later models included a microphone. These sensors were hard-wired through some circuitry that was connected to effectors that allowed it to move and rotate its "eye". The systems roamed around seeking light. When none was found they moved around searching for light. When they found light they moved toward it but did not get too close, and when the touch sensor was activated the output of the light sensor was modified resulting in behavior that caused the machines to push small obstacles away, go around heavy ones, and avoid slopes. A combination of their behavior and the construction of their recharging stations (which had very bright lights), enabled them to wander around while they were well charged and gradually find their way back to their recharging hutches. Later versions of the system included learning. While in many ways they were very simple, these machines included some of the essential properties of reactive architectures and situated theories of activity, on which my work builds. These machines also have clear relations to artificial life. Braitenberg's *Vehicles* [Braitenberg 1984] is later work that is similar (if differently motivated).

One of the early works of an integrated autonomous agent that could be characterized as a reactive architecture is the work of Fikes, Hart and Nilsson on the "Shakey" mobile robot [Fikes *et al.* 1972]. As Nilsson describes: "From 1966 through 1972, the Artificial Intelligence Center at SRI conducted research on a mobile robot system nicknamed 'Shakey'. Endowed with a limited ability to perceive and model its environment, Shakey could perform tasks that required planning, route-finding, and the rearranging of simple objects" [Nilsson 1984]. Shakey used the STRIPS planner [Fikes and Nilsson 1971] for deliberation, and created *triangle tables* for execution that allowed the execution module to be reactive to changes in the world, including taking advantage of opportunities. While very different from the approach I take in this thesis for supporting autonomous agents which are situated and reactive in the face of dynamic worlds, the work on Shakey has clearly influenced the architectures that came after it.

Behavior-Based and Situated-Activity Approaches

The behavior-based and situated-activity approaches of the middle 1980's arose because of perceived weaknesses in the dominant approaches to agents being pursued at the time. These approaches were largely planning-based approaches that emphasized deliberation. The early behavior-based and situated-activity approaches included Brooks's subsumption architecture [Brooks 1986], Agre and Chapman's Pengi system [Agre and Chapman 1987] and Suchman's critique [Suchman 1987].

The emphasis in all of these approaches was that execution in a dynamic world is a hard problem that requires different techniques than the planning-based approaches of the time. Brooks's architecture used minimal representations, and small hard-wired control structures to control robots in the real world. He advocated "using the world as its own representation" by directly driving action from raw sensory data. These control structures were organized by behaviors such as "explore" and "avoid obstacles", rather than by functionalities such as vision, deliberation and execution. By designing the agent architecture in this way, he substantially avoided the need for goals, representations, etc.

This approach yielded surprisingly robust and effective robots that could navigate, explore, follow each other, and in general survive in the dynamic real world.

Agre and Chapman pursued a different, but similarly motivated, line of research. They claimed that the plans that people use are often situated plans, that is, they only make sense when interpreted in the current situation. For example, plans to go somewhere often are of the form "go to the gas station and turn right"; the meaning of "the gas station" and "turn right" depends completely on the situation during execution. They developed a theory called "indexical-functional representation" to capture representations for such situated plans, and a theory of activity that structured its action using such situated plans. They tested the approach in dynamic worlds of real-time video games [Agre and Chapman 1987; Chapman 1990], again with strong results.

Reactive Architectures and Relation to My Work

Reactive architectures combine the power of these approaches with some of the qualities of the previous deliberative approaches, for example explicit goals and the ability to easily express deliberation. My agent architecture, described in the following chapters, is in part a reactive architecture.

Firby's Reactive Action Packages [Firby 1989], Kaelbling and Rosenschein's Gapps formalism [Kaelbling and Rosenschein 1990], Georgeff and Lansky's Procedural Reasoning System [Georgeff and Lansky 1987], Simmons's Task Control Architecture [Simmons 1991], Newell and colleagues' Soar architecture [Pearson *et al.* 1993; Newell 1990; Rosenbloom *et al.* 1993], and Nilsson's Teleo-reactive architecture [Nilsson 1994] are all reactive architectures. All of these systems combine explicit goal representations with performance characteristics that allow them to be responsive to changes in dynamic worlds as they occur. They allow low latency control paths (in the form of high-level demons, compiled circuits with short paths, or other mechanisms) to respond to time-critical events such as emergencies in a robot assistance domain, or getting out of the way of a speeding car for real-world navigation domains. Each of them also allows for higher-latency control paths to encode more deliberative activities, such as the carrying out of routine plans.

The original version of Hap, my architecture for believable agents (described in detail in later chapters) was developed in 1989. At its core it is a reactive architecture that is most similar to Firby's Reactive Action Packages and Georgeff and Lansky's Procedural Reasoning System. The creation of a reactive architecture was deliberate, because the properties of such architectures (e.g. reactivity, responsiveness, deliberation, and explicit goals) seemed important for believable agents as I discuss in Chapter 2. Beyond a basic reactive architecture, Hap has been extended in four ways to support the needs of believability. It has been tightly integrated with a model of emotion. It has been extended to support the needs of believable motion in real-time, animated worlds. It was extended to function as a unified architecture that supports all aspects of an agent's mind. And it has been built and extended to be a language for the detailed expression of personality-specific behavior. Each of these is described in more detail in the following chapters.

1.4.3 Natural Language Generation

Appelt's seminal system KAMP [Appelt 1985] combined action and natural language generation to accomplish communication goals. His approach was planning-based, using a traditional planner to reason about the information properties of both actions and language production. Combining language and action to accomplish communication goals is one of the requirements for believable agents that talk. Other requirements are described in Chapter 8.

Nearly all generation systems produce language based on the content to be expressed or the content plus what has been said before (the discourse history). This produces language that is unvarying in the style of presentation. Hovy's Pauline [Hovy 1988] was the first system to vary its output based on pragmatic goals for the utterance. Pauline produced different language if it was rushed than if was not; different language if it was trying to convince you than if it was trying to be objective; and different language if the one it was speaking to was its boss than if the hearer was a peer. Such variation greatly enriches the style of the language produced, and is clearly needed for believable agents that talk.

1.4.4 Believable Agents

The area of believable agents is new. The term itself only came into prominent use sometime after the 1994 AAAI Spring Symposium on Believable Agents. Later that year the term *lifelike computer character*[3] was coined by the creation of the Lifelike Computer Characters Workshop. This workshop was inspired by the Believable Agents Symposium and has been held yearly since then.

When starting to pursue this work in 1988, without any directly related previous work, the previous work that influenced my approach to believable agents was in three areas: artificial intelligence based story generation and understanding, computer graphics and the character-based arts.

AI Story Generation

There has been a wealth of work in story generation and understanding over the years. The most relevant of these systems to the work presented in this thesis is Meehan's Talespin [Meehan 1976]. Talespin created stories through the interactions of computer controlled agents. In a run of his system, the world was given an initial condition. Each agent had goals, states and relationships that were provided by the user initially and in response to queries by the system. Through a combination of the user carefully coordinating the answers to these questions, and the system planning and executing actions to accomplish each agent's goals, the system produced stories similar to simple Aesop fables.

Computer Graphics

Witkin and Kass [Witkin and Kass 1988] have shown that computational methods can be used to produce compelling animated movement. Their techniques use physically-based

[3]*Lifelike* and *believable* are both terms borrowed from the character-based arts. I use the term *believable* throughout this thesis because its meaning is less ambiguous. Lifelike characters in the arts and lifelike computer character in computer science are sometimes used to connote believability, but at other times these terms are used to denote work that focuses on realism, cognitive plausibility, or other concepts not necessarily central to believability.

simulation, with an emphasis on realistic motion. The resulting movement is compelling, showing elements of anticipation, squash-and-stretch, and follow-through that rival those produced by trained animators for the basic mechanical level of movement. It is not clear how to extend such physically-based methods to the non-realistic, but expressive low-level movements often found in the arts, or to the higher-levels of a character's movement. This work is also not currently able to be used in interactive systems or real-time agents, but this work is the first to suggest that for some types of movement, computer generated movement can be of the same quality as artistically generated movement.

Reynolds work at controlling flocks of birds [Reynolds 1987] was concerned with the expressive movement of groups of agents. Each of the elements in Reynolds's flocks is a simple autonomous agent that senses the location of nearby agents and obstacles and modifies its movement relative to them. It doesn't directly provide insight into how to create individual believable movement, but his agents do generate compelling movement as a group.

In 1987, John Lasseter published a paper entitled "Principles of Traditional Animation Applied to 3D Computer Animation" [Lasseter 1987a] in the 1987 SIGGRAPH Conference Proceedings. This paper was after an academy award nomination for one of Lasseter's computer animated films in the Best Animated Short Film Category. (In 1988 another film directed by Lasseter won in that category, becoming the first computer-animated film to be recognized with an academy award.) Lasseter claims that his SIGGRAPH paper was in response to people asking him what secret allowed him to create such powerful computer-animated films. His answer, as detailed in his paper, is the principles of traditional animation, as learned in the arts, applied to computer-based animation.

This leads directly to the most direct prior work for the pursuit of believable agents, the traditional character-based arts.

Character-Based Arts

Unlike academics, most artists create artifacts and say little about the method by which they create them. Fortunately, a few of the great character artists have not only created great characters, but also struggled to understand and convey *how* to create powerful characters. These are the related work from the arts on which I draw in my work.

Character-based animation, of the kind done by Disney [Thomas and Johnston 1981], Warner Brothers [Jones 1989] and Pixar [Lasseter 1987a], is concerned with the problem of how to convert a stack of blank pieces of paper into a believable animated character on the movie screen. In this process, animators have insight into the fine details of movement as well as the details of how to convey a personality in a series of pictures.

Writers both of literature [Gardner 1984] and drama [Egri 1960] have a similar difficulty to the one faced by animators. Animators must start with nothing and create a character out of a sequence of drawings and sound. Writers start with nothing and create characters through words. Because of their medium, they have the option of conveying the character through the expression of internal thought as well as action and description.

Actors are of two camps. Actors such as John Wayne or Clark Gable play the same character (often a version of their own personality) for every role. Since they are portraying their own humanity, there is little one can learn from them about how to create a given character in interactive form. Others, such as Dustin Hoffman, Robert De Niro and Marlon Brando, are admired as actors for being able to change radically from role to role. They are coveted for their acting "range". What they know about expressing a particular character could be very valuable to the building of believable agents. Constantin Stanislavski was not only an actor of this second type, but is also regarded by many as the most influential actor and thinker on acting in the 20th century. He worked for decades to understand how to effectively portray a character. His writings [Stanislavski 1968; 1961] on the topic are invaluable for the study of believability.

The rest of this thesis attempts to apply what these artists know to the construction of believable agents.

1.5 A Reader's Guide to the Dissertation

This dissertation contains many parts, and readers may not want to read all of them. Here are some suggestions for people who only want to read part of it or who are deciding whether they want to read more.

- For those who want to know what I mean by believable agents, I suggest reading the first part of Chapter 1 through Section 1.1 for a broad description, then read all of the requirements for believable agents in Chapter 2 and the issues and guidelines in Chapter 10.

  If after that you want more, I suggest reading the requirements for natural language generation for believable agents in the introduction to Chapter 8, and skimming the technical sections for discussions of issues of believability.
- Readers who want to know how this work relates to other work should read the description of related work which influenced mine in Section 1.4, and the comparison to related technical work in Section 11.2.
- For those interested in what the agents constructed using this technology are like, I suggest skimming Chapters 4, 5 and 6 for an understanding of the technology, and then reading the examples of processing in Sections 4.15, 7.8 and 8.5. For more details read the detailed description of an agent in Chapter 7. The reader might also be interested in how others have reacted to agents constructed in this architecture. Some observations are described in Section 9.2.1.
- For readers interested in the steps I've taken toward natural language generation for believable agents, Chapter 8 is fairly self contained.
- Readers interested in the details my action architecture, Hap, should read Chapter 4, and skim or read Chapters 5, 6 and 8.

For those who want to know it all, I include a description of each chapter in order.

Chapter 1 describes what I mean by believable agents in broad terms, presents the areas in which this research makes contributions, describes other technical and artistic work that has influenced this work, and gives this guide to the document.

Chapter 2 describes in detail the requirements for believable agents. These requirements come from the experience of artists who have been working to create believability for millennia, and from the special needs of autonomous agent versions of these believable characters.

Chapter 3 describes an example domain for believable agents. This is included to give a deeper understanding of the task I am attempting to accomplished by presenting the building blocks a created agent has to work with, and to give a foundation for the example agents and behavior that will come later in the dissertation.

Chapter 4 is a technical description of my agent architecture for believable agents.

Chapter 5 (along with Chapters 6 and 8) describes how Hap can be used as unified architecture for all aspects of an agent's "mind". This chapter describes how composite sensing and the "thinking" of an agent are expressed in Hap.

Chapter 6 describes the emotion model used in Hap, and how it is integrated with and expressed in Hap.

Chapter 7 presents a detailed description of a complete agent expressed in this architecture to show how all of the parts work together for the expression of a believable agent.

Chapter 8 describes my steps toward natural language generation for believable agents. It includes an analysis of the requirements, extensions to Hap to allow the direct expression of natural language generation knowledge, and an analysis of the properties this approach gives for addressing the needs of believable language production.

Chapter 9 presents an analysis of the progress this work makes toward believable agents, and empirical evidence of that progress.

Chapter 10 presents conjectures that I believe are central and crucial to making progress in this area.

Chapter 11 includes a summary of the results of this dissertation, comparison of this work to other related work, and future directions for the work.

Chapter 2

Requirements for Believable Agents

Little as I knew about movies, I knew that nothing
transcended personality.
– Charlie Chaplin [1964, p. 142]

This chapter presents an analysis of the requirements for believable agents. There are two main sources on which I draw in this analysis: artists and experience with agents.

Learning from character-based artists is centrally important. These artists have been studying believability, and advancing the art for millennia. Not attempting to learn what they know would doom us to reinventing and relearning many important lessons. Thomas and Johnston claim that the principles at the core of their art of animation were the same ones known for 2,000 years by artists in other media, but that they learned them painfully and slowly by trial and error [Thomas and Johnston 1981, p. 27]. By studying what character-based artists know about believability, I hope to partially avoid this fate.

Some of the requirements for believable agents are not known by character-based artists. This is because, for all of the similarities in the problems, the fact that we are working to create *agents* rather than *characters* causes and emphasizes additional difficulties. All of the traditional character-based artists are able to draw on their own humanity for capabilities that a designer of believable agents must explicitly create. All actors, animated characters, and characters in books are able to perform multiple actions at the same time, be reactive to their environment, and appear to have other basic human capabilities. For believable agents to have these same qualities, we must recognize them as requirements and build them into the agents.

In this description, I start by describing requirements from the character-based arts, and end with requirements from autonomous agents.

2.1 Personality

The most important single requirement for believable agents is *personality*. Thomas and Johnston, two of Disney's most influential animators, state the importance of this requirement when talking about radio dramas and animated movies that are so powerful that they draw the audience in and cause the characters to come to life in the audience's imaginations.

To achieve this level of involvement, they claim, requires personality:

> For a character to be that real, he must have a personality, and, preferably, an interesting one. [Thomas and Johnston 1981, pp. 19–21]

In their book, Thomas and Johnston go on to describe over many chapters the wealth of details that are specified to bring their characters to life. This is what I mean by the requirement of personality:

Personality: all of the particular details — especially details of behavior, thought and emotion — that together define the individual.

Every character has a wealth of such details that work together to bring it to life. Consider for a moment two characters from the movie *Casablanca*: Rick (Humphrey Bogart), the cynical owner of the Café Américain, and Captain Renault (Claude Rains), the cavalier and corrupt law in Casablanca. These are both powerful, effective characters, and there is almost nothing that these two characters do in the same way. They both drink, but for seemingly different purposes and in dramatically different ways. They talk differently. They are motivated by different desires. They hold themselves differently and walk differently. They interact with the other characters differently. The list of differences goes on and on. It is an interesting thought experiment to consider how much less powerful, less believable, they would be if they both drank in the same way, or if their other behaviors were alike. I believe that it is the wealth of these carefully chosen individual details that make them both powerful characters.

Artists who create characters repeatedly make this same point. Writers and playwrights also talk about this requirement. They often describe the need to know their characters in depth to ensure that their characters' personalities come through in their actions, responses and dialogue [Egri 1960; Gardner 1984]. When describing how to create good characters for a drama in his book *The Art of Dramatic Writing*, Lajos Egri claims:

> When you read drama criticisms in your daily papers you encounter certain terminology time and again: dull, unconvincing, stock characters (badly drawn, that is), familiar situations, boring. They all refer to one flaw — the lack of tridimensional characters. [Egri 1960, p. 35]

"Tridimensional characters" refers to Egri's system of creating and knowing completely your characters by specifying details along three dimensions: physiology, sociology, and psychology. He goes on to say:

> You must know all of these, but they need not be mentioned. They come through in the behavior of the character, not in any expository material about him. . . . You must know what your character is, in every detail, to know what he will do in a given situation. [Egri 1960, p. 42]

This same concept arises in acting. It is what actors "get into" when they "get into character". Stanislavski claims:

> Any role that does not include a real characterization will be poor, not lifelike . . . . That is why I propose for actors a complete inner and external metamorphosis. [Stanislavski 1968, p. 18]

If we want believable agents that are as real as these characters, we must somehow get the same level of detailed richness and individuality, in a word — personality, in our autonomous agents as are present in these traditional characters.

I want to make a small digression to discuss terminology. Various words are used by artists to refer to the concept of personality, often in the same body of writing. Chuck Jones, creator of many of the great Warner Brothers animated shorts and characters, refers to this concept both as "individuality" and "personality" at different times in his book *Chuck Amuck* [Jones 1989]. Actors and writers often refer to it as "character", as is evidenced by the common phrase "getting into character" to mean when an actor dons the persona he is to portray. All of these words are appropriate to describe this concept, but for the purpose of this thesis, I will use "personality" rather than "character" or "individuality" for two reasons. First, "individuality" does not seem to evoke the character aspects of this concept that seem central to its meaning. And second, I will often need to use the term "character" when referring to non-interactive believable agents (i.e. traditional characters from the arts).[1]

To further explore the range of the details that are needed for this requirement of personality, let's consider two partial character descriptions from the arts. The first is Charlie Chaplin's description of his famous Tramp character that he improvisationally presented to his coworkers shortly after he conceived the character. This early conversation is captured in his autobiography.

> You know this fellow is many-sided, a tramp, a gentleman, a poet, a dreamer, a lonely fellow, always hopeful of romance and adventure. He would have you believe he is a scientist, a musician, a duke, a polo player. However, he is not above picking up cigarette butts or robbing a baby of its candy. And, of course, if the occasion warrants it, he will kick a lady in the rear – but only in extreme anger! [Chaplin 1964, p. 144]

Chaplin says that this description of the Tramp continued for 10 minutes or more at the time. The personality of the tramp also evolved after this initial wealth of details were conceived, but even in this small excerpt we see the range of specific details that brought this character to life. He talks of motivations and desires: "always hopeful of romance and adventure", "He would have you believe ..."; emotions: "a lonely fellow"; specific actions: "picking up cigarette butts" and "kick a lady in the rear"; and he appeals to a number of seemingly contradictory stereotypes to describe the richness and depth of the details of this personality.

[1]My use of this term is also different than the technical meaning from psychology.

An important point is that he has the advantage of talking in such evocative generalities, whereas we do not. He can rely on his own knowledge and skill as he is acting to interpret these generalities and convert them to details. The builder of believable agents must actually construct these details of personality in his agents if such details are to exist.

Another example is a partial analysis of Mae West by Thomas and Johnston [Thomas and Johnston 1981, p. 546]:

Characteristics of Mae West

1. Always moving, swaying, weaving in typical poses (hand on hip or arranging hair) Rotation of body not like hula dance, but from the front.
2. Walk: Plants foot, shifts weight. Longer time on shifting than planting.
3. Talks: Drops eyes, comes up with them half-closed. Talks out of side of mouth. Little lip movement. Sideward glances—looks away disinterestedly. Starts conversation and walks away. Elusive.
4. Arms and shoulders—overlapping with body. Leave body rather than being right with it.

If you are like me, this description immediately evokes a rich image of Mae West's personality (and of Jenny Wren, the animated Disney character based on Mae West). These details of movement and mannerisms along with details of motivation, desires, goals, etc. are always present in believable characters from the arts, and are a necessary requirement for believable agents as well.

These two examples help to illustrate the wide range of details that must be specified to bring a character to life. But it is further emphasized by artists. Thomas and Johnston make the seemingly extreme statement that: "No two scenes should ever be alike and no two characters should ever do something the same way" [Thomas and Johnston 1981, p. 135]. Egri also takes this position: "No two individuals react identically, since no two are the same" [Egri 1960, p. 38].

This point can be underscored by looking at perhaps the most mundane activities of characters in the performing arts (and by some measure one of the most studied in autonomous agents) — locomotion. Typically when builders of autonomous agents create a set of agents, they build a single locomotion controller (or in some cases a few different controllers). For the purpose of *personality* and therefore for the purpose of creating a believable agent, this single mechanism or limited variation is not enough.

> Walks ... do so much to reveal personality and attitude that they are one of the animator's key tools in communication. Many actors feel that the first step in getting hold of a character is to analyze how he will walk. Even the most casual study of people on a busy street will reveal dramatic differences in how they achieve simple locomotion: long strides, short steps, determined waddling, bouncing, mincing, swaggering, rippling; there is no end to the variety. [Thomas and Johnston 1981, p. 347]

Any approach that attempts to create believable agents must allow (even require) personality-based variation to this depth, all across the wide-ranging areas of the agent.

2.2 Emotion

The second requirement for believable agents is that they appear to have emotional reactions and to show those emotions in some way. This is such a fundamental requirement in the traditional character-based arts that often artists refer to expressing the emotions of their characters as what brings them to life:[2]

> From the earliest days, it has been the portrayal of emotions that has given the Disney characters the illusion of life. [Thomas and Johnston 1981, p. 505]

In acting, as well, the portrayal of the emotions of the character is accepted as centrally important to bring a part to life. Nearly all methods for acting give techniques to make these portrayals convincing for the character being portrayed. Stanislavski devotes a good bit of his technique to understanding and conveying these emotions convincingly by drawing on experiences from the actor's own memory[3].

If agents we create are to be believable, they must appear to have emotions and expressions of those emotions that are true to their personalities.

2.3 Self Motivation

The third requirement for believable agents is that they be self-motivated. It is common in artificial intelligence to create autonomous agents that only react to stimuli, and do not engage in action of their own accord. This was also true of animation prior to 1930. When discussing how this changed, Thomas and Johnston use the phrase "really appearing to think" for the idea of self motivation:

> Prior to 1930, none of the characters showed any real thought process ... the only thinking done was in reaction to something that had happened. Mickey

[2]These same artists also refer to other aspects as the central cause of bringing the character to life. Each of these is important, and I have described and included them in this chapter.

[3]This method is described briefly in *Stanislavski's Legacy* [Stanislavski 1968, pp. 187–8].

> would see [something], react, realize that he had to get a counter idea in a hurry, look around and see his answer, quickly convert it into something that fit his predicament, then pull the gag by using it successfully.
>
> Of course the potential for having a character really appear to think had always been there . . . , but no one knew how to accomplish such an effect. . . . That all changed in one day when a scene was animated of a dog who looked into the camera and snorted. Miraculously, he had come to life! [Thomas and Johnston 1981, p. 74]

The fact that such a small, and seemingly inconsequential, action as snorting at the camera could be so powerful is an indicator of the importance of this requirement. Pluto snorting was not what was powerful; it was that he was doing it of his own accord, instead of in reaction to some external stimulus.

The power of this requirement in autonomous agents has been repeatedly underscored for me by my experience with people interacting with the test agents the Oz group and I have created. Repeatedly, people interacting with these agents are surprised and seem to have a deeper respect and interest in the agents when they realize that many of the things they do are not in response to stimuli, but rather a product of the agent's own internal drives and desires.

2.4 Change

That characters grow and change is believed by many artists to be crucial to the believability of the character. Lajos Egri devotes a lengthy section in his treatment of characters in dramatic writing on the subject of growth. In it he claims:

> There is only one realm in which characters defy natural laws and remain the same – the realm of bad writing. And it is the fixed nature of the characters which makes the writing bad. [Egri 1960, p. 60]

This does not, however, mean that the growth and change can be arbitrary. This change must be in line with the personality of the character or agent. Egri claims:

> Only in bad writing does a man change without regard to his characteristics. [Egri 1960, p. 65]

It is these two aspects that make this requirement particularly difficult. One could imagine using machine learning mechanisms, or other automatic mechanisms to give a believable agent change. But how does one ensure that the resulting change will be true to the personality in which it occurs? This is the challenge of this requirement for believable agents.

This power of this requirement has also been supported by my experience with interactive agents. The first substantial test agent our group created was a housecat named Lyotard

[Bates *et al.* 1994]. One of the aspects of Lyotard's personality was that it could grow to like or dislike the human interactor as a result of the interactions it had with him. (This growth was automatic, but the ways that the liking or disliking was shown were carefully constructed to be true to the personality of Lyotard.) The informal interactions of people with Lyotard suggests that this growth added to its believability.

2.5 Social Relationships

In nearly all forms of the character-based arts, the characters interact with other characters. These interactions are influenced by whatever relationships the characters have with one another, and those interactions in turn may influence the relationships. The power that these relationships, when done well, bring to making the character seem alive is widely agreed upon by artists. Stanislavski mentions its importance in his advice to actors, and he devotes one of his short papers to emphasizing this topic [Stanislavski 1968, pp. 133–4].

In animation, Thomas and Johnston claim that the discovery of the "character relationship" was one of the great advances of their art. Animated films had characters interacting before this, but different animators would draw each character. The advancement of "character relationship" only arrived when the same animator drew all of the characters in a scene and could then concentrate on the nuances of their relationships and interactions. Thomas and Johnston say:

> One of the first examples of this was the sequence of Bambi and Thumper on ice. . . . the Bambi and Thumper sequence had something that the [earlier] Pluto and Donald sections did not have. That was a character relationship with strong beginnings in the story department. . . . With this as a springboard, the animator continued developing this relationship, which only could have been done by one person handling both characters and completely controlling every single bit of action, timing, and cutting. . . .
>
> This new way of working with character relationships encompassed the whole range of relations between two or more characters–from the broadest to the most delicate. It involved expression scenes that often registered the most secret thoughts and inner emotions of the characters, which as they became more subtle were also more revealing. [Thomas and Johnston 1981, p. 164]

That artists believe the social relationships of their characters is central for believability is clear. But in addition, the experience of these artists points out the difficulty of this requirement. That two characters are declared as "friends" or "enemies" is not enough. Detailed behaviors and interactions that show those relationships are also needed. These behaviors need to be specific to the personality, because we know that no two characters behave the same, but they must also be specific to the relationship and other character being interacted with. In the sequence of Bambi and Thumper on ice, it is unlikely that Thumper would have reacted the same had the character having trouble been Pluto rather than Bambi. This is true even if Thumper and Pluto were friends, in part because Pluto is not as vulnerable a character as Bambi, and wouldn't engender the same feelings and interactions. There is no generic "friends" character relationship. Every one is different.

2.6 Consistency of Expression

Consistency of expression is one of the basic requirements for believability. Every character or agent has many avenues of expression depending on the medium in which it is expressed, for example an actor has facial expression, body posture, movement, voice intonation, etc.. To be believable at every moment all of those avenues of expression must work together to convey the unified message that is appropriate for the personality, feelings, situation, thinking, etc. of the character. Breaking this consistency, even for a moment, causes the suspension of disbelief to be lost.

Stanislavski suggests that this is one of the ultimate aims of a good actor, and one of the purposes of his method of study. He proposes to get consistency by an actor developing and exercising an internal "sense of truth":

> It is only when [an actor's] sense of truth is thoroughly developed that he will reach the point when *every pose, every gesture will have an inner justification*, which is to say they express the state of the person he is portraying and do not merely serve the purposes of external beauty, as all sorts of conventional gestures and poses do. [Stanislavski 1968, p. 189, emphasis in original]

The danger when this requirement is not fulfilled is well known by those who attempt to bring characters to life. Thomas and Johnston describe its effect in animation:

> Any expression will be weakened greatly if it is limited only to the face, and it can be completely nullified if the body or shoulder attitude is in any way contradictory. [Thomas and Johnston 1981, p. 444]

Angna Enters, an American mime, relates its effect in mime through an incident of a performance she witnessed in Paris:

> A remarkable and noted mime did not receive an expected laugh; instead of continuing, he broke the line of his character by turning his head slightly more toward the audience, repeated the smile, exaggerated, which turned it into a strained grimace. The audience laughed — but *at* the performer, not *with* the character. The thread was broken, and what followed [of his performance] was lost for a full minute before this personal injection was forgotten. [Enters 1965, p. 37]

This requirement is subtle; it certainly does not preclude the various types of subtle expression that people and characters engage in. As Hitchcock explains: "People don't always express their inner thoughts to one another; a conversation may be quite trivial, but often the eyes will reveal what a person really thinks or feels" (in [Thomas and

Johnston 1981, p. 470]). In these instances, this subtle communication *is precisely what is appropriate to be expressed*, and the actor, animated character, or believable agent had better be consistent in that expression.

For the meaning of dialogue, some contend that this is what always happens in communication, for example Angna Enters claims: "The way a character walks, stands, sits, listens — all reveal the meaning of his words" [Enters 1965, p. 42]. The Disney animators extend this idea to a principle for animating expressions and movements to accompany dialogue:

> The expression chosen is *illustrating the thoughts* of the character and *not the words* he is saying; therefore it will remain constant no matter how many words are said. For each single thought, there is one key expression, and while it can change in intensity it will not change in feeling. When the character gets a new thought or has a realization about something during the scene, he will change from one key expression to another, with the timing of the change reflecting what he is thinking. [Thomas and Johnston 1981, p. 464, emphasis in original]

In all of these cases the avenues of expression must work together because as Thomas and Johnston say: ". . . a cartoon character only lived when the whole drawing, as well as all the parts, reflected the attitude, the personality, the acting, and the feeling of the character" [Thomas and Johnston 1981, p. 110].

2.7 The Illusion of Life: Requirements from Agents

The final requirement I want to mention for believability is actually a collection of requirements. As evidenced by the list of requirements above, artists have strong insight into a wide range of properties that are needed for believability. But, some requirements that are needed for believable agents are overlooked by them. This may be because these requirements are taken care of by the domain in which they work. A playwright, for example, does not have to specify that his characters should be able to speak while walking because all human actors that interpret the script can talk at the same time they walk. Even in artistic domains where the artist is creating the believable character from scratch — for example, in animation and literature — simple properties such as performing actions at the same time are taken for granted. When trying to construct believable agents, this is a luxury that we do not have. If we want these agents to have the basic facilities that every actor, animated character, or nearly every living creature, has then we must build those properties in our agents. In this section I list such properties. I think you will find that they are not a surprising or contentious list, although they do provide challenges for those creating autonomous agent architectures for believable agents.

2.7.1 Appearance of Goals

All characters in the arts and nearly all creatures in the world appear to have goals. If we want our agents to be as believable, they need to also appear to have goals. As I will describe in the coming chapters, I believe one promising path to achieve this is to use explicit representations of goals in the architecture. Authors can express what goals their agent has, how they arise, and create behaviors for the agent to express those goals as appropriate for the personality being constructed.

2.7.2 Concurrent Pursuit of Goals and Parallel Action

Creatures we normally see, whether in everyday life or characters from the arts, perform multiple activities at the same time. Dogs wag their tails, and move their ears, heads and eyes while they are barking to get your attention. People routinely do things like stirring the pasta sauce on the stove at the same time they are watching television. As they do these they are performing a myriad of activities: understanding what the actors are doing and saying, having emotional reactions to the story, monitoring the consistency of the sauce, testing its taste from time to time, etc. Some activities are truly performed in parallel while others are done concurrently with a mix of interleaving of steps and parallel or overlapping action. If we want our agents to have convincing behavior, they also need to be able to perform multiple actions and pursue multiple higher level activities concurrently.

2.7.3 Reactive and Responsive

Characters in the arts are reactive to changes in their world. It is hard to imagine a character that could be believable that never reacted to a speeding car about to hit him, someone yelling his name, or someone opening the door for him as he was reaching for the handle. Certainly believable characters have been created that don't react to some events such as these; characters based on the stereotype of an absent-minded professor might require being non-reactive to some types of events. Even such specialized characters, however, have a base level of reactivity, and for the broad range of characters reactivity is an ever-present aspect of their behavior.

It is not enough to have this ability to be reactive. These reactions must be at speeds that are reasonable. Even if everything else is perfect, a character would not be believable if its responses were always delayed by a minute. One could also imagine breakdowns in believability if the responses were too fast. The responsiveness of the agent must be within the ranges people are willing to accept as believable. The ranges used by existing characters, creatures in the world and people are a good starting point. More specifically, the particular responsiveness in a situation should be appropriate to the character's personality. Reactions to a greeting can be slow if the character is tired or a contemplative personality, but they should be faster if the character is an energetic, spirited child.

2.7.4 Situated

Characters in the arts appear situated in the sense described by Agre and Chapman [Agre and Chapman 1990]: they change what they are doing and how they do it in response to the unfolding situation. This is clearly a basic need for believable agents.

2.7.5 Resource Bounded — Body and Mind

Characters in the arts seem resource bounded. They seem to have limits both on how much they can think about and in what they are physically capable of. These limitations are, of course, specific to the character. A genius is often portrayed, in part, by the character appearing to think very quickly, and Superman can physically do more than normal people. Nevertheless, all characters, even these, have limits on what they are capable of.

Typically, in autonomous agents and other aspects of artificial intelligence,[4] researchers do not place limits on what can be done. They try to get the agents to be as smart and capable as possible. For believable agents, appropriate limits both mentally and physically are needed.

2.7.6 Exist in a Social Context

Every character in the arts exists in a social context. Whether that context is the literary bookstore and lower east side New York of the film *Crossing Delancey* [Silver 1988], the stylized and hard-edged San Francisco of *The Maltese Falcon* [Hammett 1957; Huston 1941], the 16th century feudal world of *The Seven Samurai* [Kurosawa 1954], or the magical kingdom of *Aladdin* [Clements and Musker 1992], the characters understand the social conventions and other aspects of the culture and world in which they exist.

Believable agents must also be situated in the culture, social conventions and other aspects of the world in which they are to exist.

2.7.7 Broadly Capable

Characters in the arts are broadly capable. They seem to act, think, sense, talk, listen, understand, have emotions, exist in dynamic worlds, etc. Technology for believable agents need to be similarly broadly capable if it is to support the believable agents that people want to build.

2.7.8 Well Integrated (Capabilities and Behaviors)

In autonomous agents an observer can often see the boundaries between capabilities of the agent. One can tell when the agent is sensing the environment, when it is generating natural language, and when it is acting in the world. This is often because these parts of the agent are distinct modules and the external behavior of the agent radically changes when it is performing one versus another of these functions. It is not uncommon for an agent to stop moving entirely when planning a natural language utterance or when processing sensory data. Other more subtle signals of the separation of the functions are also common, for example the agent appearing to have different knowledge of the world when speaking than when acting.

[4]Cognitive Science research is an exception to this.

In characters this is not the case. Characters' capabilities seem to be seamlessly integrated, with the character freely combining them for whatever it is currently doing. For believable agents to seem alive their capabilities need to appear as well integrated.

Similarly, autonomous agents often abruptly switch between behaviors, or otherwise have distinct observable behavior boundaries. For example, a traditional robot often returns to a particular "rest" configuration between each of its behaviors. This allows the behaviors to be written more easily because they always start and end at the same position. Sometimes the distinctness of behaviors is apparent because of time delays between them, or because of the abruptness of the switch.

Characters do not always have such distinct observable behavior boundaries. It is much harder to unambiguously divide a character's stream of activity into distinct behaviors. They smoothly move from one activity to the next, often overlap portions of behaviors, and have appropriate transitions between behaviors when distinct transitions are appropriate. The behaviors of believable agents must be similarly well integrated.

2.8 Summary

I have described in this chapter requirements for believable agents. These requirements were developed from my study of the traditional character-based arts as well as attempts to create interactive, autonomous versions of these characters.

I will be referring back to these requirements repeatedly throughout the thesis, so I list them again in Figure 2.1 for easy reference.

- **Personality**
- **Emotion**
- **Self Motivation**
- **Change**
- **Social relationships**
- **Consistency**
- **Illusion of Life**
  - **Appearance of Goals**
  - **Concurrent pursuit of Goals**
  - **Parallel Action**
  - **Reactive and Responsive**
  - **Situated**
  - **Resource Bounded – body and mind**
  - **Exist in a Social context**
  - **Broadly Capable**
  - **Well integrated (capabilities and behaviors)**

FIGURE 2.1: Requirements for Believability.

Chapter 3

Example Domain

To make concrete the problem to be solved, I think it is valuable to describe the types of basic building blocks my agents will be built from: what actions will they be able to perform; what events will they be able to perceive; and in general, what kind of world will they exist in. Building a believable agent with very high-level primitive actions is different that building one which must directly control motor impulses.[1]

Because one of the goals of my work is to build agents that are believable in real-time, animated, interactive worlds, I will describe one such world here. My work has also been used to build agents in less continuous worlds with a text interface, and more physically-based worlds based on a mass-spring simulation model. A description of an agent from the text-interface world is described in [Bates *et al.* 1994].

The world described in this section was originally designed for a system called *Edge of Intention* and known informally as the Woggles. I have since modified it slightly from its original form to allow for additional exploration; one modification of note is the addition of text bubbles to allow the agents to perform speaking acts. The use of these text bubbles to allow agents to "talk" is described in Chapter 8. Edge of Intention was built in collaboration with about a dozen people from the CMU Oz and Graphics groups and others. It was first shown in the Arts Exhibition at the 1992 American Association for Artificial Intelligence (AAAI) conference. The world with two agents is shown in Figure 3.1.[2] Our goal in creating this world was to create a simple world in which we could test and showcase our believable agent work. We chose to make the world and physical bodies of the agents simple, and we intended for the complexity and richness in the system to arise from the movement, behavior and interactions of the agents with each other and with human participants. The artistic style of the world is intended to be reminiscent of the worlds created by Dr. Seuss in his popular children's books [Seuss 1975].

Technically, the world is modeled as a surface over a rectangularly bounded plane.

[1]This is a more subtle trade-off than it appears at first glance. Higher-level primitive actions might make it easier to construct an agent, but might make it harder to accomplish the requirement of **personality** because personality-based variation might be needed within the high-level actions.

[2]The original version includes four agents; three with distinct personalities and one is controlled by a human interactor. The current version allows the system to be run with any subset of the four agents.

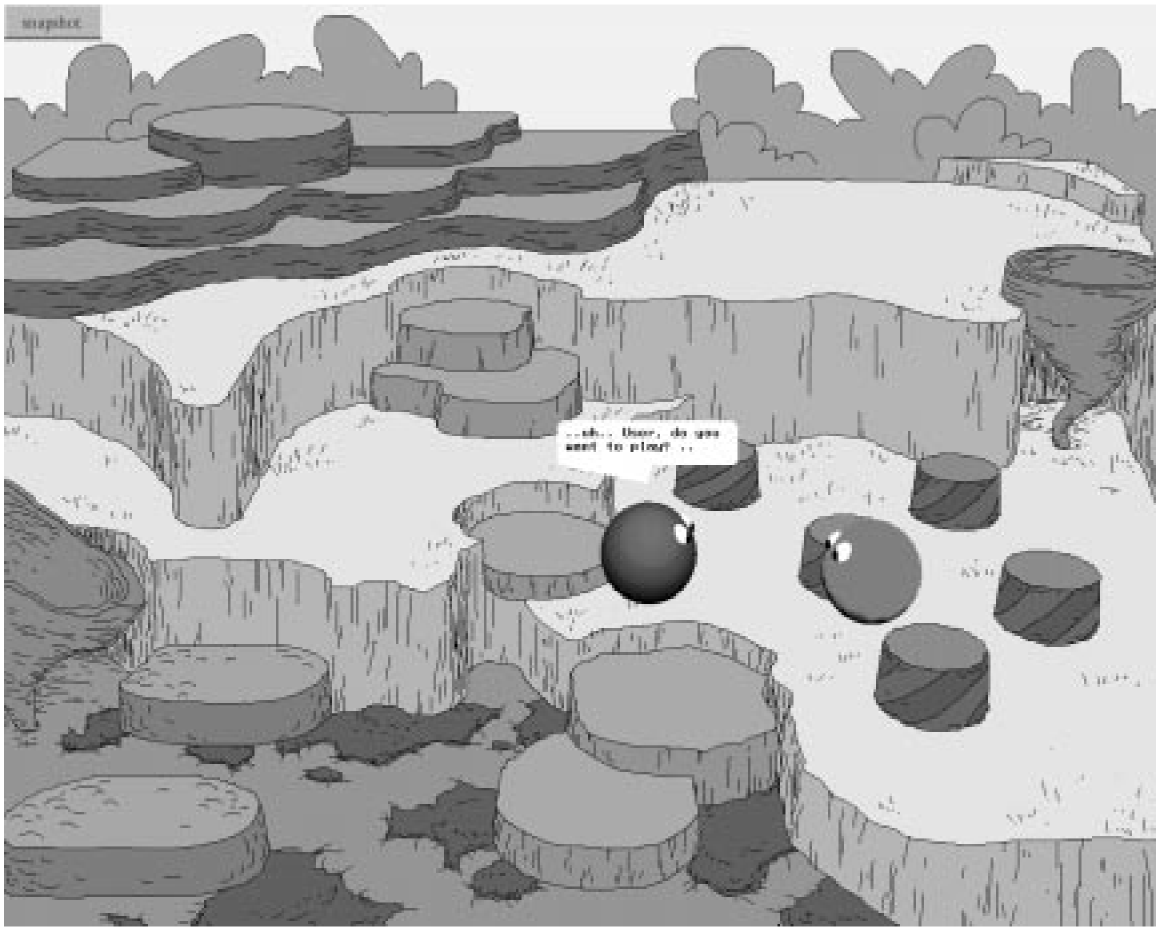


FIGURE 3.1: The *Edge of Intention* world

There is a fixed camera displaying the world on the screen, and nearly all of the world is visible from this vantage point. The surface is defined by discretely partitioning the bounded plane into a 1000 by 1000 grid, and specifying a height for every discrete point. This surface is painted for our aesthetic benefit, but the agents in the world do not see the colors. They can only sense the shape of the surface. The bushes, grass, and trees are all painted. The funnel shapes in the upper right and lower left corners are actually modeled as cylinders with the funnel appearance painted.

These two funnel shapes are part of the “chute”, which is the only mechanism in the world. It is intended to be similar to a slide in a playground or the hill for downhill skiing. It is a tube in the world that goes underground. The visible funnel shapes are the ends of the tube. The Woggles can jump in the end on the right and they will slide out the other end. The output end is pointed at the input end, and if they have enough speed when they jump in, they are able to slide around twice from one jump.

The bodies of the agents are likewise simple. A body is modeled as an ellipsoid that can squash, stretch and have a shear transformation performed on it. Each body ellipsoid has a rest shape that it returns to when it is not being actively controlled. It oscillates from the position it is in when control is released toward the rest position, damping the oscillation each cycle.

Every body also has two eyes that are each composed of a white and a black sphere. The eyes are placed in fixed positions on the body, and can be controlled by moving the black sphere's (pupil's) location within constraints that maintain the appearance of eyes.

3.1 Actions

Each body is controlled by a stream of often overlapping actions that are issued by the mind. The actions themselves are fairly low-level. Each causes between .2 and 1.5 seconds of action. Multiple actions can be executed in sequence to create movements with function and meaning such as going to a point in the world, greeting another agent, sleeping, etc. Often actions will be executed simultaneously or partially overlapping, for example a Woggle might move its eyes slightly before, but overlapping with, turning its body to give some anticipation to the turn. Uses of these actions to create meaningful behavior are given in Sections 4.15, 7.8 and 8.5.

An action is specified by a name and zero or more parameters. In the rest of this section, I describe each action in turn.

Jump and **Put** actions physically move the body from one three dimensional point to another. As suggested by the name, a **Jump** causes the body to jump from its current point to its target point. The body attempts to perform this jump with a duration specified in the **Jump** action, and normally achieves this to within a few milliseconds. The duration of a jump can be anywhere from .2 seconds to 1.5 seconds. A **Put** causes the body to slide to the target point at a speed necessary to arrive there within the given duration. This slide is a linear interpolation of the body, unless a squash or normal is specified. In this case the effect of a **SquashDir** (described below) is performed at the same time the Woggle's base is linearly moved to the target point.

In addition to moving around the world using jumps and puts, a body can squash in a number of ways. A **Squash** action causes the body to compress or expand along its vertical axis. This action adjusts the other dimensions of the body to preserve its volume. A **SquashDir** action does the same thing as a squash but in addition performs a shear transformation. The direction of the shear is given as a vector argument to the action. For both of these actions a duration argument is included. This duration specifies how long it takes to transform from the current shape to the specified shape. The transformation is linear.

When either of these actions is completed, the body oscillates back to its rest position unless another squash, jump or put action immediately follows it. To override this return to rest position, there are **SquashHold** and **SquashDirHold** actions. These variations are exactly the same as the previously described **Squash** and **SquashDir** actions except that the target shape is maintained after the action finishes. To return to the rest position from one of these actions a **SquashRelax** action must be issued.

The direction the body is facing can be changed by a **Spin** or **SpinTo** action. Both of these actions take an angle and a duration. Spin turns the given angle relative to its current heading in the specified duration. **SpinTo** turns to the given absolute angle from its current heading in the specified direction.

The direction the eyes are facing is controlled by similar actions: **SpinEyes** and **SpinEyesTo**. These actions take the same angle and duration arguments, and turn the eyes in the same way the body is turned by the **Spin** and **SpinTo** actions. The elevation of the eyes is controlled by the actions **ElevateEyes** and **ElevateEyesTo** which move the eye elevation by a relative or absolute angle respectively within the given time duration. If any of these actions would cause the eye to move outside its physical range, its movement is stopped at the boundary of that physical range.

Actions also exist that cause the eyes and direction of the body to track objects or points. Tracking actions are not given a duration. The tracking behavior specified by these actions remains active between the appropriate start action and stop action. The direction the body is facing can track a point or other agent's eyes using **StartFacePoint** and **StartFaceWoggleEyes** respectively. The eyes can do the same using **StartLookPoint** and **StartLookWoggleEyes**. The action **StartTrackWoggleEyes** is identical to issuing both **StartFaceWoggleEyes** and **StartLookWoggleEyes** with the same agent as the argument. The tracking actions for the body heading and eye direction are ended using **StopFace** and **StopLook** actions.

An agent can "speak" by issuing text strings to appear in a speech bubble above the agent's head. An image of such a bubble is shown in Figure 3.1. These strings appear at the next available position in the text bubble at the time the **Say** action is issued. This is normally directly following the last text issued. Text in speech bubbles persists for a time, and is removed when this time expires. If there is no text in the speech bubble the bubble is removed. Dots mark pauses between issued text. If an agent continuously talks, no dots are inserted. But every .7 seconds an agent pauses between **Say** actions, a '.' character is placed in the speech bubble. This gives a visual reference for pauses when an agent is talking. These ellipses are not considered when removing expired text or speech bubbles.

The radii of the body ellipsoid can be changed by a **ChangeBodyRadii** action. The new radii and duration over which to make the change are provided as arguments to the action. The color for a body can be similarly changed by specifying new hue, saturation, value, and duration arguments to a **ChangeColor** action.

Closing and opening a Woggle's eyes is accomplished by issuing **CloseEyes** and **OpenEyes** actions. These actions take one frame each to execute. (The system normally runs at 10 or more frames a second, so a frame is approximately a tenth of a second.)

A Woggle body can be caused to tremble by issuing a **StartTremble** action. Issuing a **StopTremble** causes it to stop.

3.2 Perceiving the World

The agents can perceive the world at a low level. To perceive another agent's behavior, an agent can perceive the physical properties of the agent's body and the actions that the agent is performing at the current instant in time. The physical properties that are perceived are physical location, angle the body is facing, and direction the eyes are facing. In addition, any of the actions described in the previous section can be perceived when they are executing.

This low level of perception must be interpreted by the mind of the agent in order for the agent to react appropriately to relatively large scale events like fights, sleeping, moping, etc. For example, a threatening behavior might be recognized because an agent is close to another while performing multiple quick body motion actions (squash, jump, put, ChangeBodyRadii). Of course, how patterns of perception are interpreted is subject to the social conventions of the created world, as well as the individual personality of the agent.

The agents perceive the physical environment by querying positions in the XY plane for associated heights. This allows agents, for example, to detect the edges of the world's flat regions.

In the particular agents we have constructed in this world, each agent also has an internal notion of "areas" in this world. Areas are defined by a center point and two radii that describes the largest circle that fits within the area and the smallest circle that includes the area. Each area may have multiple social or personal meanings for an agent, for example areas to sleep, areas to dance, etc.

3.3 Interactive Participation

A human participant can interact with this world by controlling one creature using a mouse and keyboard. The user-controlled agent can perform many of the actions that the autonomously controlled agents can perform, although not all. The person can cause **Jump** and **Put** actions by clicking on the point to be moved to. If the target point is near, a **Put** is used; if it is far, a **Jump** is performed. Using another mouse button, the user can control what the agent looks at. The agent will track the point or Woggle clicked on by executing **StartFacePoint** and **StartLookPoint** actions or a **StartTrackWoggleEyes** action. The user-controlled agent "says" in its speech bubble whatever the user types on the keyboard by issuing **Say** actions in response to the typing.[3] All of these actions are issued in real-time as the user presses mouse buttons or types character by character.

In addition the user can cause the User Woggle to do two different stylized motion gestures that have social meaning in this world. The first is a squash up and down, which we refer to as "hey", that in different contexts is interpreted as "hi", "bye", "want to play", etc. The second motion is a shape change that was inspired by the threatening behavior of cobras and other animals. When signaled by the mouse the user-controlled Woggle turns to

[3]The ability to control the User Woggle's eye movements and the ability to "say" things in the User Woggle's voice bubble are both extensions to the original Woggle domain. I added them both to explore natural language as described in Chapter 8.

face the nearest Woggle while shrinking its body radius in one direction and increasing its radius in the other two, to appear bigger to the Woggle it is facing. We refer to this second movement as a “puff” and an image of it is given in Figure 5.3.

3.4 The Task

The task facing an author is how to construct one or more agents that use the available perceptions of the world to produce series of potentially overlapping actions that make them seem believable when interacting with other autonomous agents and human-controlled agents.

Chapter 4

Hap

Hap is my agent architecture for believable agents. I developed Hap to try to satisfy three complementary, but distinct, goals. First of all, it is designed as a language for expressing detailed, interactive, personality-rich behavior. Ultimately, I want to allow people to build believable agents that are as rich in detail as characters from the traditional arts. This is directly in response to the first requirement for believable agents — personality — described in Chapter 2. An animator, for example, specifies a wealth of details to give life to his character. Each character has its own way of walking, its own mannerisms, its own desires. In fact it has been claimed that no two characters should do the same thing in the same way [Thomas and Johnston 1981, p. 135]. To truly enable autonomous agents that seem alive with similar richness of personality, these personality-specific details need to be specified. In addition, since the character must be interactive, additional knowledge — such as when to perform such actions, what to react to, what motivates this character — must also be specified. My first goal for Hap is that it be a language for encoding such knowledge. That means that it must be possible to encode interactive versions of personality-specific motions, as well as behaviors for when and how those motions arise, and general reactions, motivations, etc.

Central to this goal is my desire to allow artists or people with artistic vision of a particular desired personality to express that vision in autonomous form. This direct expression is in contrast to artificial life and some autonomous agent research that strives to have the richness of the agent *emerge*. I want to give the creators of believable agents direct control over the details and richness of their creations, just as traditional animators have control over the details of the characters they bring to life. It is also possible that allowing artists to directly craft the behavior of their agents is one of the best paths to high quality, artistically rich believable agents.

The second goal in designing Hap is for it to be an agent architecture that directly addresses some of the requirements for believable agents in real-time, visual worlds. I want agents that seem aware, react to their environment, pursue parallel actions, seem emotional, etc. Many of these requirements for believability (listed more completely in Chapter 2) can be addressed at least in part by the architecture. By doing this I make the job of the character builder easier, since she[1] has a framework in which to express herself. Of course, it is still necessary for her to impart the content to satisfy these requirements. Only she knows what the agent should be emotional about, and to what it should react to be true to its personality.

The third goal for Hap is that it be a unified architecture in which to express all of the processing aspects of an agent. An author writes behaviors for action, emotional processing, sensing, inference, and natural language generation in the same language. By expressing all of these in Hap there are two advantages. First by being built in Hap, these capabilities inherit some of the properties of Hap that are important for believability. This includes both architectural properties that address specific requirements for believability, as well as the general ability to express personality-specific variation within these processes. The second advantage is the ability to combine these capabilities for the purposes of the agent as a whole. At a high level, an agent could be simply pursuing two independent goals as part of a single behavior, for example walking while talking as part of a social behavior. At a lower level, a goal that is thought of as predominately in one capability could realize a subgoal using another capability. For example, a natural language generation goal will very often use physical action to accentuate what is being communicated or to carry out some of the communication. Inference and perception goals and behaviors are almost always used as part of other behaviors, and language generation often arises as part of other activity the agent is involved with. Hap facilitates all of these types of combinations, as behaviors can be written that combine goals of any type, and the goals themselves can be carried out by complex behaviors that use combinations of goals and primitive actions of any type.

Through the detailed description of the Hap architecture in the following sections and the description of capabilities built in Hap in Chapters 5, 6 and 8 some of the ways Hap addresses these goals will become clear. Each goal will be addressed again in detail in Chapter 9.

4.1 Goals, Behaviors and Primitive Actions

An author encodes an agent's behavior in Hap by writing *goals* and *behaviors*. A goal is an atomic name and zero or more values, for example `(wait_for 100)`, `(stop_fight Shrimp Wolf)`, and `(amuse_self)` are all potential goals. Unlike some notions of goals in artificial intelligence, Hap goals have no meaning in and of themselves. They are not planned for, and there is no grammar or logic in which the system can reason about them. Instead they are given meaning through the agent builder's vision of what they mean, and the behaviors written to express that vision. The initial set of goals for the agent is specified by the author.

Behaviors are written to describe the activity the agent can engage in. Each behavior is written for goals with a particular name, and can be pursued by the agent when a goal with that name is active. Behaviors primarily provide an organization of *steps* that together

[1] In this document, I alternate using "he" and "she" when referring to a person in the generic sense (without a specified gender).

describe an activity to accomplish that goal. Goals and primitive actions are both steps. For example, a simple behavior for blinking is to perform the actions **CloseEyes** and **OpenEyes** in sequence. More typically a behavior includes goals which must be elaborated to fully determine the activity. For example, a behavior for the `stop_fight` goal above is to physically move between the two who are fighting, and tell them to stop fighting. This would be expressed as a parallel organization of two goals. The complete meaning of the behavior is determined by how these goals are elaborated at runtime with their own behaviors.

Multiple behaviors can be written for the same goal. For example, a single behavior for an agent's `amuse_self` goal would not make sense for most personalities. Most creatures have multiple ways of amusing themselves that they pursue at different times. Likewise, the behavior above for stopping a fight might not be appropriate for all situations in which an agent has the `stop_fight` goal. Behaviors for this goal that are appropriate to various situations can be written. Hap chooses among these behaviors at runtime using information about appropriate situations specified by the author.

In Hap, the complete activity a given agent can engage in is expressed as a fixed set of behaviors written by the author.

Behaviors and goals are grounded in primitive actions. Primitive actions are of two types, physical actions and mental actions. Primitive physical actions are the set of actions the body is directly capable of, for example jump to a point in the world or spin some number of degrees. They differ depending on what type of body Hap is connected to. A complete list of physical actions for one world is given in the previous chapter. Mental actions are code that can be directly executed; they are written in an enhanced version of C. By policy, mental actions are small, bounded computations.[2]

The initial goals, the behaviors, the subgoals they give rise to, their behaviors, etc. are all written by an author, and comprise a large part of the personality of the agent. These goals and behaviors are used to encode personality at many levels: from motivation to the details of locomotion and eye movement.

4.2 Active Behavior Tree

At any point in time, all of the goals and behaviors an agent is pursuing are stored in the *active behavior tree (ABT)*. The ABT is the main processing data structure in a Hap agent. Initially this tree contains a root with children that are the initial goals of the agent.

Hap's basic execution is to hierarchically expanding the ABT at runtime. This expansion eventually results in a stream of actions sent to the body to be executed. Hap expands the tree by repeatedly choosing the best leaf node, and executing it. The mechanism for choosing the best node is described in the discussion of the step arbiter in Section 4.12 below.

[2]If this policy is violated, Hap will still operate but reactivity, responsiveness and its real-time properties are compromised. Complex computations are instead created using multiple mental actions and Hap's other language constructs. See Section 5.1 for discussion of this issue.

The method for executing a step depends on the type of step: primitive physical action, primitive mental action, or subgoal. Physical actions are executed by sending them to the body to be executed. Mental actions are executed by running the associated code. A goal node is executed by choosing a behavior that is appropriate for pursuing it in the current situation. This behavior is then instantiated and written in the tree as the child of the goal node. The steps of the behavior are written in the tree as children of the behavior node. These steps are available to be chosen in future execution cycles along with the other leaf nodes of the tree.

The tree contracts as goals, actions and behaviors succeed, fail or are aborted.

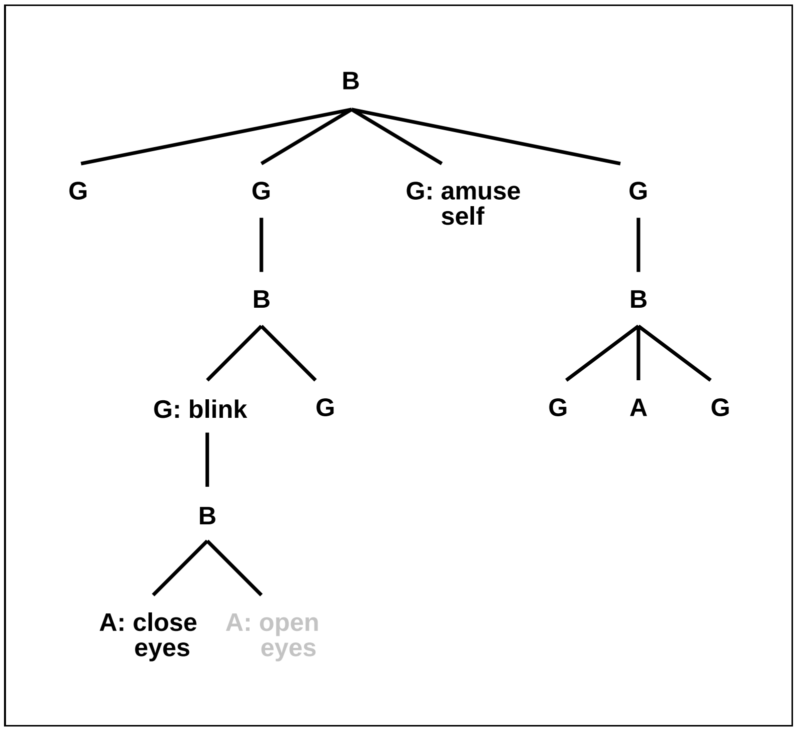


FIGURE 4.1: Example Active Behavior Tree

An example active behavior tree is shown in Figure 4.1. “G” ’s in the figure are goals, “B” ’s are behaviors and “A” ’s are primitive actions. The first level of the ABT are all of the top-level goals of an agent. Each behavior is in pursuit of its parent goal in the tree and the children of the behavior are its component steps. At each execution cycle one of the leaf steps is chosen and executed. In this example tree, the amuse self goal, close eyes action, or one of the unspecified four leaf goals or unspecified leaf action are available to be chosen next. Some behaviors impose orderings as shown by the behavior for the blink goal in this tree. The gray steps are not available to be chosen because of the order imposed by the behavior. This is described in Section 4.6.

4.3 Success and Failure of Goals and Behaviors

Success and failure for goals and behaviors are defined recursively, with the success or failure of primitive actions as the base cases. Primitive mental actions always succeed when the code has been executed. Primitive physical actions succeed or fail based on how the body performs the action in the world.

A goal succeeds when a behavior chosen for it succeeds. For example, the `amuse_self` goal mentioned earlier might succeed when a behavior to explore the world is chosen for it and the exploration completes successfully. As mentioned earlier, most agents have multiple ways of amusing themselves and this is reflected in multiple behaviors for the `amuse_self` goal. Thus there are many way for the goal to succeed.

A goal fails when it is chosen to be executed, and no behaviors apply for it. This could occur because there are no behaviors for this goal that are appropriate in the current state of the world. It can also occur because all of the applicable behaviors have been tried and failed. Each behavior is only attempted once for a particular goal. If an author wants a behavior to be attempted multiple (but finite) times for a goal, he can make multiple copies of the behavior. This would be desirable for a behavior that turns a key to start a car, for example.

Behaviors fail when any step fails. For example, if an agent is pursuing a behavior to go to work by car and the car doesn't start, the behavior fails. Any other steps in the behavior are then aborted, and another behavior may then be chosen to pursue the goal to go to work. Perhaps, in this case, a behavior to take the bus might be chosen.

This definition for the failure of a behavior may seem overly restrictive, as one can imagine optional steps that could be part of many behaviors. As we shall see later, this notion is expressible within Hap, but it is important to understand the basic definitions before this and other relaxations of these definitions are explored.

A behavior succeeds when all its steps succeed. No explicit test for success is necessary for a behavior and the goal it is in service to to succeed. As I discuss below in Section 4.3.1, this seems appropriate for much of a believable agent's routine behavior. If an explicit test is desired for the behavior to succeed, one can be written (see Section 4.14.2).

Whenever an action or goal succeeds or fails, the active behavior tree is modified by the cascading effects of that success or failure. These effects are described in pseudo code in Figure 4.2. These effects start whenever a mental action succeeds, a physical action succeeds or fails, or a chosen leaf goal fails because no behaviors are applicable for it. At that point, `succeed_step` or `fail_step` is called on that action or goal. `Succeed_step` performs the modifications to the ABT that are necessary whenever a goal or primitive action succeeds: that step is removed, and the parent behavior succeeds if there are no more steps to execute. `Fail_step` performs the modifications to the ABT that are necessary whenever a goal or primitive action fails: cause the parent behavior to fail. When a behavior succeeds, `succeed_behavior` causes the goal that behavior was in service to (its parent goal) to succeed. When a behavior fails, the fact that it failed is recorded in its parent goal. This information is used to ensure that each behavior is attempted at most once for

```
define succeed_step(S)
  parent := parent_behavior(S)
  remove(S)
  if ( parent has no children ) then
    succeed_behavior(parent)

define fail_step(S)
  fail_behavior(parent_behavior(S))

define succeed_behavior(B)
  succeed_step(parent_goal(B))

define fail_behavior(B)
  add behavior to list of failed behaviors in parent goal
  remove(B)

define remove(Node)
  removes the node and its subtree from the ABT.
  Any executing actions in the subtree are aborted.
```

FIGURE 4.2: Basic cascading effects of success and failure of behaviors, goals and actions.

a given goal. The failed behavior is then removed, along with its subtree if it has one, allowing another behavior to be chosen for this goal in later execution cycles.

When a node is removed from the tree, all of its children are also removed. In some cases this results in executing actions being removed from the tree. In this case, these actions are aborted.

4.3.1 Discussion: Routine Activity and Tests for Success

Not requiring an explicit proposition to be true for a behavior (and therefore the goal the behavior is in service to) to succeed is an unusual feature of action architectures like Hap that deserves further comment. I chose this feature because of my belief that it is difficult to create a complete and accurate domain model of the world and that such specifications are unnecessary for much of the behavior of a believable agent. To illustrate these ideas consider this example:

> Peter has the goal to communicate to his friend Jim in another city. He tries telephoning, but gets no answer. He then uses a behavior to write and send a letter. He finishes the behavior by putting his letter in a mailbox, and goes on with his day.

In this example, how does Peter know that his goal was satisfied? He could wait until he finds out definitively that his message was received (by way of a reply or return receipt from the post office, for example). As a more reasonable if less correct approximation, he could have some model about the reliability of the mail system: that a message written in a letter, and placed in an envelope which has been addressed to the appropriate person satisfies the goal of conveying the message to that person. But this expression is simply a reiteration of the behavior and the fact that each step was successfully finished. So, rather than require that the criterion for each goal's success be expressed in a predicate logic expression, Hap allows the completion of a behavior for a particular goal be the criteria for success.

Similarly one could argue that some behaviors are not intended to make any condition true. For many personalities, the goal to "dance" succeeds precisely because the behavior of dancing was performed.

I believe that much of routine behavior has a similar quality to these examples in that completing a behavior for the goal is an appropriate criteria for success. The traditional approach, that a goal's success is determined solely by the truth or falsity of a predicate, can be realized by an appropriate encoding (as described in Section 4.14.2), although this feature has rarely been used in the agents constructed in Hap.

4.4 Situated Variation

An author can write multiple behaviors for a given goal, to capture the different ways the agent pursues that goal. When any instance of the given goal is chosen for execution, Hap chooses from these behaviors by attempting to choose the best behavior at that moment. To inform this decision an author can write *preconditions* and *specificities* for behaviors.

The author can write a *precondition* for each behavior to specify under what circumstances the agent might choose this behavior to carry out a chosen goal of the given type. If a precondition is false, the behavior will not be chosen. Preconditions are boolean-valued expressions which can include references to the goal (including values embedded within the goal), the internal state of the agent (including the agent's emotional state, set of active goals, etc.), and properties of the external world. In addition to yielding a value, a precondition can bind variables to values for later use in the body of the behavior.

Behaviors can be labeled as more *specific* than others by writing a numeric *specificity*. If no specificity is given, the default is zero. This allows specialized and general-purpose versions of a behavior to be built and for this relationship to be indicated. For example, an agent might have a normal behavior for starting her and most other cars. For her husband's temperamental car that has electrical problems, the agent might have a more specific behavior that includes pulling a fuse before turning the key so the brake lights aren't shorted out. Except for pulling the fuse and reinserting it, the behavior might be the same as the less-specific one that is used in other circumstances. In addition to a higher specificity value, such a behavior would have an appropriate precondition to only apply to the particular car as well. Specificities need not order all behaviors for all goals. They are only needed to order behaviors for a given goal, and of those only those behaviors whose preconditions can be true at the same time, and for which the author wants to specify a preference.

Whenever a goal is chosen to be executed, Hap must choose a behavior for it. (When none can be chosen, the goal fails.) The chosen behavior and component steps are placed in the active behavior tree as the subtree of the goal. The steps are then available to be chosen for execution, perhaps being expanded with their own behaviors in turn, and eventually resulting in actions being executed. If the behavior chosen for this goal fails for any reason, the behavior is removed, and the goal is again available for execution, allowing a new behavior to be chosen to pursue it. This can result in Hap pursuing multiple behaviors in turn for the same goal. If one views behaviors as a type of plan this activity can be viewed as a type of backtracking search in the real world. The difference from classical backtracking search is that the world may have changed between the time when one behavior is aborted and another is chosen. This change could be the result of actions the aborted behavior performed or simply because time has elapsed. Because of changes in the situation, the set of behaviors that are appropriate in the current situation may be different than those available previously.

Whenever a goal is chosen to be executed, Hap chooses a behavior for the goal by the method presented in Figure 4.3. Hap only chooses among behaviors that have not failed already for this goal instance, ensuring that each behavior is attempted at most once for a given goal instance.[3] The precondition must be true in the current situation in order for a behavior to be chosen. This ensures that the author thought the behavior appropriate for the agent in this situation. Of the unfailed behaviors that apply in the current situation, those with higher specificity values are attempted first. If there are multiple behaviors with the highest specificity values that have not failed for this goal and that apply in the current situation, Hap chooses randomly among them.

1. If a behavior has failed previously for this goal instance, it is not available to be chosen.
2. Behaviors with false preconditions are not available to be chosen.
3. More specific behaviors are preferred over less specific ones.
4. If multiple behaviors remain, Hap chooses randomly among them.

FIGURE 4.3: Behavior Arbiter – Method for choosing among multiple behaviors for a goal.

[3]As mentioned previously, if an author wants a behavior to be attempted multiple (but finite) times, this is accomplished by including multiple copies of the behavior. With appropriate preconditions this allows behaviors to be intelligently repeated when the situation seems appropriately favorable. If the author wants infinite repetition, this is accomplished using other annotations, as described in Section 4.8.2.

4.4.1 Discussion: Use of Behaviors, Preconditions and Specificity to Express Personality

It should be noted that the aim in writing these behaviors, preconditions, etc. is slightly different than the aim people have for other similar action architectures. The purpose here is not to write effective behaviors to accomplish the goal with preconditions that optimally choose among them. Rather, the aim is to express the behaviors this character engages in when it has this goal, and how it chooses among them. Expressions of arbitrary preference, such as preferring chocolate over strawberry, and other forms of character-specific details are completely appropriate uses of preconditions and specificity when building believable agents and should be encoded.

To illustrate this difference between my approach and the traditional one, consider the previously mentioned goal of stopping a fight. The traditional approach to writing behaviors[4] for this goal might be to think of what ways would be most effective to stop a fight, for example, physically restraining the combatants vs. telling them to stop vs. getting help to stop the fight. The next step for the agent builder would be to decide in what situations these methods would be effective. For example, physically restraining the combatants only works if they are weaker than you; telling them to stop only works if they can hear you; etc. This approach, carried out in enough detail, could give one an effective set of behaviors for stopping fights, but does not express a personality.

In my approach, one starts with a character in mind, and expresses what that character does when it has a particular goal, and how it chooses among these behaviors. So, using the same `stop_fight` goal, let's assume the character the author is building is a mother of five small children, and that she has a temperament like Rosanne from the television show *Rosanne*. The behaviors, preconditions and specificities need to express not what would be most effective at accomplishing the goal, but rather what most effectively captures the personality the agent builder has in mind. Thus the author might choose the same set of behaviors and add a behavior that threatens the combatants with a beating if they don't stop. The preconditions might be very different, for example there might be no precondition for the behavior to tell them to stop, and the only precondition for the behavior that threatens to beat them if they don't stop might be that the combatants be related to her. These preconditions have nothing (or very little) to do with the situations in which these behaviors are effective at actually stopping the fight. Rather, they capture something about the type of personality being built. This character is simply more likely to yell than to physically act in these situations.

When one views the behaviors, preconditions and specificities as ways of expressing a personality instead of as ways of making the agent's goals most likely to succeed, the job of building these behaviors, preconditions and specificities becomes different. If one is trying to make the most effective behaviors and preconditions for accomplishing the goal, the best decision may depend on information that isn't available to the agent, for example, the relative strengths of the combatants vs. the agent's own strength. This is part

[4]Similar architectures may use other names such as plans or plan schemas, but the concept remains the same.

of an increasingly studied problem in AI of planning with uncertainty. When the aim is to express a desired personality, these decisions often only require what is known to the agent. The mom based on Rosanne chooses to yell rather than to physically act because the ones fighting are her children; this agent doesn't need to know any uncertain information to choose among its behaviors. The fact that yelling may not be very effective at stopping the fight is incidental; yelling is what this personality does when her children are fighting. The aim here is to express the personality. If that means building behaviors that often (or even always) fail, then that is exactly what should be done.

Of course, creating agents that are not effective at any of their tasks (even if true to their personalities) is only interesting for so long. The point is not that believable agents don't need to be concerned with competence, but that competence is secondary to personality. Nearly all personalities (even the Rosanne-based mom) are ultimately competent at some level, but the ways they achieve that competence is individual and expressive of their personality.

4.5 Reactivity

An agent built in the language so far described would be mostly oblivious to changes in the environment. Hap provides two types of reactive annotations to address this need: *success tests* and *context conditions*.

These reactive annotations were designed to support two types of reactivity: recognizing the spontaneous achievement of an active goal or subgoal, and realizing when changes in the world make the pursuit of an active behavior nonsensical. These two types of reactivity are illustrated by this example.

> An agent has the goal of opening a locked door. She has two applicable behaviors: get a key from her purse, unlock the door, and open it; or knock and wait. If, while looking for a key, someone opens the door for her, she should notice that her goal was satisfied and not keep working to accomplish it. In the same scenario, if she were searching in her purse for the key when her mischievous nephew snatched the purse, she should abandon that behavior[5] and try knocking (and perhaps deal with the nephew later).

Optionally associated with each goal instance is a *success test*. The success test is a method for recognizing when a goal's purpose in the enclosing plan is spontaneously achieved (capturing the first type of reactivity above.) Like a precondition, it is a boolean expression that can include references to the values in the goal, the internal state of the agent, and properties of the external world. If at any time this expression is true, then this goal succeeds and is removed from the tree. The subtree rooted at the goal node is also removed in the process since its only purpose was working toward the goal. The condition

[5]A different appropriate response might be to pursue the nephew now as a subgoal to get the key back. I make no claims about whether abandoning a behavior or repairing it is appropriate in a given situation or for a given personality. Hap supports both. Encoding repairs is described in Section 4.8.4.

expressed in a success tests is sufficient, but not necessary, for a goal to succeed. A very unreactive agent might use `false` for most success tests and therefore never recognize when goals are spontaneously achieved.

Optionally associated with each behavior is a *context condition*. This condition is intended to notice situations in which things have changed sufficiently for the behavior to no longer be appropriate. For example, a behavior for an amuse goal that involves playing with a ball no longer makes sense when the ball has been irrevocably lost. This kind of change is recognized by a context condition specific to the behavior. When a context condition becomes false, the associated behavior fails and is removed from the ABT. The subtree rooted at this behavior is also removed and any subsidiary goals, behaviors, or actions are aborted. As when a behavior fails for other reasons, the goal it was in service to remains, and another behavior can be chosen to pursue it.

The condition expressed in a context condition is necessary, but not sufficient, for the behavior to work. A very unaware character might use `true` for most context conditions. This corresponds to not being very aware of the world changing out from under a previously plausible behavior.

The conditions under which a behavior can be started (expressed in Hap as the precondition for a behavior) and the conditions that must remain true for it to continue to be a plausible behavior (a context condition) are not the same concept, and very often the two conditions will be different for the same behavior. For example, consider an agent performing a communication goal. It may have multiple behaviors to choose from to convey the information, such as one that conveys the information in a friendly, pleasant manner, and one that conveys the information in a direct and aggressive manner. The agent might choose the friendly phrasing if it is feeling happy. It might abort that behavior if it became less happy, but it would be desirable in some personalities for the threshold for aborting to be lower than that for deciding in the first place. Having two different conditions to express these two aspects of the behavior of an agent is an important property of Hap for expressing the personality of an agent.

Neither the success tests nor context conditions in the agent need be complete or totally accurate for the agent to function. Hap's other processing properties allow some partial recovery for missing or incomplete conditions. For example, if a context condition is left out, the agent continues performing the behavior. In some cases, a step of the behavior will fail because of the changed context. In these cases, the failure of the behavior (and opportunity to pursue another one for this goal) is delayed, but does occur.

In addition, as in preconditions, the aim in writing context conditions and success tests is not to make behaviors that optimally accomplish the goals of the agent. Rather, it is to allow an author to encode the reactivity of the personality being built. Particular characters only recognize opportunities and contingencies that are appropriate to their personality, and those are the ones that should be expressed in these annotations. As with preconditions, this encoding may be easier in some ways than trying to create behaviors that are most effective at accomplishing the goals. The opportunities that an agent pursues are often the result of recognizable situations at the time the behavior is executing.

4.6 Types of Behaviors

Behaviors organize steps to carry out a goal. Their organization depends on the type of the behavior. There are three types of behaviors in Hap: sequential behavior, concurrent behavior, and collection behavior. The steps of a given behavior are available for execution based on the type of the behavior. In a *sequential behavior* each of the steps is available to be executed only after the preceding steps have finished successfully. (If any step fails the behavior fails, removing itself and any remaining steps as described in the previous section.) A *concurrent behavior* and a *collection behavior* both allow all of their steps to be pursued in any order. As you will see below in Section 4.12 and Section 4.13 this often results in concurrent pursuit of goals and in parallel actions being executed.

The difference between a concurrent behavior and a collection behavior is in how they succeed and fail. All steps of a concurrent behavior must succeed in order for it to succeed, and any steps that fail cause the behavior to fail. This is the same as in sequential behaviors. A collection behavior specifies a collection of steps that should be attempted to accomplish the given goal. Whether they succeed or fail is not important. When all steps have completed (by succeeding or by failing), the behavior succeeds.

The root node of the ABT is a collection behavior node, with the initial goals of the agent as its children. This allows all of the initial goals of the agent to be pursued in parallel, and allows them to succeed or fail without affecting their siblings.

More complex structures of steps can be created by nesting these basic types of behaviors. For example, multiple sequential threads of behavior can be built by using a concurrent behavior with goals all of which are pursued by sequential behaviors. Section 4.14.2 shows how such nested behaviors can be more easily expressed.

Figure 4.4 shows how the cascading effects of success and failure are extended to accommodate the different types of behaviors. Collection behaviors in particular change the definition of these effects because a failed step does not cause the parent collection behavior to fail, and may in fact cause it to succeed if it was the last step. The effects must also be modified for sequential behaviors because only one step of a sequential behavior, the current step, is in the ABT at a time. This ensures that the steps of the behavior are executed in order. When the current step succeeds, the next step of the behavior is placed in the tree so that it can be executed. If the current step is the last step of the behavior, the behavior succeeds when that step succeeds.

In addition to these changes in the cascading effects of success and failure, there are additional ways to initiate these effects. As before, the success or failure of a primitive action or the failure of a goal because of no applicable behaviors initiates these effects at the `succeed_step` and `fail_step` level. In addition, any step in the tree which has an associated success test that evaluates `true` succeeds, and causes the cascading effects defined by `succeed_step` of that step. Analogously, any context condition that evaluates false causes the associated active behavior to fail with effects defined by `fail_behavior` of that behavior.

```
define succeed_step(S)
  parent := parent_behavior(S)
  remove(S)
  if ( parent is a sequential_behavior ) then
    if ( S was last step of parent )
      succeed_behavior(parent)
    else
      place next step of parent as child of parent
  else   ;; note: parent is collection or concurrent behavior
    if ( parent has no children ) then
      succeed_behavior(parent)

define fail_step(S)
  parent := parent_behavior(S)
  if ( parent is collection ) then
    remove(S)
    if ( parent has no children ) then
      succeed_behavior(parent)
    else
      no-op
  else
    fail_behavior(parent)

define succeed_behavior(B)
  succeed_step(parent_goal(B))

define fail_behavior(B)
  add behavior to list of failed behaviors in parent goal
  remove(B)

define remove(Node)
  removes the node and its subtree from the ABT.
  Any executing actions that are removed are aborted.
```

FIGURE 4.4: Cascading effects of success and failure of behaviors, goals and actions.

4.7 Example Behavior

```
(concurrent_behavior stop_fight (kid1 kid2)
       (precondition feeling energetic and
                     $$kid1 and $$kid2 are nearby and
                     $$kid1 and $$kid2 are my children)
       (context_condition $$kid1 and $$kid2 not too far away)
  (with (success_test I am between $$kid1 and $$kid2)
    (subgoal move_between $$kid1 $$kid2))
  (subgoal communicate stop fighting concept))
```

FIGURE 4.5: Example behavior.

An example behavior to illustrate the syntax of Hap is shown in Figure 4.5. The full grammar of Hap syntax is shown in Appendix A. The syntax for test conditions in preconditions, success tests and context conditions are not shown, because they are not central to the language. In general they are boolean valued match-expressions over anything in the agent's memory, anything the agent can sense, and reflection of Hap's internal processing state. The syntax for a concept in a communication goal is also not shown. Communication goals are described in detail in Chapter 8.

This behavior is one of the methods for the agent to stop a fight between two of her kids. The precondition shows the conditions under which it can be chosen: that the two fighting are her kids, that she is feeling energetic, and that the fight is nearby. The context condition shows the conditions under which she will continue to pursue this behavior. If the kids move too far away, then she will not continue to chase them. This behavior has two goals as its steps: one to move between the two kids and the other to tell them to stop fighting. These two steps will be pursued concurrently because the behavior is a concurrent behavior.

A success test is added to the `move_between` goal, so that if the fight moves around the parent, she will recognize the opportunistic success of that goal. Annotations to goals are added using a `with` form. If no `with` form is given then all possible annotations are given default values. In this way the second goal is given a default success test of `false`, so it will never opportunistically succeed.

4.8 Expressiveness

To round out the ability to express a detailed interactive behavior in Hap, I need to describe a number of details that increase the expressiveness of the framework I have described so far. The first is a special type of step and the rest are annotations that modify the interpretation of any steps they are applied to.

The step can be used in a behavior anywhere any other step is used. The additional annotations to steps can be added using a `with` clause as shown in Figure 4.5.

Each of these details arose while building specific agents, to express some concept that the agent builder wanted for the personality being built. After being added to the architecture, these language features have been pervasively used in all of the agents built in Hap. For me it is hard to imagine building a character in Hap that does not use each of these features repeatedly.

Many of these annotations can be viewed as allowing exceptions to the default execution behavior of Hap. By providing language support for expressing these exceptions, Hap requires less work by the author to program "around" the default behavior. In Chapter 10 I suggest that authors will always want to break "rules" that are instituted by the architecture because they will always come up with artistic visions that were not anticipated. In Hap, I have attempted to make the architecture support such "breaking of the rules". The experience of myself and others who have used the system suggest that these exceptions are useful for the detailed expression of interactive personalities. This usefulness is suggested by the code examples in the coming chapters, especially the detailed description of a complete agent in Chapter 7.

4.8.1 Wait Step Type

In addition to goals and primitive actions, Hap includes a special type of step called `wait` that can be included in a behavior. A `wait` step is a step that is never chosen to be executed. Its main purpose is to be placed in a sequential behavior at the point when that behavior should be paused. Because it is never picked for execution, when it is reached in a sequential behavior, it suspends that behavior until it is removed. It can be removed by an associated success test becoming true, or by success or failure of one of its ancestors in the tree.

The most common way to use a `wait` step is with a success test that encodes the situation in which the behavior should continue.

```
(sequential_behavior generic_demon (argument1 ...)
  (with (success_test  firing condition for the demon)
    (wait))
  demon body_step_form ... )
```

FIGURE 4.6: Structure of a demon in Hap.

A special case use of `wait` is to encode the common concept of a *demon*. This is done by placing a `wait` step as the first step of a sequential behavior. The firing condition for the demon is added as a success test for this `wait` step, and the body of the demon is placed in the second and later steps of the behavior. The structure is shown in Figure 4.6. If the author wishes to create a demon whose body is a concurrent or collection behavior (rather than sequential) this is done by creating an intermediate subgoal that is the only element in the demon body, and creating a single behavior for this subgoal of type concurrent or collection. This structure for one whose body is a concurrent behavior is shown in Figure 4.7. Either

```
(sequential_behavior generic_demon (argument1 ...)
  (with (success_test firing condition for the demon)
    (wait))
  (subgoal generic_demon_subsidiary $$argument1 ...))

(concurrent_behavior generic_demon_subsidiary (argument1 ...)
  demon body_step_form ... )
```

FIGURE 4.7: Structure of a demon with concurrent body steps expressed in Hap.

type of demon is enabled whenever the sequential behavior is in the active behavior tree. Whenever the condition becomes true the demon fires, by the firing of the success test and removal of the `wait` step, and the body can then be executed.

4.8.2 Persistence of Steps

It is often desirable for an agent to have a number of goals that are always present. Such goals might include `respond_to_danger`, `amuse_self`, or others depending on the personality being encoded. Hap allows this by providing a `persistent` annotation that can be added to any step. If a step has this annotation, it is not removed when it succeeds or fails. It can only be removed when the subtree it is a part of is removed (which happens when one of its ancestors succeeds or fails).

It is common in the agents I have built to have a number of goals at the top level that are persistent, by being marked with a `persistent` annotation. These goals typically have a good bit of structure under them as they are the fundamental elements of the agent's behavior.

Specifying that a goal is persistent is also sometimes appropriate in goals that are part of more transient behaviors. For example, consider a behavior a housecat might use to tell a human to feed it. The basic behavior for this communication might be to alternately rub its empty food bowl, look up at the human, and meow. There is a problem with this behavior if the person picks the cat up for any reason. So one might build this behavior with two parts: the first is a goal that causes the described behavior for some time, and then gives up; the second goal needs to respond to being picked up by jumping back down. It is undesirable for the second goal to be removed when it is completed because the cat may be picked up again. It needs to be a persistent goal. (The behavior for this goal is an example of a demon as described above: a sequential behavior with a wait step as the first step, a success test on this wait step that recognizes when the cat has been picked up, and a second step to jump down.)

Hap also provides `(persistent when_succeeds)` and `(persistent when_fails)` annotations for steps. The `(persistent when_succeeds)` annotation causes the associated step to be continuously pursued as long as it is successful, but if it ever fails the normal effects of a step failing occur, for example causing the enclosing behavior to fail. The `(persistent when_fails)` annotation causes the associated step to be repeatedly pursued until it succeeds. In that case the normal effects of a step succeeding occur, for example continuing to the next step in a sequential behavior.

4.8.3 Ignore_Failure

When a goal or other step fails, this failure causes the enclosing behavior to fail, which may cause the parent goal to fail, etc. This reaction is not always desirable. In some cases, an author may want to include that a goal be attempted as part of a behavior, but that the attempt need not be successful for the behavior to succeed. For example, consider building a behavior for responding to a stranger who has just done you a good turn that cost her money. Depending on the character being built, this behavior would likely include two goals: one expressing thanks, and the other paying for the cost and time. If the second goal did not succeed, this would not be cause for the behavior to fail; for some personalities the expression of thanks and the attempt to pay is enough of a response in this situation.

Hap allows one to express such a behavior by adding an `ignore_failure` annotation to the goal to pay. The goal will then be attempted in good faith, possibly succeeding. But, if it fails, the enclosing behavior will not fail as a result.

4.8.4 Effect_Only

Normally in any behaviors, all of the steps have to be attempted in order for the behavior to succeed. Using the `ignore_failure` annotation above an author can allow some of the steps to fail, but they must still be attempted. It is sometimes useful to have truly optional steps that need not even be attempted as part of the behavior. In Hap, this idea of a truly optional step is only possible in the context of a concurrent or collection behavior. (If one were placed in a sequential behavior, it would have to be attempted because of the sequencing constraint: when all of the steps before it in the behavior have finished it is the only thing available in this behavior to be attempted, and any steps after it cannot be pursued until it is completed.) Optional steps are possible in Hap by adding an `effect_only` annotation to the step.

Optional steps can be used in two meaningful ways in concurrent or collection behaviors. The first is to express an optional part of a behavior that happens or not by pure chance. For example, when people say a person's name in a sentence and that person is nearby, they sometimes glance at the person and sometimes they don't. This can be expressed in Hap by using the `effect_only` annotation in a single concurrent behavior with two goals. The first goal is a goal to say the person's name. The second goal is an optional goal (annotated effect_only) to glance toward the person. Because of the effect_only annotation on the glance goal, only the first goal is necessary for the behavior to complete, so the behavior may complete with no glance. But, if the second goal is chosen and executed before the first goal completes, then the glance will occur with the utterance.

The second use of the effect_only annotation is to create an optional demon that once triggered must be pursued. This is done by giving the demon a positive priority modifier. (Priority is discussed in Section 4.11. Basically, higher priority goals are pursued before lower priority goals.) Before it is triggered, the wait step is never chosen. When triggered, all of the steps of the demon have a higher priority than the other steps of the behavior, and so are preferred to them. This can be used to make demons that respond to minor contingencies in a behavior, or for other uses. In fact, this is exactly the situation of the housecat trying to tell a human to feed it, that was described above in Section 4.8.2. The behavior I proposed in that section has two parts: one part that tries to physically convey that the food bowl is empty by rubbing against it while meowing at the person for some time; and the other part that is a demon that responds to being picked up. As I mentioned in that section the demon needs to be persistent in case the person picks the cat up multiple times, but it also needs to be optional in case the person never picks up the cat. The behavior should succeed if the first part succeeds even if this step is never attempted. It would likely be appropriate to make the behavior fail if either goal fails. This can be accomplished by using the effect_only annotation and a (persistent when_succeeds) annotation, and not including the ignore_failure annotation. Then if the cat is unsuccessful at jumping down when picked up, the behavior would fail.

4.8.5 Number_Needed_for_Success

The final annotation for expressiveness is number_needed_for_success. Normally all of a behavior's steps must succeed for the behavior to succeed, or some of the steps can be explicitly annotated as optional by using the annotation effect_only. This is not always desirable. There are cases, for example, where two goals should be pursued concurrently, but if either one succeeds, the behavior should succeed. (See Figure 5.4 and surrounding description for an example of this need and use.) Effect_only does not capture this expression because the author doesn't know ahead of time which goal will succeed and which will become optional. To capture this case, Hap provides the annotation number_needed_for_success.

This annotation can only be used in concurrent or collection behaviors. In these behaviors is defines the number of steps that must succeed (or succeed or fail in the case of a collection behavior) in order for the behavior to succeed. Steps that are marked effect_only are not included in the count, so this annotation can be meaningfully combined with effect_only. For example a behavior could be written with two effect_only goals and three other goals with a number_needed_for_success annotation with value two. Hap would pursue all five goals concurrently, and the behavior would succeed whenever two of the three goals not marked effect_only succeed. Combining this annotation with nested behaviors allows still more expressiveness.

4.9 Full Cascading Effects of Success and Failure

Now that I have described all of the annotations that affect success and failure of goals, I am in a position to describe the full effects of success and failure. As mentioned before, the success and failure of goals and behaviors are defined recursively. These effects are presented in pseudo code in Figure 4.8 which is an expanded version of the more basic descriptions given in Figures 4.2 and 4.4.

These effects are initially triggered in the same ways described in Section 4.6: the success or failure of a primitive action; the failure of a goal because no behaviors are applicable; and the firing of a success test or context condition. The success of a primitive action or firing of a success test causes the effects specified by `succeed_step` on the associated ABT node. Both the failure of a primitive action or failure of a goal because it is chosen and no behaviors apply cause the effects specified by `fail_step` on the associated ABT node. And, the firing of a context condition causes the effects specified by `fail_behavior` on the associated ABT behavior node.

Whenever a step succeeds, through the initial causes above or because of the recursive causes described below, the effects specified in `succeed_step` occur. First, if it is marked `persistent` or `(persistent when_succeeds)`, then the step is reset with the effects specified by `reset_step`. `Reset_step` is described below; it adjusts the ABT so the step is again available to be pursued.

If the step is not persistent, then it is removed along with any of its children. (If an executing action is removed, it is aborted.) If the parent behavior is a sequential behavior, then the next step of that behavior is placed in the tree as its child. If there is no next step, the behavior succeeds with the effects of `succeed_behavior`. If the parent behavior is not a sequential behavior then nothing is done beyond removing the step, unless the step was the last needed for the behavior to succeed. This is the case if `Num_needed_for_success` steps not marked `effect_only` have been removed. `Num_needed_for_success` is the value specified by the behavior's `number_needed_for_success` annotation if present. Otherwise, it is set to the number of steps in the behavior that are not marked `effect_only`. This is equivalent to succeeding if there are no remaining steps, if there are no remaining steps that have not been marked `effect_only`, or if `N` steps that aren't marked `effect_only` have been removed and `N` or fewer steps are needed for success due to a `number_needed_for_success` annotation in the behavior. In this situation, the behavior is caused to succeed with the effects of `succeed_behavior`.

Whenever a step fails, through the initial causes above or through the recursive causes described herein, the effects specified in `fail_step` occur. If the step is annotated with `ignore_failure` then the effects of `succeed_step` occur instead. Otherwise, if it is annotated as `persistent` or `(persistent when_fails)`, then it is reset (in the manner described in `reset_step`) so that it is available to be pursued anew.

If the step is not persistent nor marked `ignore_failure`, then it is removed (as described for `remove`) and the further effects depend on the type of the parent behavior. If the parent behavior is a collection behavior, then failure of any step is treated the same as success of a step, so the same effects occur as above when a step succeeds in a collection behavior:

```
define succeed_step(S)
  if ( S marked persistent or (persistent when_succeeds) ) then
    reset_step(S)
  else
    parent := parent_behavior(S)
    remove(S)
    if ( parent is a sequential_behavior ) then
      if ( S was last step of parent )
        succeed_behavior(parent)
      else
        place next step of parent as child of parent
    else   ;; note: parent is collection or concurrent behavior
      if ( have removed Num_for_success of parent's non-effect_only children ) then
        succeed_behavior(parent)

define fail_step(S)
  if ( S marked ignore_failure ) then
    succeed_step(S)
  else if ( S marked persistent or (persistent when_fails) ) then
    reset_step(S)
  else
    parent := parent_behavior(S)
    remove(S)
    if ( parent is collection ) then
      if ( have removed Num_for_success of parent's non-effect_only children ) then
        succeed_behavior(parent)
      else
        no-op
    else if ( S not marked effect_only ) then
      fail_behavior(parent)

define succeed_behavior(B)
  succeed_step(parent_goal(B))

define fail_behavior(B)
  add behavior to list of failed behaviors in parent goal
  remove(B)

define reset_step(S)
  if ( S is a goal ) then
    remove(child(S))
    clear list of failed behaviors in S
  else ;; note: S is physical or mental primitive action
    abort if executing and mark as not executing

define remove(Node)
  remove the node and its subtree from the ABT.
  any executing actions that are removed are aborted.
```

FIGURE 4.8: Full cascading effects of success and failure of behaviors, goals and actions.

nothing is done beyond removing the step unless this is the last step needed for the behavior to succeed. The conditions for this are the same as described above.

If the parent behavior is not a collection behavior, then the failure of the step causes the parent behavior to fail unless the step is annotated as `effect_only`. If it is marked `effect_only` then nothing is done beyond the removal of the step. The effects of a behavior failing are described below in the description of `fail_behavior`.

As `succeed_behavior` specifies, whenever a behavior succeeds it causes the parent goal to succeed with all of the effects described above for `succeed_step`.

As specified by `fail_behavior`, whenever a behavior fails that behavior is added to the list of failed behaviors for the parent goal, and the behavior is removed from the ABT. The recorded information is used to ensure that each behavior is attempted at most once per goal (unless that goal is reset).

`Reset_step` specifies how the ABT is adjusted to make a step again available to be pursued. For a goal this means removing any behavior that is currently pursuing this goal (as specified by `remove`), and clearing the record of failed behaviors for the goal. In this way, the goal is a leaf goal that is available to be chosen for execution. It also has no recorded failed behaviors, so all of the behaviors for this goal are available to be pursued in turn.

If the step being reset is a primitive physical action the action is aborted if it is executing and marked as not executing. If the step is a primitive mental action nothing is done. (A primitive mental action executes atomically, so it is never marked as executing.) In this way it is again a non-executing leaf goal, available to be chosen for execution.

Removing any node from the ABT causes all of the children of the node to also be removed. Nothing else is done unless an executing action is included in the removed nodes. For any executing actions that are removed, an abort signal is sent to the body to abort the action.[6] Removing a behavior causes its parent goal to become a leaf goal of the ABT. As with all leaf goals, it is then available to be chosen for execution which can result in another behavior chosen to pursue it.

4.10 Conflicts

Concurrent behaviors and collection behaviors introduce parallel threads of activity to a Hap agent. This possibility exists at the very highest level because the root node of the ABT is a collection behavior node. This is an important property for believable agents, as most personalities juggle and interleave multiple activities or parts of the same activity simultaneously. There is an issue, however, of which activities can be pursued simultaneously and which cannot.

[6]An abort signal may or may not be successful. When not successful, the fact that the action is executing is recorded elsewhere for use by Hap's conflict mechanism (see Section 4.10) to avoid issuing actions that use the same resources or conflict in other ways.

I know of two ideas that seem important in determining what activities can be mixed. The first is physical constraints; for example, most agents simply cannot move in two directions at the same time. These constraints are given by the capabilities of the agent's body in the domain to which Hap is connected. The second idea is that some activities conceptually conflict, even though there is no physical resource conflict. For example, while sleeping the primitive actions being executed are rather sparse and do not by themselves preclude concurrently executing other primitive actions. Nevertheless, in personalities that don't sleepwalk, sleeping conflicts with actions that significantly move the body. Similarly, adults have conversations while eating dinner, but would typically not play a game while eating. Children might not have this same conceptual conflict between playing and eating a meal.

Hap allows authors to specify pairs of goals or actions which are incompatible in a *conflict list*. Actions that conflict because of primitive resource conflicts can be specified as such a pair, or this information can be provided by a function from the domain. When a Hap agent is running, Hap prevents these conflicts by not allowing any conflicting goals or actions to execute at the same time. The process of enforcing these conflicts is described in more detail in Section 4.12.

4.11 Priority

Every goal or other type of step in the active behavior tree has a priority that is used by Hap when choosing which goals to attend to. Each priority is an integer value, and together the priorities in the active behavior tree define a partial order of the steps in the tree at any given time.

Priority is assigned to goal instances rather than to types of goals, because identical goals could have different priority depending on the context in which they arise. For example, the goal of going to an area as a subgoal to a run away goal would likely have a higher priority than the same goal in pursuit of an exploration goal.

Priorities of the initial goals of the agents are given by the author by specifying a priority annotation for each goal. This ordering should represent the agent's general preferences for pursuing one goal over another. For example if an agent has three top level goals — `amuse_self`, `sleep_when_tired`, and `respond_to_danger` — an author might choose to give them priorities 10, 20, and 30,[7] respectively, because responding to danger is more critical to the agent than sleeping or playing, and sleeping is more critical than playing. Goals with equal preference or urgency would be given equal priority.

Priorities of steps lower in the active behavior tree are inherited from their parents with possible modification. Each step in a behavior can have a `priority_modifier` annotation, a number that is added to the priority of the parent goal to specify the priority of the step.

[7]An obvious extension to Hap is to allow priorities to change based on changes in the situation, for example the goal to sleep might become higher priority as the agent becomes more tired. Such an extension is an easy addition, but it has yet to become needed in the agents I have built. Neal Reilly in his dissertation [Neal Reilly 1996] explores this topic as it relates to emotional changes.

Steps that are conceptually of lower priority than the parent goal can be given a negative priority modifier, while steps that are conceptually of higher priority can be given a positive priority modifier. In the example of an agent going to an area in pursuit of a run-away goal or an exploration goal, the goal to go to an area would inherit the priority of the parent, and so no priority modifier would be necessary. Consider the above agent with three top level goals: `amuse_self`, `sleep_when_tired`, and `respond_to_danger`. Let's consider the case where it is not tired and no danger is present, so it chooses to amuse itself. One behavior an author might write for the `amuse_self` goal is to play a game with a friend. This behavior might have four steps: choose a friend, go to the chosen friend, invite the friend to play, and play. In this simple agent, one would likely not want any modifier to the priorities of the first three goals, but one might want one for the fourth goal. After committing to play a game, a polite agent would prefer to play the game over sleeping. This can be accomplished by attaching a priority modifier of 15 to the play goal of this behavior.

4.12 Execution of Hap

Hap executes by repeatedly performing a four step loop given in Figure 4.9. Each of these steps is described below.

1. If needed adjust the active behavior tree for actions that have finished.
2. Else if needed adjust the ABT based on changes in the world.
3. Else if needed adjust suspended goals.
4. Else pick a leaf step to execute, and execute it.

FIGURE 4.9: Hap execution loop.

First, Hap updates the ABT for the actions that have finished. Completed actions cause the associated step in the ABT to succeed or fail. This will cause the step to be removed (unless it is marked with an applicable persistent annotation). It may also cause other changes in the ABT, as the success or failure causes the enclosing behavior to succeed or fail, perhaps affecting the parent goal, etc. These cascading success and failure effects are spelled out in Section 4.9.

The second part of a Hap execution cycle is to update the ABT based on changes in the world or internal state of the agent. Conceptually, all of the success tests and context conditions in the active behavior tree are evaluated. For any success test that is true, the associated step is made to succeed. This success can have cascading effects in the ABT, just like successful actions as described above. In addition, any descendents of this step are removed, which can potentially cause executing actions to be aborted.

For any context condition that is false, the associated behavior fails. It is removed along with any of its descendents. Any executing actions that are descendents of this behavior are also aborted. The goal that this behavior was in service to is again a leaf node of the ABT, and as such, it is available for execution allowing another behavior to be chosen for it.

Third, the set of active goals is updated by adjusting which goals are marked `suspended` and which are not. This uses the information about actions that conflict as well as about goals that conceptually conflict (described in Section 4.10). No two steps that conflict are allowed to execute at the same time. Hap enforces this property by marking appropriate nodes in the ABT as `suspended`. A step (and any subsidiary nodes) is marked as `suspended` when a conflicting step with a higher priority is executing. Actions are considered executing if they have been sent to the body to be executed. Goals are considered executing if they have been expanded with a behavior, that is, if they are not a leaf goal. Steps may also be marked `suspended` because of conflicting steps with an equal priority. For conflicting steps with equal priorities, whichever is chosen to be executed first is allowed to execute, and the other is marked as `suspended`. Steps marked as `suspended` are unmarked whenever the step that caused them to be marked is removed from the tree for any reason. They are then available to be pursued, unless they are re-marked as `suspended` because of another conflicting step.

Fourth, Hap chooses a leaf step to execute. This choice is done by the method given in Figure 4.10, and is described in more detail here. First, Hap only chooses from leaf

1. Never choose `wait` steps, executing actions, or steps marked `suspended`.
2. Prefer steps with higher priorities.
3. Prefer to continue the same line of expansion.
4. If multiple steps still remain, randomly choose among them.

FIGURE 4.10: Step Arbiter – Method for choosing which leaf step to execute next.

steps that are available to be executed, that is, all leaf mental actions, leaf physical actions that have not been sent to the body for execution, and leaf goals that are not marked `suspended`. Of those available leaf steps, higher priority steps are chosen over lower priority steps. If multiple steps remain as possible choices, Hap prefers to work on the same goals and behaviors it has recently been pursuing. This is to maintain some sense of focused purpose to a Hap agent's activity, rather than having it switch from behavior to behavior without reason.[8] Specifically, this choice is accomplished by recording at each concurrent or collection behavior the last goal that was pursued from this behavior. Leaf

[8]This pursuit of the same line of expansion does not preclude opportunistically pursuing other goals when they are cheap, convenient and compatible. This is described in Section 4.13.1.

goals that are descendants of this goal are preferred over descendants of other goals in the concurrent behavior. When that goal finishes, all of the remaining steps of this concurrent behavior compete on equal footing again. If multiple leaf goals are equally preferred after each of these steps, Hap chooses randomly among them.

Once a step has been chosen, it is executed. Each type of step has its own method of execution. A primitive physical action is sent to the body to be executed. That step node is then unavailable to be chosen while the action is executing. A primitive mental action is executed by simply performing it. (This actually takes the form of evaluating arbitrary code, and thus can affect any part of the Hap mental state.) A subgoal is executed by choosing a behavior for it. The behavior is chosen from those written for this goal by the method described in Figure 4.3. The chosen behavior is written in the active behavior tree as the child of the goal. The steps of the behavior are written as children of the behavior and are themselves then available to be chosen to be executed the next time through Hap's execution loop.

After executing the chosen step as appropriate, Hap continues execution with the first stage of its execution (in Figure 4.9).

4.13 Real-Time Hap

As mentioned in the description of a domain for these believable agents (in Chapter 3), Hap is designed for agents in real-time, three-dimensional worlds. Hap has also been used in non-real-time worlds which are more abstractly modeled, but those worlds are strictly less demanding than real-time worlds such as the one described in Chapter 3. In this section I first describe one previously presented property of Hap and then describe additional features to respond to the challenges of a real-time domain for believable agents. These features of Hap are all to respond to three challenges raised by the task of building real-time believable agents.

The first challenge is one of the requirements for believable agents (described in Section 2.7.2): for believable agents to have convincing behavior, they need to be able to perform multiple actions in parallel and pursue multiple higher level activities concurrently.

Second, when an animated creature is performing a sequence of actions, the next action must often be known in advance to determine in detail how to animate the current action. For example, depending on whether the agent is immediately jumping again or stopping, the way the agent transfers momentum during the final part of a jump is different.

Third, creatures in a real-time domain must think fast enough to keep up. In my domain primitive actions have durations between 100 and 1500 milliseconds. The creatures must be able to respond to their own and other creatures' actions as they are occurring. In addition, to appear active they must be able to produce actions at the same rate as they are being completed by their own bodies. This demand is increased because multiple actions are often executing simultaneously. This is the requirement of responsiveness that is described in Section 2.7.3.

4.13.1 Concurrent Pursuit of Goals

The first of these challenges is addressed by the mechanisms already described for concurrency in Hap. Hap allows agents to hold multiple parallel goals through the top level set of goals and through concurrent and collection behaviors which arise during execution. Hap manages these multiple goals by concentrating on the most critical according to its step arbitration mechanism, and attends to other goals after the current goal completes or as events trigger demons of higher priority.

In addition, Hap attempts to attend to as many of its active goals as is possible by first attending to the most critical and mixing in others as time allows. In each decision cycle, Hap chooses the highest priority goal of the available leaf goals. This thread of behavior is attended to until it finishes or is interrupted, as above, or until it becomes suspended. For example, when a **Jump** action in a sequential behavior is sent to the body to be executed, the behavior cannot continue until the action completes. When a thread is suspended Hap uses the available processing time (in this case approximately 1200 milliseconds of real time) to attend to the other, perhaps unrelated, available goals. A thread of behavior can be suspended for three reasons: the current step in the thread is an executing action; the current step is the special step form `wait`; or the current step conflicts with a higher priority executing goal or action and is therefore marked `suspended` (as described in Section 4.12). As this greedy, multi-thread expansion continues, this can result in multiple actions being issued to the body to be executed in parallel.

4.13.2 Lookahead

The second challenge is that smooth motion in a real-time, animated world requires that the agent know the next action for a particular set of muscles before the current action finishes. To allow the motor control part of the animation system to provide smooth motion, Hap attempts to provide the next action for each thread before the current action finishes. One hundred milliseconds prior to the completion of an action, Hap assumes that the action will complete successfully. It then can use this time to compute the next action along that thread. If an action is produced, it is sent to the motor system to be executed after the current action. All of Hap's reactive mechanisms apply to these pending actions as well as to normal Hap execution, so in the event that Hap chooses to abort a pending action, a message is sent to the motor system and it is removed.

Of course, if the agent is currently attending to something more critical than this thread, it will continue to be attended to and the next action will likely not be computed. The motor control system will assume that action for this set of muscles is temporarily ending. Also, if the current action fails after it has been assumed to finish successfully, the agent must recover from its incorrect assumption using its various reactive mechanisms, such as success tests and context conditions.

4.13.3 Responsiveness of Agents

The third challenge of a real-time world is that the agents be responsive enough for the speed of the world. They must be able to think at a speed that is appropriate for what people expect of them in order to be believable. This partially means that they need to be able to come up with the next action quickly enough so they are not pausing unduly while walking or performing other behaviors; but it also means that they must react at reasonable speed to stimuli in the world or from human interactors. What "reasonable speed" means depends on the character being built and the expectations of the human interactors. In my domain, primitive physical actions take between a tenth of a second and 1.5 seconds to perform, and that is the order of reaction time that is needed for many agents in this domain.

There are three properties of Hap to aid in the needed responsiveness.

Hap Is Not a Planning-Based Architecture

The basic mechanism for making decisions in Hap is not a planning process. Hap does not need to reason from first principles in order to decide how to pursue its goals. Instead, all goals are pursued by choosing a behavior from a fixed set of author-written behaviors. The mechanism for choosing a behavior is also strictly bounded and in practice efficient, as described below. This has the effect of there being few decisions and little processing between deciding to act on a goal and issuing the first primitive actions toward that goal.

Of course, Hap's mechanism is rich enough to encode arbitrary reasoning and planning approaches, but its more natural use is to directly build the routine behaviors that a personality engages in for particular situations. There are times when larger reasoning is needed to express a particular personality, but all activity need not use these expensive mechanisms. It would also not be appropriate for them to. Most personalities don't reason from first principles to decide how to play a game or even to decide what route to take from work to the store. People have routines — routine ways of responding to greetings, routine ways of amusement, etc. — and so should believable agents. In the situations where planning is appropriate for the personality being built, specific planners can be built. Care should be taken to not make this the only path to a response to particular stimuli unless the practical reaction time by the planner is acceptable or the delays are appropriate for the personality. Hap's reactive mechanisms can be useful to manage these issues; one simple mechanism is a context condition that times out if the planning behavior is taking too long and lets a less costly method be used instead. Agents built in Hap have included various special purpose planning and other reasoning behaviors.

Selective Sensing

Sensing takes time. Hap limits this cost by only sensing what is needed for the agent at any given point in time. Hap creatures employ task-specific sensors which are automatically turned on or off as needed. Each sensor observes a low level aspect of the world and notifies the mind when that aspect's value changes. Typical sensors for the Woggle domain described in Chapter 3 are “can I see Woggle X jumping” and “what is the position of Woggle X”.

The aspects of the world which must be known to evaluate an agent’s preconditions, success tests and context conditions are noted when these conditions are written by associating a list of sensors for each condition. Hap automatically manages the sensors by turning them on and off when appropriate. As a leaf subgoal is chosen to be executed, sensors needed to evaluate the preconditions for that goal’s behaviors are automatically turned on, and then turned off again after a behavior is chosen. Likewise, when a particular goal or behavior is present in the ABT, the sensors relevant to evaluating any associated success tests or context conditions are turned on. When that goal or behavior is removed from the tree because of success, failure or irrelevance, the sensors are turned off. Because the same sensor may be needed for several different conditions at a time, the sensors are shared and reference counted. In agents that have been built in Hap this yields roughly a factor of two reduction in active sensors as a result of this sharing.

Compilation and Incremental Evaluation of Match Expressions

Typically a Hap agent has a large number of continuously monitored conditions (context conditions and success tests) active at any given time. In order for these agents to run fast enough for a real-time animation system, I believe it is useful to evaluate them incrementally. Hap provides this incremental evaluation of the conditions by implementing them using the Rete algorithm [Forgy 1982]. Thus characters written in the Hap language are compiled to RAL [Forgy 1991], a production system extension of the C programming language that includes a Rete incremental matcher implementation. The character’s ABT is represented in working memory, with context conditions, success tests, and preconditions compiled to rules. These rules fire to prune and expand the tree, with additional runtime support rules and functions included to complete the architecture.

4.14 Programming Language Details and Features

Expressing a detailed interactive character’s behavior in Hap is an artistic expression just as painting or animation is. The difference is that the tools that must be used are not a brush or pen and ink, but rather a programming language. In Section 4.8, I discuss some of Hap’s features to enable this expressiveness. And, in fact, this expressiveness is one of the major continuing themes that is spread throughout this chapter and the rest of the disseration.

To make Hap as a programming language and tool of expression better and more easy to use, it is useful to have a number of well-understood concepts from programming languages in the language. These do not change the fundamental power of Hap as a programming language, but they do make it easier to use.

4.14.1 Data Flow and Scoping

Hap is lexically scoped. Goals and behaviors can be viewed as expanded versions of function calls and functions. In addition, Hap allows an author to use dynamic scoping when desirable. An example behavior that illustrates the scoping in Hap is given in Figure 4.11.

```
(concurrent_behavior run_away (from_what)
       (precondition can see $$from_what)
       (locals
          (speed compute_speed(sense_my_location(),$$from_what)))
       (dynamic_locals
          (direction compute_direction($$me,$$from_what)))
       (context_condition not a dead end in $$direction)
  (subgoal move_in_direction $$direction)
  (subgoal adjust_direction_when_needed $$from_what))
```

FIGURE 4.11: Example behavior to show scoping.

Each agent can include various global variable declarations, in addition to the lexical and dynamic variable declarations included in each behavior. Global variables can be referenced and set from any behavior.

Similar to a traditional function body the outer-most scope of the behavior is created by the formal variables, in this case `from_what`. The goals for this behavior must have the name `run_away` and a single argument that is bound to the variable `from_what` when this behavior is chosen to pursue the goal.

The precondition of the behavior, in addition to determining if this behavior is appropriate in the current context, can use and bind variables as well. In this example, the precondition references the argument `from_what` and creates no bindings of its own.

The `locals` form allows the declaration and initialization of local variables. The initialization expressions can reference any of the previously defined variables. In this case a variable, `speed`, is defined and initialized with a value from a function call that references the previously defined variable `from_what`. It also uses a primitive sensor, `sense_my_location`, to get the agent's current location.

The `dynamic_locals` form allows the creation of dynamic variables. These variables can be referenced by all of the forms in the behavior body as well as by any behaviors that are in the call structure from this behavior. For example, any behavior for `move_in_direction` can reference the `direction` variable. This declaration defines a dynamic variable `direction`. The initialization of this variable references the global variable `me` as well as the lexical variable `from_what`. It could also reference any variables created in the `precondition` or `locals` forms.

The `context_condition` annotation is placed after all of the variable declarations and can reference any of the declared variables. In this case it references the computed direction as part of its test. Similarly, all other behavior annotations are placed below `context_condition`, and like it can reference any of the variables of the behavior.

Any steps or annotations on steps can reference any of the defined variables.

If the same variable name is defined twice, the lower one (the inner scope) is the one that is available to be referenced.

Values can be returned from any behavior using a `return_values` form. These values are ignored unless the invoking goal has a `bind_to` annotation. In this case the values included in the `return_values` form are bound to the variables in the `bind_to` form, and can be used by any later steps in the behavior. If a dynamic variable binding is desired, a `dbind_to` form can be used instead of the `bind_to` form. An example of these forms is shown in Figure 5.2. More interesting uses of this feature are described in the generation of natural language in Chapter 8.

4.14.2 Idioms and Macros

Just as in many programming languages, there are common idioms of usage in Hap for recurring control structures. Demons (described in Section 4.8.1) are one such idiom. All demons have a structure similar to that shown in Figure 4.6. It is sometimes useful to give such idioms their own syntax to make their writing faster and easier. Hap allows an author to extend Hap's syntax by a built in macro mechanism similar to the macro mechanisms of Lisp, Scheme or C.

It is not always desirable to give common idioms their own syntactic form. In particular, if the meaning of the structure would be obscured, or if opportunities of expression would be removed, then the idiom should not be expressed as a macro. For a discussion of issues relating to this see Section 5.1.2.

One idiom that I do not express as a macro is an explicit test that must be true for a behavior to succeed. This is expressed in Hap by including an extra step as the last step of the behavior.[9] This step is a goal name which has no behaviors for it (I usually use the name "fail"), and a success test that encodes the required test for success. When the behavior reaches this step, if the condition is true, the success test will fire and the behavior will succeed. If the condition is false, the step will be pursued, no behaviors will apply (because none exist), the step will fail. This will then cause the behavior to fail.

Hap's macro mechanism allows an author to create syntactic extensions for frequently recurring syntactic patterns. Normally the steps of a behavior are either an action, mental action, subgoal or a `wait` step. (These can optionally be enclosed by a `with` clause to include any desired annotations.) If a given step form is not recognized as one of these forms, the author-specified macro definitions are consulted. If any of these match, the step

[9]If the behavior is not sequential, then one must be created. An extra sequential behavior is created with two steps. The first step is a goal with the previous concurrent or collection behavior as its only behavior. The second step is the last step of the behavior as described above.

form is transformed by the macro definition into a new step form which is then processed by the Hap compiler. Each macro definition maps a given step form into a new step form, and optionally also returns a list of new behaviors to be compiled.

```
Input step form: (demon <name> <condition>
                    <body-step1> ... <body-stepN>)


                         ⇓


Output step form: (with in_lexical_scope
                    (subgoal <name>))

Extra Behaviors: (sequential_behavior <name> ()
                   (with (success_test <condition>)
                      (wait))
                   <body-step1> ... <body-stepN>)
```

FIGURE 4.12: Macro definition for demon.

For example, a macro definition to create a demon form is given in Figure 4.12. This definition captures the common idiom of a Hap demon and allows it to be expressed as a single step that can then be used in any behavior. The demon form requires a `<name>` for the demon, a match `<condition>` that expresses when the demon becomes active, and the steps to perform when it becomes active (`<body-step1> ... <body-stepN>`). It is translated into a subgoal step and a behavior to express the demon.

There is one new concept that is needed for macros to be natural to use. This is the `in_lexical_scope` annotation. The demon step above is completely textually included in the behavior that it is part of. The extra behavior that is produced as a result of the macro expansion is not part of the same lexical scope, because all behaviors are defined at the top level. The `in_lexical_scope` annotation causes the behavior to be treated as if it were textually included in the same place as the invoking subgoal. This allows it to reference all of the variables that are available to the original `(demon ...)` step.

There are three syntactic extensions that are so frequently used that I want to include their descriptions here: `sequential`, `concurrent`, and `par_cond`.

`Sequential` and `concurrent` are forms that facilitate more complex behavior structures than simple sequential, concurrent or collection behaviors. They do this by making it easy to construct nested behavior structures. For example, one might want a behavior that executes two sequential threads concurrently. This can be done by creating a concurrent behavior with two `sequential` forms as shown in Figure 4.13. The steps of the sequential threads of activity would be the steps of the `sequential` forms. The nested control structure is created by expanding each `sequential` form into a subgoal and associated

```
(concurrent_behavior example ()
  (sequential
     thread1_step1
     ...
     thread1_stepN)
  (sequential
     thread2_step1
     ...
     thread2_stepN))
```

FIGURE 4.13: Example use of sequential form to created complex behavior structure.

sequential behavior. Thus the form presented in Figure 4.13 results in three behaviors: a top-level concurrent behavior with two steps (subgoals), and two subsidiary sequential behaviors each in service to one of the two subgoals.

Similarly, the `concurrent` form allows sets of concurrent steps to be embedded in sequential or other behaviors. The macro definitions for `sequential` and `concurrent` are given in Figures 4.14 and 4.15.

```
Input step form: (sequential <step1> ... <stepN>)


                             ⇓


Output step form: (with in_lexical_scope
                    (subgoal <generated-name>))

Extra Behaviors: (sequential_behavior <generated-name> ()
                   <step1> ... <stepN>)
```

FIGURE 4.14: Macro definition for sequential.

Another frequently used idiom that is included as a macro is `par_cond`. `Par_cond` allows conditional branching similar to a `cond` statement in Lisp or Scheme or a `switch` statement or nested if statements in C. The syntax and macro definition for `par_cond` is given in Figure 4.16. Conceptually, when this step is executed each of the match expressions, `<test1>` through `<testM>`, is evaluated in parallel. Of those that are true, one is chosen randomly, and the associated body steps are then executed in order. This is accomplished by replacing the step with a single goal with many behaviors. Each behavior

```
Input step form: (concurrent <step1> ... <stepN>)


                         ⇓


Output step form: (with in_lexical_scope
                    (subgoal <generated-name>))

Extra Behaviors: (concurrent_behavior <generated-name> ()
                   <step1> ... <stepN>)
```

FIGURE 4.15: Macro definition for concurrent.

```
Input step form: (par_cond
                   (<test1> <body1-step1> ... <body1-stepN1>)
                   ...
                   (<testM> <bodyM-step1> ... <bodyM-stepNM>))


                         ⇓


Output step form: (with in_lexical_scope
                    (subgoal <generated-name>))

Extra Behaviors: (sequential_behavior <generated-name> ()
                        (precondition <test1>)
                   <body1-step1> ... <body1-stepN1>)

                 ...

                 (sequential_behavior <generated-name> ()
                        (precondition <testM>)
                   <bodyM-step1> ... <bodyM-stepNM>)
```

FIGURE 4.16: Macro definition for par_cond.

has a precondition that is one of the test conditions. When the step is executed, all of the preconditions are evaluated. Of those that evaluate true, one of them is chosen, and the associated body is executed. Note that the test conditions can include anything that can be included in a precondition, for example sensing expressions.

`Par_cond` is an example of a syntactic extension that should be carefully used. It should only be used for simple choices between small segments of behavior or computation. This is because it hides many of Hap's features and opportunities of expression such as the ability to express context conditions for the sequentially executing groups of steps in each clause.

Each of these syntactic forms is frequently used in my agents. An author can create any such syntactic extensions that he or she finds useful.

4.15 Example of Processing

To illustrate how Hap works and to suggest how one uses it to express a particular personality, as well as the assistance it gives toward the requirements for believability, I present here a brief detailed example of Hap execution. This example will describe only a fragment of a complete agent, because I have not yet described various aspects of the architecture, such as emotion, composite sensing, natural language generation, and inference. These aspects are presented in later chapters.

This excerpt is part of an implemented agent called Wolf in the domain described in Chapter 3. A detailed description of Wolf and a more extended example of his processing are given in Chapter 7. Wolf was designed to be an aggressive personality and a show-off. The original character sketch that guided Wolf's development is quoted here.

> Wolf - mean guy. leader of wolf/dog pack. He provokes dog to do kind of mean things to shrimp. grey to black. likes to make fun of shrimp or pig, as appropriate. Also has an aesthetic side which makes him like to dance. Can practice dancing with dog. They can do awesome choreographed dances. He also likes to do certain athletics, to show how great he is.

Shrimp is a shy, timid character, and Pig became known as Bear when implemented. He is a big, wise, protector type of character. Dog was never implemented. He was originally a kinder, gentler sidekick to Wolf.

Wolf's top level goals include `amuse_self`, `respond_to_fight` and `blink` along with many others. All of the agents I have built or helped build have `amuse_self` and `blink` goals, but `respond_to_fight` is included in Wolf because of his personality. He notices when there is a fight in the world because he views it as an opportunity for him to have fun; he enjoys taking part in fights.

As I will describe, in this excerpt Wolf chooses to pursue his `amuse_self` by starting a game of follow-the-leader. He chooses the User to invite, and starts jumping toward him by issuing a **Jump** action. As he is flying through the air with his first jump, he blinks in parallel because of a `blink` goal in a concurrent behavior. About a third of a second later, he notices a fight in another part of the world. This recognition causes Wolf to have a goal to gang up on the fight. This goal is higher priority and conflicting with the `amuse_self` goal, so the `amuse_self` goal and all of its children are suspended. The **Jump** can not be suspended because Wolf is flying through the air. The new `gang_up` goal is expanded, eventually giving rise to a **Jump** action toward the fight. This is queued after the executing **Jump** action, allowing the motor system to create an appropriate landing for it, and Wolf is ready to head toward the fight. The entire excerpt is a little over one second in duration.

When we join Wolf, he has just chosen `amuse_self` as the next goal to pursue. This is because many of his other top level goals are paused demons that are waiting for an appropriate situation to react to. When choosing a behavior for `amuse_self`, Wolf has seven behaviors to choose from. They include watching the person standing in front of the screen;[10] releasing aggression; watching other creatures in the world; exploring the world; playing a game of follow-the-leader; jumping through the chute; and dancing on the pedestals. The set of behaviors chosen by the author for this goal is a very direct way of encoding the personality, as are the behaviors written for other goals in the system. For example, Shrimp does not have a behavior to release aggression as a behavior for amusing himself.

Hap chooses among these behaviors using the behavior arbiter presented in Figure 4.3 and surrounding description. Because the `amuse_self` goal has just been chosen for the first time, none of these behaviors has failed for it yet, so they are all available to be chosen by the first step of the behavior arbiter. The first two behaviors have preconditions which must evaluate to true in order for them to be chosen: the first behavior requires that someone be standing within sonar sensor range in order for the Woggle to look at them; and the second requires that Wolf feel aggressive.[11] Since neither of these conditions are true, Hap removes these behaviors from consideration. The remaining five behaviors all have default specificities of zero, so Hap chooses randomly among them, settling on the behavior to play follow-the-leader.

This behavior is a sequential behavior that gives rise to a single subgoal to `lead_someone`. The behavior is instantiated, and added to the active behavior tree along with the first (and in this case only) subgoal of the behavior. The next time around the execution cycle that goal is also available to be executed.

Because no higher priority steps have arisen (very little time has elapsed), Hap's step arbiter continues the same line of expansion by choosing this new goal and executing it by choosing a behavior for it. It has two possible behaviors. The first one has as its precondition an expression that matches any Woggle that is not equal to himself (Wolf). This has the effect of randomly picking among all of the other agents. The second behavior is more specific than the first, and its precondition matches any Woggle that is not equal to himself, and which he likes. Liking or disliking is a property of the emotion system that is discussed in Chapter 6.

The preconditions for both behaviors evaluate to true, but the second has a higher

[10]The position of a person in front of the screen can be sensed using associated sonar sensors.

[11]How emotions arise and are expressed is described in Chapter 6.

specificity, so it is chosen. When the precondition evaluates to true it matches the user-controlled Woggle, and that binding is used in the body of the behavior. If this behavior fails, the other behavior could then be attempted, allowing Wolf to try to play with another Woggle.

The body of this sequential behavior has three steps: a mental act to compute and remember some data for the behavior; a subgoal to get the chosen Woggle to play; and a subgoal to lead in the game. The behavior is placed in the tree along with the first step, which is then chosen for execution. It is executed. As all mental actions, it succeeds after the code is executed. This causes it to be removed from the tree and the next step in the behavior is placed in the tree. This step is a goal with the name `get_follower` and a value which is the agent matched in the precondition. This goal is chosen for execution and the only behavior for it is chosen, and instantiated for it. This is a sequential behavior that goes to the chosen Woggle, invites him to play, and succeeds or fails based on his response. This behavior also has a context condition that causes it to fail if the behavior takes too long to execute. The context condition is intended to capture Wolf's impatience in these matters.

The goal to go to the chosen Woggle is expanded, eventually resulting in a **Jump** action being issued to the body to be executed. At this point, the thread of behavior is suspended waiting for the **Jump** to complete.

Meanwhile, a success test for a wait step fires in the behavior for another top level goal. This behavior is the one that makes the Woggle periodically blink. The success test fires when enough time has elapsed since the behavior has been instantiated. (The time of instantiation is recorded in the argument to the behavior.) This behavior is a three step behavior: a wait step with the time based success test; the primitive action to close the Woggle's eyes (**CloseEyes**); and a primitive action to open the Woggle's eyes (**OpenEyes**). Since the success test just fired, the wait step is removed, and the **CloseEyes** action is placed in the active behavior tree. Since this action doesn't conflict with the executing **Jump** action or any of the executing goals, it is chosen for execution and sent to the body to be executed in parallel with the **Jump**.

The goal to blink is a persistent goal, so it resets after completing a blink, again ready to wait and blink. It also conflicts with the agent's goal to sleep. This prevents the agent from blinking while sleeping.

While moving through the air the **CloseEyes** action completes; the **OpenEyes** action is chosen and executed; and that action also completes. The sequential behavior then succeeds since all of its steps succeed. This causes the blink goal to succeed, but since it is marked persistent, it resets, as described above, and remains in the tree. It is then chosen for execution again, and the same behavior is chosen for it. The wait goal is again in the tree with a precondition waiting for time to elapse.

After finishing blinking the agent is still flying through the air performing its **Jump** action. In the partial agent with only three top level goals, there is nothing more that Hap can execute for part of this **Jump**, so no Hap processing happens for part of this **Jump**.

Toward the end of the jump, Wolf recognizes that a fight is going on. This causes the demon in the behavior for `respond_to_fight` to fire. This goal is higher priority than the `amuse_self` goal, so its steps are chosen for execution before those of `amuse_self`. The body of the behavior is a `par_cond` expression. The first clause is a condition that is true one third of the time (using a random number generator). The body of this clause is a mental action that marks the fight data structure as responded to. If this clause is chosen, Wolf will have no reaction to the fight and the other behavior to play will continue. The second clause is a default clause, so it will be chosen if the first clause is not. Its body is the same mental action followed by the goal to gang up on the agent being attacked. (The information about who was being attacked is provided by the behavior that recognized the fight. This recognition cannot be performed by primitive sensors, because what is being recognized is a pattern of activity over time. How such recognizers are written is the subject of Section 5.2.)

The second clause is chosen, the mental act is executed, and the subgoal `gang_up_on` is chosen and expanded. This goal conflicts with the goal to play follow-the-leader, and is higher priority, so that goal and all of its children are suspended. Any executing actions of a suspended behavior are aborted if possible, but in this case the body is flying through the air so aborting is not possible.

The `gang_up_on` goal is further expanded, eventually resulting in Wolf jumping toward the ongoing fight. This **Jump** action is queued behind the executing **Jump**, enabling the motor system to create an appropriate landing for the current **Jump**.

To summarize the behavior that is observed in this example, Wolf chooses to pursue his `amuse_self` by starting a game of follow-the-leader. He chooses the User to invite, and starts jumping toward him by issuing a **Jump** action. As he is flying through the air with his first jump, he blinks in parallel because of a `blink` goal in a concurrent behavior. About a third of a second later, he notices a fight in another part of the world. (How this is recognized is the subject of Section 5.2.) This recognition causes Wolf to have a goal to gang up on the fight. This goal is higher priority and conflicting with the `amuse_self` goal, so the `amuse_self` goal and all of its children are suspended. The **Jump** can not be suspended because Wolf is flying through the air. The new `gang_up` goal is expanded, eventually giving rise to a **Jump** action toward the fight. This is queued after the executing **Jump** action, allowing the motor system to create an appropriate landing for it, and Wolf is ready to head toward the fight. The total elapsed time is a little over a second.

4.16 Summary

In this chapter I have described Hap, my architecture for believable agents, in detail. The following chapters describe how additional capabilities of an agent can be built in Hap, and how all of these can be used together for the creation of a complete believable agent.

Chapter 5

Hap As a Unified Architecture

Now that Hap has been described in detail we can turn our attention to how it is used to implement the additional capabilities of the agent. In this and the coming chapters I describe four capabilities: arbitrary computation ("thinking"[1]), composite sensing, emotion, and natural language generation.

Many agent architectures build such capabilities as separate modules. The approach I take is to tightly integrate them: I use Hap's mechanisms, originally designed for action generation, to support each of these capabilities. In so doing, I use Hap as a unified architecture for believable agents in that all[2] aspects of a believable agent's mind are implemented in the Hap architecture and language. I take this approach for a number of reasons. First, the requirements for believability apply equally well to these capabilities as they do to action producing behaviors. Artists, and audiences looking for believability, don't partition characters along the lines that artificial intelligence typically does; they expect all aspects of the character or agent to satisfy the needs of believability. For example, a believable agent cannot be true to its personality when acting, but not when generating language or "thinking". The approach I take for addressing the requirements of believability relies heavily on properties of Hap, and it seems advantageous to use them.

A possible side effect of this approach is that Hap many not be up to the task. It is not obvious that an architecture that was created for producing action can be used in a natural way to generate language or arbitrary computation. When viewed as a research strategy, I have viewed this as an opportunity rather than a problem. Using Hap to express these widely varying aspects of an agent's mind is an opportunity to learn where it is deficient, and then to change the architecture to fix these deficiencies. The hypothesis is that enhancements that are useful to one part of the mind are probably useful to other capabilities as well. My experience along this path has reinforced this hypothesis. Many of

[1]Arbitrary computation is useful for expressing many aspects of an agent's thinking. These can range from simple computations or inference, to larger computations, for example path planning. The ability to express such computation is useful when building an agent. It is only part of the "thinking" that is necessary, however. Hap's expansion of goals from motivation to primitive actions, the generation of emotions, and other processing of the architecture are each also a type of thinking.

[2]This list of four capabilities is not intended to be exhaustive of the capabilities needed for believable agents. Section 11.3 mentions how natural language understanding might be expressed in Hap, for example.

the features of Hap described in the previous section arose because of the particular needs of specific capabilities. After these features were added to Hap, they have been widely used in all aspects of agents.

In addition to the above, I believe there are three concrete advantages to implementing these capabilities in Hap.

First, by implementing them in Hap we avoid damaging the responsiveness and real-time properties of the rest of the agent. If one were to implement large computations, such as the planning process of language generation, in C and execute them in the same process as the rest of the mind, the language computation would interrupt Hap's execution loop for the duration of the computation. The agent would essentially freeze. No demons, context conditions or success tests would be able to fire. Goals that are higher priority would not be pursued. Actions that succeed or fail would not be recognized by the mind, and so on. This would destroy any suspension of disbelief that might otherwise be present. Creatures in the real world and in art simply do not arbitrarily freeze either physically[3] or in their mental thinking and reacting.

There are other solutions to this problem, such as running other desired computations as separate operating system processes or threads. This approach would solve the problem of not hurting Hap's real-time properties but would lose out on the next two advantages.

The second advantage to implementing these capabilities in Hap is that they inherit the architectural support for believability that Hap provides. This includes support for reactivity, situated variation, concurrent pursuit of goals and parallel action, emotional variation (described in Chapter 6), and other support. An analysis of this support is given in Chapter 9.

The third advantage to using the same language and architecture for all aspects of the agent is that it may make it easier to tightly integrate the varied aspects of mind. Strong integration is one of the requirements for believability that has not yet been addressed.

In the rest of this chapter and the next one, I describe the first three of these capabilities, arbitrary computation, composite sensing and emotion, in turn and describe how they are implemented in Hap. Chapter 7 gives examples of how they are used together to build a complete agent. Finally, Chapter 8 describes steps toward integrating natural language generation with the rest of an agent by allowing its direct expression in Hap.

[3]This should not be confused with "holds" in animation, which seem to be a form of freezing of the action. When looked at carefully, this concept from animation is a perfect illustration of the damage to believability that comes from freezing the action completely. There is a need in animation for freezing the action: "When a careful drawing had been made of a pose, it was held without movement on the screen for a few frames . . . to allow the audience time to absorb the attitude". But this comes with a cost in the believability of the character: "When a drawing was held for that long, the flow of action was broken, the illusion of dimension was lost, and the drawing began to look flat". To avoid this loss to believability, Disney found a way "to 'hold' the drawing and still keep it moving!". They did this by developing the idea of "moving holds" that keeps the character moving for those few frames between two poses that both capture the same attitude. This gives the audience time to absorb that attitude while not freezing the action and thus destroying the illusion of life [Thomas and Johnston 1981, p. 61].

5.1 Arbitrary Computation

Hap allows arbitrary computation in primitive mental actions, and this is an important place where many forms of "thinking" of the agent occur. This computation is expressed in an extended version of C that among other things allows the code to access Hap variables. As described in the previous chapter, by policy these mental actions are restricted to small, bounded computations. The reason for this policy is to avoid the disruption of Hap's real-time properties described above.

When large or unbounded computations are desired in a Hap agent, one can build these computations using Hap goals and behaviors.

5.1.1 How

When building computations in Hap, the data structures and primitive operations are defined by C. Small units of computation are written in mental actions. To be able to build large computations out of these small, bounded chunks we need reasonable control structures and the ability to communicate between the chunks. In this section I first describe how one can communicate between the chunks of computation, and then describe how one can express such control structures in Hap. The control structures I describe are sequencing, conditionals, loops and functional abstraction in Hap.

The point of this section is not to prove that Hap is Turing-complete, but rather to suggest that arbitrary computation can be naturally expressed in Hap. This is not something that can be proven, but I hope that the following descriptions and examples are suggestive of it. Further evidence is provided by the descriptions of computations built in Hap in the next three chapters. These include aspects of the emotion system in Chapter 6, aspects of a complete agent in Chapter 7 and aspects of natural language generation in Chapter 8.

Communication Between Computational Chunks

In each of these control structures, communication between chunks of computation is through normal Hap variables, each of which can have any needed C value as its value.

As described in Chapter 4, variables are available in Hap in four different contexts. *Global variables* are specified along with the behaviors and initial goals of the agent. Any behavior can reference or set these variables. *Parameters* are specified for a particular behavior and are given initial values when the behavior is chosen to accomplish a particular goal. The values present in the goal are bound to the formal parameters in the behavior. These variables are lexically scoped, and are therefore visible to any expression textually within the behavior. *Local variables* and *dynamic local variables* can also be specified within a behavior. Local variables are lexically scoped. Dynamic local variables have dynamic scope and can therefore be referenced or changed by expressions that are created as a result of this behavior. As in all normal programming languages with these features, global variables and dynamic variables should be used with care to avoid programming errors. Hap variables are accessed by the expression `$$<variable name>`. This expression can be arbitrarily embedded in C code within Hap behaviors.

Sequencing

Sequencing of mental actions (and therefore the C code that they contain) is possible by placing multiple mental actions in a normal sequential behavior. The sequential behavior causes each of its steps to be evaluated in turn. Notice that local variables can be declared in the body of the sequential behavior, as is described in Section 4.14.1.

Functional Abstraction

Functional abstraction is possible by using Hap goals and behaviors. The function to be performed is placed in a behavior with a unique name. Any arguments to this function are placed as arguments to the behavior. This behavior can then be "invoked" by using a goal with the same name. Actual parameter values to correspond to the formal parameters in the behavior are specified by the values in this goal. Note that using behaviors and goals in this way as functions and function invocation is appropriate as long as one only writes a single behavior for the goal.[4] If one writes multiple behaviors, Hap will choose among them at execution time.

Conditionals

Writing multiple behaviors for the same goal is the method used for writing conditionals in Hap. The test conditions are each placed in the precondition of a behavior. The code to be executed if the condition is true is placed in the body of the behavior. Any information that is needed to evaluate the conditions or body code is passed as values of the goal. Unlike conditionals in some languages, in this encoding all of the test conditions are evaluated in parallel. If multiple test conditions evaluate to true, Hap chooses arbitrarily among them.

The `par_cond` form, described in Section 4.14.2, is a syntactic extension for this expression. When it is used, the author loses some of the opportunities of expression that are possible with the direct encoding via goals and behaviors. Thus, both are useful, and the choice of which to use depends on the needs of the author.

Looping

Looping in Hap can be performed through two mechanisms. First, recursion can be used by using functional abstraction as described above. It should be noticed that such recursion is not properly tail-recursive; the active behavior tree will grow with each recurrence. Looping

[4]Writing more than one behavior for a goal can be thought as overloading operators, in the object-oriented programming sense, if the conditions in which different behaviors apply are disjoint. This can also be used for conditionals as the next section addresses. When the conditions in which the different behaviors apply are not disjoint, the semantics is more complicated as described in detail in Chapter 4.

without growing the active behavior tree is possible by using a persistent annotation. This will create an infinite loop. To add an exit condition, one can use a success test on the invoking goal.

Using these control structures one can break any large or unbounded computation written in C into nearly arbitrarily small computations[5].

Example

To illustrate how one can express arbitrary computation in Hap, consider the simple example of writing a function to compute factorial[6] of a number.

```
(sequential_behavior factorial (n)                              (1)
       (locals (result 1))                                      (2)
  (with (success_test is n zero?)                               (3)
    (sequential                                                 (4)
      (with persistent                                          (5)
        (sequential                                             (6)
          (mental_action "$$result = $$result*$$n;")            (7)
          (mental_action "$$n--;")))))                          (8)
  (return_values $$result))                                     (9)
```

FIGURE 5.1: Iterative behavior for computing factorial written in Hap.

One can write this function in Hap in an imperative style by using the constructs for sequencing, functional abstraction and the second type of looping describe above. Such a behavior is shown in Figure 5.1. The lines are numbered in the right margin for easy reference in this description. Lines (6) through (8) are a simple example of a sequentially ordered computation, that performs one step of the loop for computing factorial. You will recall from Section 4.14.2 that the `sequential` form expands to a goal with a unique name and a single sequential behavior for that goal with the two steps as its steps. This expanded behavior is treated as if it is in the lexical scope of the original behavior. Thus it can access or change the parameter `n` and the local variable `result`. Line (5) causes an infinite loop of this goal and behavior, and line (3) and (4) turns this infinite loop into the equivalent of a while loop. The extra sequential plan is necessary to allow the persistent goal to be removed. Remember that a persistent goal can only be removed if one of its ancestors

[5]One is ultimately limited by the granularity of a unit of computation that is expressible in C. Practically, we have found no reason to break things up smaller than reasonable expressions of several statements in length or small, bounded loops.

[6]factorial of a positive number `n` is defined as the product: `1 * 2 * 3 * ...* (n - 1) * n`.

in the active behavior tree is removed. Line (2) declares and allocates a local variable to accumulate the result of the computation, and line (9) causes the accumulated result to be returned when the behavior succeeds. Line (1) encapsulates this computation in a behavior that can be used like a function. To execute this function on a particular number, the goal `(factorial num)` is placed in the ABT.

```
(sequential_behavior factorial (n)                          (1)
       (precondition is n zero?)                            (2)
  (return_values 1))                                        (3)

(sequential_behavior factorial (n)                          (4)
       (precondition is n greater than zero?)               (5)
  (with (bind_to result)                                    (6)
    (subgoal factorial "$$n-1"))                            (7)
  (return_values "$$result * $$n"))                         (8)
```

FIGURE 5.2: Recursive behavior for computing factorial written in Hap.

Such a computation could also be written recursively in Hap, as is shown in Figure 5.2. A conditional to decide between the base case and other cases is encoded in the two behaviors and their preconditions. The first behavior expresses the base case of the recursion, and the second behavior expresses the recursive case. The recursive call is a subgoal with appropriate arguments as shown in line (7).

5.1.2 Discussion: Thinking in Hap

A common tendency when presented with mappings such as those above is to then think solely in terms of the familiar metaphors and to ignore the unfamiliar metaphors of the underlying language, in this case, Hap. While useful for getting started in expressing computations in this language, it is a dangerous tendency for the long term use of Hap for two reasons. First, thinking in terms of loops, conditionals and functions instead of behaviors, goals and reactive annotations can lead to errors because the meaning is actually grounded in the latter. If one thinks only in terms of the familiar concepts, it is easy to forget that the tests in conditionals are evaluated in parallel, and that unpredictable things happen if multiple tests are true simultaneously. In the behaviors in Figure 5.2 both a test for zero and a test for greater than zero are necessary.

The second reason it is important to think in terms of the underlying Hap language is to capitalize on opportunities of expression. These opportunities can be important for even the most mundane of computations. For example, consider the task of path planning. Because the test conditions in a conditional are actually preconditions, Hap's full range of primitive sensors and reflection can be included in these expressions. Thus, when building a path planner one could (and probably should) choose different routes when energetic than when tired, or make other variations based on the sensed environment or internal state of the agent. A path planner could also be fitted with reactive annotations that abort parts of the planning when the goals of the agent or other information change. These types of expression are central to being able to capture a personality in code, and this is an important benefit to expressing such computation in Hap.

5.2 Composite Sensing

As described in Chapter 3, the agents in the sample Woggles domain can perceive the world at a low level. To perceive another agent's behavior, an agent can perceive the physical properties of the other agent's body and the actions that the other agent is performing at the current instant in time. The physical properties that are perceived are physical location, angle the body is facing, and direction the eyes are facing. In addition, the set of currently executing primitive physical actions can be perceived. The agents perceive the shape of the world by querying positions in the XY plane for associated heights.

In order for the agent to react appropriately to relatively large scale events like fights, sleeping and moping, the available stream of low level perceptions must be interpreted by the mind of the agent. In general, these large scale events are recognized because of patterns of perceptions over time. Of course, how patterns of perception are interpreted is subject to the social conventions of the created world, as well as the individual personality of the agent. For example, different cultures perceive different things as threatening or friendly, and different individuals within a culture also vary in these perceptions.

In Hap, simple sensing is done through primitive task-specific sensors as described in Section 4.13.3. With these sensors an agent can query any of the above listed low-level events. To recognize larger scale events one builds sensing behaviors in Hap.

These recognizers take advantage of normal Hap mechanisms, such as demons, success-tests and context conditions, as well as Hap's mechanisms to encode patterns of behavior. The combination results in behaviors that recognize patterns of events over time.[7]

5.2.1 Example Sensing Behavior

To make this approach concrete, consider a specific example sensor: one to recognize threats in the Woggles' world. Threats in the Woggles' world are similar to physical threats in our world. The Woggles have a notion of personal space, with invasion of that space causing some uneasiness in many situations. An agent making quick movements toward another agent can be viewed as threatening that agent, especially if the movements are made within the other agent's personal space. In addition, the Woggles have a particular

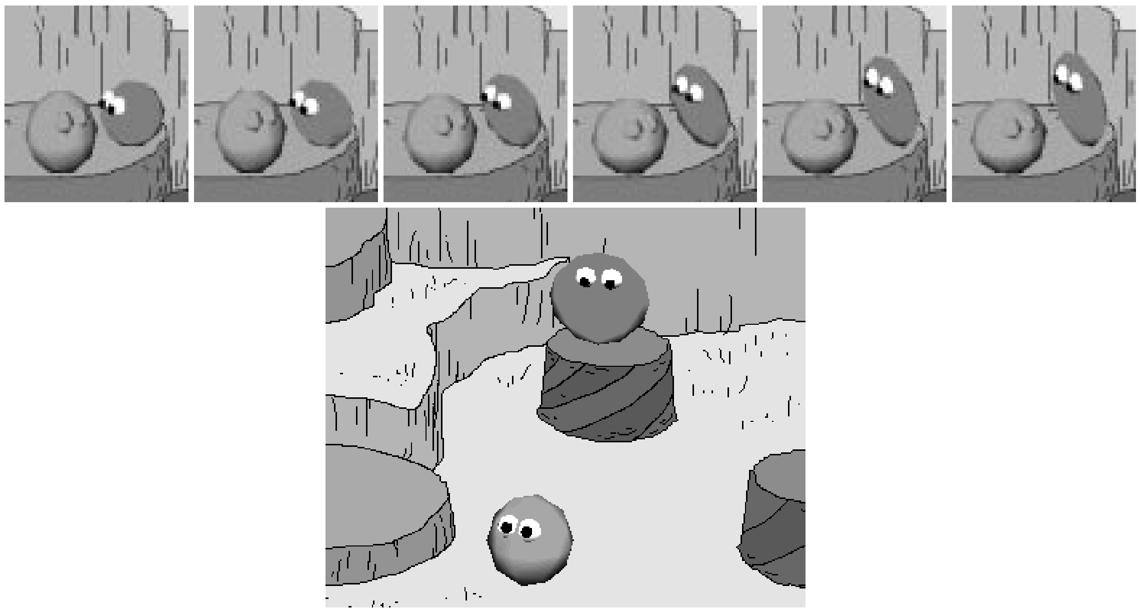


FIGURE 5.3: A sequence of snapshots showing the appearance of a puff movement over time, and a single snapshot showing the appearance of a puff from head on.

body motion that is used for threatening. An image of this motion, called *puff* can be seen in Figure 5.3. It is modeled after the way animals make themselves look larger to the entity they are threatening, for example, the way a cobra spreads its hood, a cat arches its back, or a human puffs out its chest. Of course, these social conventions are subject to each agent's interpretation, and misinterpretations do happen.

Given that these types of threats are made both by the user and by other agents, each agent needs to be able to recognize when it or another agent is being threatened. The fact that the user performs these threats means that we cannot just sense the information from the mind of the other agents.

A behavior to perform this recognition is presented in Figure 5.4. The basic pattern it recognizes is when an agent enters the personal space of the potential victim, and either does multiple quick movements or puffs at the victim. A complete description of it follows.

To construct a sensor to recognize threats of this form, we first need a demon that recognizes when a Woggle comes too close to the agent. This is done, as shown in lines (1) and (3)–(6) of Figure 5.4, by using a sequential behavior whose first step is a wait goal with an appropriate success test. In this case, the success test uses primitive sensors to recognize when a Woggle comes within a certain distance (less than this agent's notion of personal space) of the target. The agent's current notion of personal space is stored in a

[7]This approach does not address the orthogonal and difficult research issue of interpretation of ambiguity in the face of multiple possible interpretations.

```
(sequential_behavior recognize_threaten (victim)
       (locals (aggressor unbound))
  (with (success_test someone is within $$personal_distance of $$victim and
                      $$victim can see someone;
                      set $$aggressor to someone)
    (wait))
  (subgoal verify_threaten $$aggressor $$victim)
  (mental_act signal $$aggressor is threatening $$victim now))

(concurrent_behavior verify_threaten (attacker victim)
       (number_needed_for_success 1)
       (context_condition $$attacker is within 4*$$personal_distance of $$victim and
                          less than a minute has elapsed)
  (subgoal recognize_puff $$attacker $$victim)
  (sequential
    (subgoal recognize_quick_act $$attacker)
    (subgoal recognize_quick_act $$attacker)
    (subgoal recognize_quick_act $$attacker)))

(sequential_behavior recognize_quick_act (who)
  (with (success_test $$who performs a Squash, Put, Jump or ChangeBodyRadii action
                      with a duration < .2 seconds)
    (wait)))

(sequential_behavior recognize_puff (who at)
  (with (success_test $$who is executing ChangeBodyRadii action with argument
                      values in the range needed to look like a puff)
    (wait)))
```

FIGURE 5.4: Sensing behavior to perceive threats in the Woggle's world.

global variable. This success test, as a side-effect of firing, records which Woggle came too close in a local variable of the behavior. The second step of the sequential behavior, in line (7), is a goal to verify the threat by that Woggle. And the third step, in line (8), is a mental act to record that that Woggle is threatening the target. If the verify goal fails, the behavior fails, and the third step is not executed. If it succeeds, the third step records the aggressor and other behaviors can match that information to decide how to react.

To construct a behavior to verify the threat we need to be able to recognize quick physical actions and a `puff` action toward the victim. These are expressed as the last two behaviors in Figure 5.4, lines (18)–(25). They are both sequential behaviors with a single `wait` step. Thus, these behaviors can only succeed if the success test associated with the `wait` step becomes true. Otherwise they will remain in the tree until one of their ancestors is removed. The success tests for these behaviors recognizes the appropriate actions as they happen. The first one, in lines (19)–(20), recognizes when any of four actions that move the body is done quickly (with duration less than 200 milliseconds) by `$$who`. The success test for the `recognize_puff` behavior, in lines (23)–(24), recognizes when `$$who` performs a **ChangeBodyRadii** action with parameters in an appropriate range to make it look like the puff action in Figure 5.3.

Using these recognizers one can build a behavior to verify that a threat is being made. One such behavior is shown as `verify_threaten`, in lines (9)–(17). It takes the alleged attacker and victim as its arguments. Since a threat can be either a single puff or a sequence of several quick actions toward the victim, it uses a concurrent behavior. This allows all steps of the behavior to be pursued concurrently; the first step, in line (13), is a `recognize_puff` goal, and the second step, in lines (14)–(17), is a `sequential` form to cause a sequential behavior with three `recognize_quick_act` steps. In this way both types of recognition can be pursued concurrently.

Since only one of these patterns is required for a threat, we need to indicate that only one of the steps of our `verify_threaten` behavior is needed for the behavior to succeed. This is the `(number_needed_for_success 1)` annotation in line (10).

Now we have most of the recognition behavior written. But for a threat to actually take place the threatening actions need to happen near the victim, and if the threat is of the form of a sequence of quick actions they need to be reasonably close together in time. If the aggressor moves into the victim's personal space and then moves on, then no actual threat takes place and the sensing behavior should recognize this. Similarly, if the aggressor moves into the victim's personal space and stays there without performing threatening actions or performs a sequence of quick movements over the course of a long time, then no threat takes place (although such activity should probably be recognized as annoying). The context condition in lines (11)–(12) is included to capture this idea; it encodes the conditions in which the `verify_threaten` behavior continues to make sense: when the attacker stays reasonably near to the victim and does not take too long to threaten. If it moves farther away or takes too long, then the behavior will fail, and no threat will be signaled.

An agent enables this behavior by creating `recognize_threaten` goals for whichever agents it desires to monitor. Typically an agent will have a persistent goal to continuously monitor whether it itself is threatened. Some agents have persistent goals to recognize when their friends are threatened. And any agent can create such a goal whenever the situation warrants. For example, if an agent you are trying to cheer up tells you it is sad because it keeps being threatened, the behavior that responds to this knowledge might be to watch for such threats in the future and intervene. Part of such a behavior would be a `recognize_threaten` goal. Additional behaviors must be written to react appropriately to the results of this goal and behavior.

It should be noted that these sensing behaviors typically are not very expensive computationally. Whenever such goals are present in the ABT, and sufficient processing has occured to allow the goal to be chosen and expanded, the behavior will sit suspended until an agent enters the personal space of the one being monitored. The success test that is monitoring this condition will not fire until such an event occurs, and the `wait` step, like all `wait` steps is never chosen for execution. Thus this portion of the ABT remains unchanging until the success test condition fires.[8] While suspended, the behavior takes no Hap processing. Hap instead attends to and expands other goals and actions of the agent.

Unless agents are repeatedly threatening each other or performing actions that cause the behavior to partially recognize a threat, this behavior is computationally cheap.

5.2.2 How It Executes

When an agent enters the personal space of the agent being monitored, the success test fires, and the `verify_threaten` goal is available in the active behavior tree to be chosen. Thus the agent's attention is drawn to a potential threat, and this behavior decides whether one is in fact taking place. This behavior watches the subsequent actions to see if either a sequence of three quick actions or a puff is performed at the potential victim. If either of these happens the `verify_threaten` behavior will succeed. Only one is necessary because of the `number_needed_for_success` annotation. This success will cause the `verify_threaten` goal to succeed and the following mental action will signal the recognition of the threat. Any other behaviors in the agent can then respond to this information.

Notice that the recognition of the puff action and three quick actions can take place or partially take place in any order. This is because there are two goals looking independently for the puff and three quick actions. So for example, if a quick action is followed by a puff the recognition occurs properly. The quick action would cause the sequential behavior to progress to the second `recognize_quick_act` goal, and the puff action would cause the `recognize_puff` goal to succeed, causing the `verify_threaten` behavior to succeed and the threatening to be signaled.

If neither three quick actions nor a puff happen before the potential attacker moves away from the victim, the behavior will fail because of the first clause in the context condition. This will cause the `recognize_threaten` behavior to fail as well, and the threat will not be signalled. If the `recognize_threaten` goal is persistent it will reset itself, and be available

[8]The monitoring of the conditions themselves is cheap because of the compilation to Rete.

to recognize a threat or potential threat again. The same will happen if the potential attacker takes too long to threaten.

This behavior illustrates how sensing behaviors can be written in Hap. Similar sensing behaviors are written to recognize sleeping, when another agent is sad, invitations to play, teasing, the moves to be followed in a game of follow the leader, etc. Such behaviors are an important part of building a believable agent.

5.3 Summary

This chapter has presented how two capabilities, arbitrary computation (“thinking”) and composite sensing, needed for agents can be expressed in Hap. This is part of the description of how Hap functions as a unified architecture for all aspects of a believable agent’s processing. Chapters 6 and 8 describe how additional capabilities of an agent are built in Hap, and in Chapter 9, I describe the advantages of using Hap as a unified architecture.

Chapter 6

Emotion and Integration with Hap

As is described in the requirements for believable agents in Chapter 2, emotion is a critical part of believable agents. What a character is emotional about and how it shows those emotions is one of the central defining categories of details that make up the personality of the character. This idea is widely held in character-based arts such as character animation, drama, and film. The book *Disney Animation: The Illusion of Life* claims:

> To bring a character to life, it is necessary to imbue each single extreme drawing of the figure with an attitude *that reflects what he is feeling* or thinking or trying to do. [Thomas and Johnston 1981, p. 146, emphasis added]

This claim is not an isolated one. Thomas and Johnston repeatedly make this point as do almost all authors when writing about creating characters.

To address this need, I and the other researchers of the Oz project use an explicit, but integrated, emotion system. This is a deliberate choice; others working in emotion for agents take different approaches, such as emotions emerging from other properties of the architecture [Sloman 1987]. We chose to use an explicit emotion system because of the flexible artistic control that it affords. Such artistic control has been useful for creating believable agents. The nature and advantages of this approach are discussed in depth in [Neal Reilly 1996].

6.1 Desired Properties of Emotion System

During the construction of different agents, I have used different emotion system versions. Each of these versions has had different properties with their own advantages and disadvantages. In my experience, there are four properties that are most important to an emotion system's usefulness.

- **Personality-specific and flexible generation of emotions**: The author needs to be able to specify what this agent becomes emotional about and how emotional it becomes, because these details are a critical part of expressing a personality. The method of generating emotions needs to be flexible, because artists might conceive of many ways in which they want emotions to arise in the agents they create. I do not want the architecture to irrevocably limit them to some preconceived notion of how this is done. However, some limits that can be worked around when necessary might be needed in order to achieve the other three properties.

- **Flexible expression of these emotions**: Different characters show their emotions in dramatically different ways. I want to allow a wide range of modes of expression in my architecture, so that authors can create agents that richly express their emotions.

- **Automatic management of emotions and framework to support their expression**: Insofar as it does not overly conflict with the above two properties, it is useful for the management of emotions to be automatic. For example, an automatic mechanism for storing and decaying emotions once they are generated removes a burden from the agent builder. Likewise, a structured framework for expressing the emotions can be useful if it is not overly restrictive.

- **Understandability of generated emotions**: If an emotion system provides the above three properties, then an author is able to specify to what his agent becomes emotional, and how it expresses those emotions. For this to work, the author must understand the emotions that are automatically generated using the first information he provides, so that he can create appropriate expressions of those emotions. This is complicated by any automatic management that the system provides. The author needs to intuitively understand the ways emotions are automatically generated, and the ways they change over time in order to be able to create expressions of those emotions appropriate to the personality being constructed.

The next sections describe my approach to providing support for emotions in believable agents. I first describe in Section 6.2 the model of emotions used in Hap. This model is the work of Scott Neal Reilly. The rest of the chapter describes how this emotion system is integrated with Hap. This integration is joint work of Neal Reilly and myself. (Neal Reilly's dissertation [Neal Reilly 1996] describes an extended version of this model, and also includes an approach to social behavior for believable agents.)

Section 6.3 addresses the first desired property above: how an author specifies what an agent becomes emotional about and how emotional it becomes. Section 6.4 addresses the next two desired properties: flexible expression of emotions and appropriate automatic management. Section 6.5 addresses the final desired property: the understandability of the resulting emotions. Finally, Section 6.6 describes how the emotion system itself is implemented in Hap to capture properties I believe important for believable agents.

6.2 Neal Reilly's Emotion Model

To enable these properties in my architecture I use an emotional system developed by Scott Neal Reilly called Em. The full Em system is described in Neal Reilly's thesis [Neal Reilly

1996]. In this chapter I describe a simplified, but representative, version. I describe it from the point of view of its integration with Hap, and the way in which this facilitates the expression of detailed envisioned personalities. My primary purpose in describing it is to discuss emotion issues here and in later chapters. The version of Em I describe is the same one implemented and used in the Edge of Intention agents described previously.

Neal Reilly's emotion model takes a cognitive approach to emotion generation. It draws in part from the work of Ortony, Clore and Collins [Ortony *et al.* 1988]. In this model emotions arise from events and the relationships of these events to important goals of the agent. For example, happiness might arise in a competitive child because he just won a game and winning the game is an important goal for him. In the simplified version of Em that I describe here, there are five possible emotion types: **happiness**, **sadness**, **fear**, **gratitude**, and **anger**. **Happiness** arises when a goal important to the agent succeeds. **Sadness** arises when an important goal fails. **Fear** arises when the agent thinks an important goal is likely to fail. **Gratitude** and **anger** are directed emotions. An agent feels **gratitude toward X** when an important goal succeeds and the agent believes someone or something (**X**) helped it succeed. Likewise an agent feels **anger toward X** when an important goal fails and the agent believes **X** caused or helped cause its failure. **Fear** can arise as an undirected emotion, as above, or as a directed emotion; **fear of X** arises when the agent believes **X** will be the cause of an important goal failure. For each of these emotions the intensity of the emotion is directly related to how important the goal is to the agent.

This emotion model also contains a social *attitude*: **like**. Each **like** attitude contains a direction which is the object or agent that the attitude is directed toward, and a numeric value which expresses the level of **like**. Negative values indicate dislike toward the object or agent.

6.3 Emotion Generation

Our first requirement for the emotion system is that an author be able to specify what the agent becomes emotional about and how emotional it becomes. Since we are using a cognitive model of emotion in which emotions are generated from goals of the agent, we must first allow our authors to specify which goals are important and can therefore cause emotions. This is done by adding an optional *importance* annotation to a goal. An importance annotation is added using the form `(importance <expression>)`, where `<expression>` must evaluate to a non-negative number that represents how important this goal is to the agent.

If an author gives a goal a non-zero importance value, that goal will automatically generate **happiness** and **sadness** emotions when it succeeds or fails. It can also generate **anger**, **gratitude** or **fear** emotions under conditions which I describe below. The intensity of the generated emotions is directly proportional to the importance value given by the author. If no importance annotation is given, the importance of the goal defaults to zero and no emotions will be generated for it.

In Hap the importance and priority of a goal are specified separately. This is because, for many goals and personalities, the emotional reactions an agent has to its goals are different from how urgently the agent pursues the goals. Consider, for example, a child who is told to take a bath before going to bed, and who then trudges off to the bathroom only to find that there is no hot water. Most of the kids I know when faced with this turn of events would not be sad because their high-priority goal failed, but would instead get a big grin on their face because they are happy not to have to take a bath. In such a situation, it is appropriate for the goal to take a bath to be high priority even if the child does not want to take a bath, but it is not appropriate for the goal to have a high importance value. In fact it would likely be appropriate to have the reverse goal (to not take a bath) have a high importance.[1] By separating a goal's priority from its emotional importance (and in this case by using a *passive goal* as described below in Section 6.3.3), personalities such as the above child can more easily be built.

Importance annotations are similar to priority annotations in that they are specific to instances of a goal rather than to all goals of a particular type. This is for the same reason as given for priority: a very different emotional importance value may be appropriate for the same goal in different contexts. For example, the goal to run through the park might have very different importance values depending on whether it is part of an exercise behavior or a race behavior. Importance values are different from priorities in that they are not inherited from parent goals. This is because, for most personalities, it would not be reasonable for all subgoals of an important goal to also generate emotion. If winning a tennis match is an important goal, most competitors would not feel the same emotional intensity for every point that is won or lost. (Particular personalities, such as one based on John McEnroe, might have some emotional response to every point, but even for him the responses would likely not be the same as for the entire match.) Except for the generation of emotions, an importance annotation has no functional role in a Hap agent.

6.3.1 Inferring Credit, Blame and Likelihood of Failure

To generate **anger**, **gratitude** or **fear**, the agent needs to have other knowledge in addition to the fact that an important goal has succeeded or failed. For example, in order to generate **gratitude** in Neal Reilly's model, the agent must believe that someone or something helped or is responsible for an important goal's success. Likewise, generating **anger** requires determining that the agent blames another agent or thing for an important goal failure, and generating **fear** requires the agent believing that an important goal is likely to fail. Hap's normal processing does not automatically provide this information.

The primary reason Hap does not automatically compute this information is that there is no single mechanism to compute it; it is personality-specific. When and how an agent blames others for failures is specific to its personality, as is the agent's assignment of credit for goal successes and how the agent infers that a goal might fail. For example, some personalities are very cavalier in blaming others for failures, whereas other personalities

[1] Such a goal would need to not be pursued, or be pursued carefully for the child to stay out of trouble with the parent. Goals that cause emotional reactions but are not necessarily pursued are described in Section 6.3.3.

tend to blame themselves even in the face of strong evidence to the contrary. In addition, such generalities are not even enough to fully capture a given personality's variety in this type of reasoning. How credit and blame are assigned almost certainly depends on the specific situation. An individual that is stressed may make different choices than one that is calm or sad, and an individual might make different choices when assigning blame for familiar goals versus novel goals. Similar variation exists when a particular personality is deciding whether a goal is likely to fail. In short, this reasoning is yet another arena for the detailed expression of personality.

This type of detailed expression of personality is one of the things Hap was specifically designed to support. There are a number of opportunities in Hap to encode this type of reasoning for the agent being built. The first is available through reflection of Hap's processing. The active goals and behaviors of the agent can be referenced by any of Hap's decision making match expressions: preconditions, context conditions and success tests. In addition, when an important goal succeeds or fails, this event is recorded in memory, so that it is available to be referenced by these conditions. Using this information about which goals have succeeded or failed, behaviors can be written to infer blame or credit about these events. Using reflection of which goals are active, behaviors can be written to infer likelihood of failure for these goals. These behaviors can use information about the goals, as well as the other sources of information available to all Hap behaviors: sensing of the external world, reflection of internal state, etc. Such inference behaviors can be written using any inference technology desired by the author. Chapter 5 describes how one can express such computations in Hap.

Once this information is known by the agent, the emotion system automatically generates **anger**, **gratitude** or **fear** emotions as appropriate.

One simple example of these emotions being generated arises during the game of follow-the-leader in Wolf, the aggressive personality from the Woggles. If the game is aborted before it is finished, Wolf has a simple special-purpose inference behavior that blames the agent he was playing with for the failed goal. This is completely appropriate for Wolf's self-centered personality: in his mind he could not have caused the failure, so it must have been the other guy; there is no reason to think further. The failed goal by itself gives rise to **sadness**, and the failed goal combined with the inferred assignment of blame gives rise to **anger toward** the other agent.

Explicit reasoning behaviors to infer blame or credit are not always necessary. This is because an author will often have expressed this content in other aspects of the behavior. Success tests, for example, encode some of the conditions under which a goal succeeds. Sometimes these conditions express situations in which another agent or object helps the goal to succeed. When this is the case, and if it is appropriate for the personality being built to recognize this help, it is a simple matter to have the success test signal this help to the emotion system. Likewise context conditions are expressions by the author of some of the situations when a behavior fails, and some of them naturally have available to them the cause of the failure. These conditions, when appropriate to the personality, can record the cause in memory. If this behavior failure causes an important goal failure, this information can be used by the emotion system to cause anger.

Behaviors themselves also sometimes naturally encode this information. A given goal can succeed by multiple methods depending on which behavior is chosen for it. The difference between these methods in some cases captures the difference between another agent helping and not. For example, if an agent has the goal to move a heavy object, it might have two different behaviors for doing it. The first might be to move it itself, with some risk of hurting itself, and the second might be to get help from another agent to move it. If the second one is chosen and succeeds, then **gratitude toward** the agent that helped might be appropriate. The author writing these two behaviors could encode this by adding a mental act that gives credit to the helping agent as the final step of the second behavior.

6.3.2 Discussion: Inference to Express Personality

It should be remembered that the goal to strive for when building believable agents is to have the agent make inferences that are appropriate to its personality. As is underscored by the simple inference made by Wolf above, where he blithely assigns blame to whoever he is playing with if his fun is aborted, the accuracy of such inferences is only important if accuracy is appropriate to the personality being built. This makes the task of writing such inferences different than it would be in more traditional areas of artificial intelligence. It is not necessarily easier; accuracy is often important, but one should always have the personality in mind when writing these behaviors, and only encode inference or other processing that is appropriate to the behavior and situation. For example, carefree personalities would not perform a large amount of reasoning about whether a goal is likely to succeed. When building a personality that worries, such large reasoning would be exactly what is needed to express the personality. (Such a personality might actually need more complex reasoning than required by accuracy alone, if the personality is an extreme worrier.) Because Hap is a real-time architecture, the time taken to do this reasoning might show up in slower responses to events, etc. And the results of this reasoning might give more objectively accurate fear responses. (Unless, of course, the large reasoning was inaccurate itself, which might be appropriate for particular personalities.)

6.3.3 Passive Goals

Success and failure for many goals are determined by normal Hap processing as described above and in Chapter 4. For some goals, whether they have succeeded or failed must be sensed or inferred. If an agent has the goal that his favorite sports team win this game or that a roulette wheel stop on a particular number, Hap's normal mechanisms will not determine if the goal fails. Sensing must be performed. Neal Reilly calls such goals *passive* goals because they tend to be goals that are not actively pursued by the agent (although there are certainly people who perform actions that they think will affect the outcome of the roulette wheel or their favorite sports team). When such goals are actively pursued by the agent, they can be expressed as normal Hap goals. When they are not pursued they can be expressed as

Hap goals that are not chosen to be executed[2] or expressed in other data structures. In either case, explicit behaviors must be written to determine when they succeed or fail in order for them to generate the four emotions that depend on this information. These behaviors can be sensing behaviors as described in Section 5.2, "thinking" behaviors as described in Section 5.1, or a combination of both.

6.4 Emotional Expression

It is important to express the emotions that are generated. This is the point of generating them. If generated emotions do not ultimately have impact on the activity of the agent, then they are simply data structures that have been written into without ever being used.[3]

Emotions arise by the mechanisms presented in the last section. The architecture supports their expression in all parts of an agent in four ways. First, generated emotions are decayed over time, eventually resulting in their removal. Second, generated emotions are automatically combined into summaries. Next these summaries are mapped (using an author-specified, personality-specific mapping) to styles of action. And finally, the emotional state is available to all of the decision making elements of the agent so that all parts of the agent's behavior can vary based on the current emotional state. Each of these is described in turn below.

6.4.1 Automatic Decay of Emotions

Emotions in traditional characters don't last forever. A character becomes angry; the anger lasts for a while; and it eventually goes away. To address this issue for agents, Em automatically decays the individual emotions periodically. When the intensity of the emotion decays to zero, the emotion is removed.

6.4.2 Summarizing Emotions

As described in Section 6.3, any important goal can potentially give rise to any of the five emotions. This can result in multiple emotions of the same or different types. An agent might be slightly happy because a `greeting` goal succeeded, medium happy because a goal to `play` succeeded, and sad because its `dont_be_teased` passive goal failed. Each of these events would cause a separate **happiness** or **sadness** emotion as appropriate. At any given time, there might be many such emotions present in a given agent. An author might

[2]This can be done by giving them a behavior with a wait goal as its only step. Once expanded, the goal will no longer be pursued by Hap.

[3]This is actually why there are only five emotions in the model presented in this chapter. This model was originally created for the Woggles. At the time we knew there would be limited time to write their behaviors, so we chose a small set that we thought would be the most powerful. Our aim was to use the limited time we had to express this small set as fully as possible, rather than using a larger set of emotions with each emotion given less attention.

want the agent's behavior to vary based on these individual emotions, but it is also useful to be able to write behaviors that vary based on the *overall* level of happiness or sadness the agent is feeling at a given moment.

To facilitate this second form of variation, the emotion system automatically summarizes the emotional state of the agent, by combining emotions of the same type. All of the **happiness** emotions are combined into a single summary **happiness** emotion that expresses the agent's combined level of happiness. Similarly all of the **sadness** emotions are combined. The directed emotions, **anger toward X**, **gratitude toward X** and **fear of X**, are combined when the emotion and direction, **X**, are the same. They are also each combined into an undirected **anger**, **gratitude** and **fear** summary that captures the overall level of anger, gratitude and fear that the agent is experiencing due to all of the emotions of these forms.

Conceptually, each of these summaries is computed continuously so that the information is always current. These summaries provide sets of data that capture the emotional state of the agent at different levels of detail, allowing an author to think about and express variation at any of these levels.

6.4.3 Behavioral Features: Mapping to Styles of Action

Characters don't always show their raw emotions. They may be quite afraid, while showing a calm demeanor, or seething inside while acting very polite and civil. In order to provide architectural support for these types of expression, Em provides a mapping ability from the raw emotions to *behavioral features*. Each behavioral feature represents a style of acting. They are an author-specified set.[4] The initial set is: **act_cheerful**, **act_glum**, **act_alarmed** and **act_aggressive**.

This mapping is specified by the author so it can be tailored to the personality being expressed. This mapping for a candid, expressive or naive personality might be a direct mapping: **happiness** to **act_cheerful**, **sadness** to **act_glum**, **fear** to **act_alarmed** and **anger** to **act_aggressive**. A schoolyard bully, on the other hand, might map **fear** and **happiness** to **aggression** as well as **anger** to **aggression**.

Just as with the summaries in the last section, conceptually the behavioral features are continuously updated, so they are always based on the current emotional state.

6.4.4 Expressing Emotions

Ideally the agent's emotions should be able to influence any aspect of its behavior that the author desires. We allow this in Em and Hap by allowing any of Hap's behaviors to reference any of the parts of the emotional state.

[4]In Neal Reilly's explanation of this work, many of the aspects are ultimately author-specified, as is appropriate in order to be able to flexibly express any believable agent. I have kept to a concrete single emotion system in this description because emotion is not the focus of my work, and this concrete system is sufficient for exploring the issues of emotion in believable agents.

These references can be in any of Hap's decision making expressions: in preconditions when deciding on which behavior to choose for a goal, in success tests, in context conditions, or in mental actions. In this way, any aspect of the agent's behavior can be affected by the current emotional state, from the motivational level to particular choices of actions or subtle timing. Because the other capabilities of the agent are also encoded in Hap, they can also include variation based on emotion. An agent could do path planning for movement differently when angry than when happy. Or it can perform its composite sensing differently in different emotional states. The workings of the emotion system itself can be sensitive to the emotional state. For example, one could build an agent that assigns blame to others much more freely when angry than when happy.

All levels of the emotional state are available for these choices: the behavioral features, emotion summaries, and raw emotions. This allows an author to write behaviors that vary based on the general emotional state or more specific aspects of the state. One interesting use arises naturally in the case of the schoolyard bully (described in Section 6.4.3) that maps **fear** into **act_aggressive**. Most of the bully's behavior might express the state given by the behavioral features including **act_aggressive**, while a few artistically-chosen subtle behaviors, for example a nervous twitch or movement of the eyes, might express his raw **fear** emotion when present.

6.5 Understandability: Importance, Emotion Intensities and Decay

In order for an author to be able to write the emotional variation in an agent's behavior it is important for the author to intuitively understand what all of the elements of the emotional state mean. Some of this understanding is automatic because the author is the one that specifies the importances of the agent's goals, and the initial intensity of an emotion is the same as the importance of the goal that gives rise to that emotion.

The agent will rarely be reacting to freshly generated goals, however. At any given time, the bulk of the raw emotions will have been partially decayed. If an author wants to write behaviors that reference these, then she must intuitively understand how emotions decay over time. Similarly, the summarized emotions are automatic combinations of these raw emotions, so an intuitive understanding of what this combination means is necessary to meaningfully write variations based on these summaries. The behavioral feature mapping is author written, but is based on these summary emotions, so it has the same need.

Neal Reilly gives several choices in his dissertation for how the automatic parts of his emotion system can work. He suggests that different authors might choose different models from his choices. When one considers this need of understandability of the automatic processing, I believe one set is clearly better. This is the model I use in my characters. (I used another model in earlier characters and had difficulty with understandability.) Using this model we can define a set of rules of thumb that make reasoning and intuitions easier about the connection between author-specified importances; the intensities of emotions as they arise, combine and decay; and the match expressions that reference these emotions.

This system of automatic management is defined by two rules:

- For the summaries, emotional intensities are combined logarithmically. If two emotions of intensity i and j are combined they result in a summary with intensity $\log(2^i + 2^j)$.
- When decaying emotions, each raw emotion is decayed such that it is reduced by 1 each second. The summaries and behavioral features are recomputed using the new raw values.

These choices for the automatic management result in good properties from the standpoint of understandability. One of the useful properties is that an emotion summary intensity n will last for n seconds, decaying one unit per second. This is true whether it is the result of a single raw emotion or a group of raw emotions combined. This and other rules of thumb to help authors intuitively understand the connection between importance values and the resulting emotions is given in Figure 6.1.

Rules of Thumb for Meaning of Emotions

- intensity of each emotion is initially equal to the importance of the associated goal.
- emotion summaries are logarithmically combined: (two emotions of intensity 6 combine to a summary of intensity 7).
- one point of decay (in emotion summary or raw emotion) takes approximately one second.

FIGURE 6.1: Rules of Thumb for authors of emotions of causes and expressions of emotions.

This system of automatic management is simple. One could imagine many variations for particular types of expression. Neal Reilly [Neal Reilly 1996] discusses various types of variation one might want, for example one might want emotions to decay at different rates. But, for any chosen system of automatic management, it is important for an author to be able to intuitively understand the flow of emotions over time.

6.6 Implementation in Hap

Section 6.3 describes how part of the processing necessary to generate emotions is implemented in Hap and Section 6.4 describes how Hap is used for the expression of emotions once the emotions are generated and automatically managed.

The rest of the processing of the emotion system is handled by three top-level persistent demons and their behaviors: `em_decay_demon`, `handle_goal_success` and `handle_goal_failure`.

`Em_decay_demon` is a demon that fires once every second. It is typically a low-priority demon (although this can be changed for a particular personality). This causes it to sometimes be delayed in executing. This is why the decay of emotions described in Figure 6.1 is approximate. When it fires, its behavior executes by: reducing every raw emotion by one;[5] removing raw emotions that have intensity zero; updating the emotion summaries; and updating the behavioral features.

`Handle_goal_success` and `handle_goal_failure` are also top level demons. They fire whenever an important goal succeeds or fails. Hap allows this reflection of its processing by recording these events in memory.

The first step of these demons is the goal `infer_credit` or `infer_blame`, respectively. These goals are annotated with `(persistent when_succeeds)` and `ignore_failure` so they cause all of the applicable inference behaviors to execute.

The behaviors for these goals are written by the author to express any methods by which this personality infers credit or blame. (Other methods besides explicit inference as described in Section 6.3.1 could also be used.) These behaviors have access to the goal that succeeded or failed and all of its arguments, as well as all of Hap's normal sources of information: sensing of the external world, reflection of internal state, etc. Such inference behaviors can be written using any inference technology desired by the author. Chapter 5 describes how one can express such computations in Hap.

The next step of both demons is a goal to generate the appropriate raw emotions: **sadness** or **happiness**, respectively, and **anger_toward** or **gratitude_toward** if blame or credit was inferred (or signaled by other aspects of Hap's processing as described above).

The emotion summaries and behavioral features are then updated.

By encoding the emotion system in Hap, we gain the same properties important for believability that are described at the beginning of Chapter 5. One interesting result of this encoding is that the processing of emotions is scheduled in real-time competition with other aspects of the agent's processing. This allows an author, by changing the relative priorities of the emotion goals and other goals of the agent, to create different personality-specific variations. For example, if the emotion processing is high priority, the agent will tend to get less done when being emotional. This effect would be enhanced if the agent had passive goals that caused emotions based on memory or other emotions. One could create an agent that becomes increasingly sad or afraid and unable to function over time. Conversely, by making the emotional processing lower priority than other activities, an author can create a personality that is less emotional when it is busy. For this to be noticeable, the other activities would have to be fairly computationally expensive, because the emotion processing is cheap. One could enhance this by creating specific "busy-work" behaviors that the agent does when it wants to take its mind off of its worries. Neither of these are the

[5]If more than a second has elapsed each emotion is reduced by an amount proportional to the elapsed time.

answer to a complete personality; that still requires many more details, but both could be part of many interesting personality variations.

Some of these effects show up in limited form when these goals are given middle or low priority. These limited effects introduce small timing variation in behaviors, generation of emotions and decay of emotions.

Chapter 7

Constructed Agents

In the previous chapters I have described my architecture for believable agents. In this chapter I want to show how one can use all of the aspects of this architecture to build a complete agent, by describing an implemented agent in detail. The agent I describe is Wolf from the Woggles.

7.1 The Personality to Be Built

The aim in building Wolf was to construct a believable agent that can interact with the other agents and the user, and that exhibits a particular personality. Wolf was designed to be an aggressive personality and a show-off. This personality is described in this original character sketch:[1]

> Wolf - mean guy. leader of wolf/dog pack. He provokes dog to do kind of mean things to shrimp. grey to black. likes to make fun of shrimp or pig, as appropriate. Also has an aesthetic side which makes him like to dance. Can practice dancing with dog. They can do awesome choreographed dances. He also likes to do certain athletics, to show how great he is.

Shrimp is a shy, timid character, and Pig became known as Bear when implemented. He is a big, wise, protector type of character. Dog was never implemented. He was originally a kinder, gentler side-kick to Wolf.

The domain of the Woggles is described in Chapter 3. This domain was constructed to be a simple world, in which most of the complexity is in the behavior of the agents and their interactions with each other and the user.

In the rest of this chapter, I describe in detail the full agent, Wolf. I start by describing its behaviors. Then, I describe how those behaviors are structured and organized by Wolf's top-level. Third, I describe how the emotions arise and are expressed in Wolf. The chapter finishes with a discussion of where Wolf's personality is captured and an extended example

[1] This sketch and an introduction to Wolf were presented in Chapter 4. It is included here for easy reference.

play game (follow-the-leader)	watch people (Woggles)
do the chute	fight
sleep	dance
blink	greet other
run away	cheer-up other
wander/explore	

FIGURE 7.1: Main activities performed by Wolf and other Woggles

that is intended to give some sense of what it is like to interact with or observe Wolf for four minutes.

In the example code fragments, I have stayed as close to Wolf's original, running code as possible, only modifying them as necessary for clarity of presentation.

This chapter is dense with the many necessary details to specify an interactive personality. Those wishing a less detailed introduction to Wolf might want to skim the chapter and read Section 7.8.

7.2 Main Activities and Their Expression as Behaviors

The main activities Wolf (and the other agents, Shrimp and Bear) engage in are listed in Figure 7.1. This choice of activities was motivated by the other designer's and my desire to choose a set that would be artistically powerful, and allow us to build believable agents that could interact with each other and the human user in interesting ways.

As you can see by this list they have normal routine life-activities of sleeping and blinking. They also perform various activities alone that are similar to ones that people might engage in. They wander and explore in their world. They watch other Woggles as a person might watch people in a restaurant or a mall. They dance, a solo activity for them, and they "do the chute", an activity intended to be similar to playing on a slide in a playground or going downhill skiing. As mentioned in Chapter 3, the chute is a tube in the world that goes underground. When the Woggles jump in the end on the right they slide out the other end, and if they have enough speed when they jump in, they are able to slide around twice from one jump because the output end is pointed at the input end.

Wolf and the other Woggles also perform several social activities. They greet each other. They sometimes try to cheer up other sad Woggles. They fight, run away and play games of follow-the-leader together.

Each of these activities is encoded as one or more behaviors. One of the simplest behaviors, blinking, is shown in Figure 7.2. The behavior to blink, `do_blink`, is a sequential behavior that causes a **CloseEyes** action and a **OpenEyes** action in sequence. A delay of 150 milliseconds between the two actions is introduced by the `wait_for` goal in line (7). `Wait_for` is used by many behaviors to introduce time-based delays. The behavior for it

```
(sequential_behavior blink_every___or_so (delay)                          (1)
  (subgoal wait_for "(random()%$$delay + 2)*1000")                        (2)
  (with (success_test eyes already closed (other than by do_blink))       (3)
    (subgoal do_blink)))                                                  (4)

(sequential_behavior do_blink ()                                          (5)
  (act "CloseEyes")                                                       (6)
  (subgoal wait_for 150)                                                  (7)
  (act "OpenEyes"))                                                       (8)

(sequential_behavior wait_for (msec)                                      (9)
  (locals (stop_time current time + $$msec))                             (10)
  (with (success_test time now >= $$stop_time)                           (11)
    (wait)))                                                             (12)
```

FIGURE 7.2: Behaviors that cause Wolf to blink.

is shown in the same figure in lines (9)–(12). It contains a single `wait` step with a success test that causes the step to succeed whenever enough time has passed. Remember that a `wait` step is never pursued and can only be removed by an associated success test or removal of one of the step's ancestors. The current time is sensed initially by the value expression for the local variable `stop_time` in line (10), and then sensed over time by the `success_test` in line (11). While the wait is in effect, this behavior is effectively paused, and other behaviors in Wolf will be executing.

`Do_blink` is called by the behavior `blink_every___or_so`. Wolf has one instance of this goal with a single argument, the number 7. This goal is top-level and persistent, so it is never removed. The behavior waits a random number of seconds and then causes a blink. The behavior then succeeds; the persistent goal is reset, and the behavior runs again. The value 7 in the goal cause the pauses between blinks to be between 2 and 8 seconds. The pause could be longer depending on available processing time for the agent.

Playing the game of follow-the-leader is one of Wolf's more complicated activities. There are two sets of behaviors for this activity: one to lead in a game of follow-the-leader and the other one to follow another in such a game. The basic structure for the behavior to lead, `lead_the_follower`, is shown in Figure 7.3. This behavior is called with the other Woggle that Wolf wants to play with as an argument of the goal. The behavior is then three steps: get that Woggle to play, lead for a time, and then stop playing. These steps are shown in lines (1)–(4).

Getting the Woggle to play is accomplished by asking it to follow using body language: going up to the Woggle, greeting it, and then jumping away. (This body language was motivated by observations of how non-linguistic creatures, such as cats, get people to

```
(sequential_behavior lead_the_follower (who)
  (subgoal get_follower $$who)
  (subgoal lead_for_time $$who)
  (subgoal stop_playing $$who))

(sequential_behavior get_follower (who)
       (context_condition less than 15 seconds elapsed)
  (subgoal hey $$who)
  (subgoal jump_away_from $$who))

(sequential_behavior lead_for_time (who)
  (with (success_test more than 45 seconds elapsed)
    (subgoal ltf_do_fun_stuff $$who)))

(concurrent_behavior ltf_do_fun_stuff (who)
  (with persistent
    (subgoal ltf_go_to_interesting_place))
  (with (priority_modifier 1) persistent
    (subgoal ltf_play_in_interesting_places))
  (with (persistent when_succeeds)
    (subgoal ltf_do_fun_acrobatics))
  (with persistent
    (subgoal change_speeds $$who))
  (with (priority_modifier 5)
        (persistent when_succeeds)
    (subgoal monitor_follower $$who)))

(sequential_behavior stop_playing (who)
  (with (success_test attempted goal for more than 20 seconds)
    (subgoal hey $$who)))
```

FIGURE 7.3: Basic structure of lead behaviors for follow-the-leader.

follow them.) The first two of these steps — going up to the other Woggle and greeting it — are accomplished by the `hey` goal, in line (7), and behavior, which is not shown. The behavior for `hey` is a robust behavior that approaches the target, gets in its field of view and performs the greeting gesture: an appropriately-timed and sized squash up followed by a squash down. The `hey` behavior continuously attempts this sequence until it is successful. It will chase a moving Woggle until it has successfully gotten close enough, moved into its field of view and performed the body motion. During this behavior it is always looking at the Woggle it is trying to greet, as well as often moving toward it, giving some indication of its intention. Nevertheless, for an uncooperative or oblivious Woggle (either human or Hap controlled), this behavior might never terminate. This is why the context condition is included in the `get_follower` behavior in line (6). This context condition causes `get_follower` to fail if Wolf has spent too much time (15 seconds) trying to invite the other Woggle.

In the social conventions we created for this world there is no way for a Woggle to respond in body language to an invitation to play follow-the-leader; either they play or they don't. Thus, this is the end of the `get_follower` behavior. Determining whether the other is playing or not is accomplished by the `lead_for_time` behavior.

The behavior for `lead_for_time` has a single step that is the goal `ltf_do_fun_stuff`. As you will see in the description of the behavior for this goal, it continues indefinitely or until it fails. The success test added to the goal causes it to succeed when the leading has been pursued for 45 seconds.

The behavior for `ltf_do_fun_stuff` has five steps that are pursued concurrently as shown in lines (12)–(23). The first three cause the activity of leading. The first step is the goal `ltf_go_to_interesting_place`. The behavior for this goal picks a place randomly from the places Wolf considers interesting, and then moves toward it. Because this goal is marked `persistent`, when it finishes, it is available to run again with Wolf picking a new interesting place to lead toward.

The next goal, `ltf_play_in_interesting_places`, is a demon[2] that recognizes when Wolf is in one of two places that he considers particularly interesting. If Wolf is in one of these places, this demon fires and Wolf leads around in the interesting location for a while. This goal is higher priority than the goal `ltf_go_to_interesting_place` because of the `priority_modifier` of 1. Because it is higher priority, it interrupts that behavior while it runs. Both behaviors primarily issue movement actions, so even if there is sufficient brain time to mix in the lower priority `ltf_go_to_interesting_place` behavior, it would only advance to attempting to issue its next movement action before being suspended because of a resource conflict with the movement actions of `ltf_play_in_interesting_places`. The `ltf_play_in_interesting_places` behavior has a `wait_for` goal as its last step.

[2]More precisely, it is its behavior that encodes the demon. `Ltf_play_in_interesting_places` is a normal goal, with a single sequential behavior. The first step of the behavior is a wait step with a success test that encodes the firing condition of the demon. The later steps of the behavior express the body of the demon. This structure is described in Sections 4.8.1 and 4.14.2. Because explicitly describing this structure for every demon goal would quickly become tedious, when a goal's only behavior has this structure, I will describe it as a demon by way of shorthand.

This introduces a delay in the stream of movement actions it issues to allow the `ltf_go_to_interesting_place` behavior to move away from the current location. This pause is important because `ltf_play_in_interesting_places` is marked `persistent`, and so it resets when it is done, ready to issue new movement actions if Wolf is in an "interesting" place. If there were no pause it would play in the same "interesting" place for the duration of the game.

While leading, the last goal of the behavior, `monitor_follower`, periodically monitors to make sure that the follower is keeping up. The information it provides about how well the follower is keeping up is used by the fourth goal, `change_speeds` to adjust the speed with which Wolf leads. In addition, when the `monitor_follower` behavior notices that the follower is not keeping up, it pauses Wolf's leading movements and issues actions to show Wolf's impatience.

The basic structure of the `monitor_follower` behavior is given in Figure 7.4. Every time the behavior runs, it starts by glancing toward the follower, line (5). The second step of the behavior, in lines (6)–(7), is a wait step. If the follower keeps up, the behavior will do nothing more. If it falls behind, the success test associated with the wait step will fire when the follower gets a certain distance away from Wolf, and the rest of the behavior will execute. The next step, in line (8), then records when the follower fell behind using a mental action to record the current time in a local variable.

The next step is the goal `ltf_wait_to_catch_up` in line (9). In addition to executing, this goal conflicts with all of the actions that cause physical action. As shown in Figure 7.3, line (21), the `monitor_follower` goal is annotated with a `priority_modifier` of 5, causing it to be higher priority than any of the action-producing goals of the lead behavior. This higher priority is inherited by `ltf_wait_to_catch_up` and combines with the conflicts to cause Wolf to stop its leading movements if it ever gets to this point in the behavior. (If a higher priority goal, such as one to run away, causes movement, it would take precedence over the conflicting `ltf_wait_to_catch_up` goal and the movement would happen.)

In addition to stopping Wolf's leading and other movement that is not high priority, Wolf continues to execute the `ltf_wait_to_catch_up` behavior, given in lines (15)–(21). The behavior first causes Wolf to look at and track the follower (using **StartTrackWoggleEyes**), and then makes Wolf's body its normal shape (using **Squash** with a squash value of 0). He then executes the goal `ltf_continually_monitor_progress` of the follower. This goal has a success test, line (20), that causes it to succeed if the follower catches up by coming near to Wolf. Until that happens it performs an infinite loop (caused by the persistent annotation on `ltf_monitor_progress` in line (23)) of the goals `ltf_wait_for_guy` and `ltf_look_around` in sequence, lines (24)–(26). `Ltf_wait_for_guy` causes Wolf to look at the follower while performing movements that show his impatience. `Ltf_look_around` randomly looks around the world. While this behavior is going on, the context condition that applies to the `ltf_wait_to_catch_up` behavior, line (16), is being continuously evaluated. If too much time elapses, it fires causing the behavior to fail. This failure causes cascading effects, resulting in the entire `lead_the_follower` behavior and goal failing. This context condition and enclosing behavior is what recognizes a follower quitting prematurely, playing badly (in Wolf's opinion), or never starting to play in the first place.

```
(sequential_production monitor_follower (guy)
       (locals (time_fell_behind 0)
               (i 0)
               (delay_to_next_monitor 0))
  (subgoal glance_at $$guy)
  ;; wait for him to fall behind
  (with (success_test $$guy is too far away from me)
    (wait))
  ;; wait for him to catch up
  (mental_act "$$time_fell_behind = simtime();"))
  (subgoal ltf_wait_to_catch_up guy $$time_fell_behind)
  ;; if he catches up, ...
     ;; remember how long it took him in case you want to change your speed
  (mental_act $$ltf_lag_time = now - $$time_fell_behind)
  (subgoal glance_at $$guy)
     ;; delay until next monitor – amount of delay based on emotion
  (mental_act "$$i = intensity of aggression toward follower")
  (mental_act "$$delay_to_next_monitor = 1000 + 9000 * $$i/10")
  (subgoal wait_for $$delay_to_next_monitor))

(sequential_production ltf_wait_to_catch_up (guy start_wait_time)
       (context_condition time is before start_wait_time + FTL_CATCHUP_TIME)
  (act "StartTrackWoggleEyes" guy)
  (act "Squash" 0 100)
  (with (priority_modifier 40)
        (success_test $$guy is near me again)
    (subgoal ltf_continually_monitor_progress guy)))

(sequential_production ltf_continually_monitor_progress (guy)
  (with persistent (subgoal ltf_monitor_progress guy)))

(sequential_production ltf_monitor_progress (guy)
  (subgoal ltf_wait_for_guy guy)
  (subgoal ltf_look_around))
```

FIGURE 7.4: Basic structure of ltf_monitor_follower for lead_the_follower.

`do_blink`	`gang_up_on_fight`	`sleep`
`lead_for_time`	`threaten_other`	`sigh`
`get_follower`	`stay_in_view`	`mope`
`ltf_wait_to_catch_up`	`puff`	`run_away`
`goto_screen`	`bash`	`look_around_nervously`
`watch_person_behind_screen`	`dance_on_pedestals`	`raw_hey`
`goto_perch`	`jump_through_chute`	`hey`
`watch_a_woggle_until_its_boring`	`copy_squashes`	`go_to_woggle`
`wander_explore`	`copy_jumps`	`goto_region`
`release_your_aggression`	`catch_up`	`sad_goto`
`threaten_for_fun`	`follow_the_leader`	`scared_goto_region`
`threaten_when_threatened`	`goto_bed`	

FIGURE 7.5: Main Action-producing Behaviors in Wolf

At this point in the behavior there is a race between the follower catching up and triggering the success test associated with `ltf_continually_monitor_progress`, line (20), causing the behavior to succeed, and it taking too long for the follower to catch up and the behavior failing, causing the `lead_the_follower` behavior itself to fail.

If the follower catches up, this is the last step in the `ltf_wait_to_catch_up`, so the behavior and goal succeed, and the conflict with movement goals is automatically removed by Hap. This allows the movement behaviors of leading to continue when they are next processed. First however, the monitor behavior continues executing because it is higher priority. The next step of `monitor_follower`, in line (11), causes Wolf to glance toward the follower. The behavior then computes in lines (12)–(13) how long it will wait until checking again to see if the follower has lagged behind. This time is computed as a linear interpolation between 1 and 10 seconds based on how aggressive Wolf is feeling toward the follower (by referencing his **aggressive toward** behavioral features). If he is not feeling aggressive, he will wait only a second between monitoring, paying more attention and perhaps waiting for the follower more. If he is feeling maximally aggressive he will wait 10 seconds between monitoring, perhaps getting far ahead of the follower before noticing. The computed delay is caused by the last `wait_for` goal in line (14) of the behavior. The behavior succeeds and restarts from the beginning after this step because the invoking goal is marked `(persistent when_succeeds)`.

In addition to `blink_every___or_so` and `lead_the_follower`, Wolf has several other recognizable action-producing behaviors. The full list of Wolf's main action-producing behaviors is given in Figure 7.5. Wolf has a great many more behaviors than are listed here, but other behaviors are either not action-producing (these are described below), or they are behaviors subsidiary to the ones listed in Figure 7.5 that do not seem distinct enough to warrant independent mention. Taken together, all of these behaviors provide much of the knowledge for Wolf to engage in the activities listed in Figure 7.1. The listed behaviors are not presented in any particular order. Some of them have been described in detail previously, but I describe each briefly here to give an understanding of the range of behaviors Wolf engages in.

- `do_blink`, `lead_for_time`, `get_follower` and `ltf_wait_to_catch_up` are all described in detail previously in this section. `Do_blink` causes Wolf to blink. `Lead_for_time` is the behavior that leads in a game of follow-the-leader. `Get_follower` invites another Woggle to play, and `ltf_wait_to_catch_up` causes Wolf to wait, and express this waiting through impatient movements, when he notices that his follower has fallen behind.

- `goto_screen` and `watch_person_behind_screen` both treat the screen as if it is a glass wall between the Woggles' world and the real world. `Goto_screen` causes Wolf to move to spots in the world that look like he is near the glass. `Watch_person_behind_screen` uses information from attached sonar sensors to allow Wolf to look at the person in front of the computer. Because of the implementation of the sonar sensors, this behavior only works when there is a single person in front of the screen.

- `goto_perch` causes Wolf to go to one of the places from which he likes to watch things.

- `watch_a_woggle_until_its_boring` causes Wolf to watch a chosen Woggle for a while.

- `wander_explore` causes Wolf to explore the Woggle world, paying particular attention to edges and corners in the world.

- `release_your_aggression` is a behavior that is only triggered when Wolf is feeling sufficiently aggressive. It then does physical behaviors to release the aggression, by either picking a fight or dancing.

- `threaten_for_fun`, `threaten_when_threatened` and `gang_up_on_fight` are the ways in which Wolf gets into a fight. He initiates them for fun; if threatened, he will threaten back; and when he sees a good fight in the world he may choose to join in.

- `threaten_other` is the behavior that actually does the fighting. It coordinates the next three behaviors to try to be intimidating to the one he is threatening.

- `stay_in_view`, `puff` and `bash` are behaviors that are used by Wolf to perform his threaten behavior. `stay_in_view` uses aggressive movements, e.g. **put** actions with short duration, to stay close to the chosen Woggle and in its face. `puff` was described in Section 5.2 and pictured in Figure 5.3. It is a movement in which Wolf changes his body radii to appear bigger from the front (somewhat like a cobra spreading its hood). `bash` is the same movement as a puff, while also leaning in toward the Woggle to appear more intimidating.

- dance_on_pedestals performs a somewhat stylized dance on the five pedestals in the lower right of the world (Figure 3.1) by jumping from pedestal to pedestal and spinning.
- jump_through_chute causes Wolf to jump in the chute, much as a child might slide on a playground slide.
- copy_squashes, copy_jumps and catch_up are all behaviors of follow_the_leader used when Wolf is following the User or other Woggle in a game of follow-the-leader. copy_squashes and copy_jumps both try to duplicate the movements of the leader Woggle, and catch_up causes Wolf to take short-cuts to catch up when he has fallen behind the leader.
- goto_bed picks a bed and goes to it to sleep. Wolf picks one of the two bed rocks (in the lower left corner of the world Figure 3.1) arbitrarily. Other agents pick differently, for example Shrimp tries to pick one that his friend is sleeping on and avoids ones that Woggles he is afraid of are on.
- sleep is the behavior that causes Wolf to sleep. He closes his eyes, doesn't move and "breathes" (by squashing slowly up and down).
- sigh performs two slow squashes in sequence to look like Wolf is sadly breathing.
- mope causes Wolf to move to a random place in the world in a sad manner (see sad_goto below) and then sit there and sigh until he isn't sad anymore.
- run_away is only performed when Wolf is really scared of some other Woggle. He then moves away from them quickly while looking back at them repeatedly in a nervous way.
- look_around_nervously cause Wolf to look around in a nervous way. He does this by darting his eyes quickly around at a number of spots.
- raw_hey is a stylized body motion that is taken in the Woggle culture as a greeting, part of an invitation, or to mean "let's stop playing" depending on the context. These are each described in the description of recognize_and_interpret_hey below. It and hey below were mentioned previously in this section as part of Wolf's get_follower behavior. The movement itself is a squash up of around 1.5 times the Woggle's original height with duration of around a third of a second, followed by a squash back to normal with the same third of a second duration.
- hey is the behavior that is most often used to perform a hey. It moves to the target of the hey, moves in front of it at a comfortable social distance and does a raw_hey.
- go_to_woggle, goto_region, sad_goto and scared_goto_region are all behaviors that enable Wolf to move around in the World. go_to_woggle causes Wolf to move toward a target Woggle when it stays still or moves independently. The other

```
recognize_threaten
monitor_follower
recognize_and_interpret_hey
fear_being_hurt_distance_demon
recognize_falling_behind
recognize_and_remember_jumps
recognize_and_remember_puts
recognize_and_remember_squashes
```

FIGURE 7.6: Composite Sensing Behaviors in Wolf

three behaviors all cause the movement to a non-moving region. The sad and scared variations cause the movement to be colored by those emotions. `Sad_goto` is slower, uses smaller jumps, and has `sigh` goals intermixed. `Scared_goto_region` causes Wolf to move quickly, and inserts `look_around_nervously` actions in between movements. All of these are varied in how fast they move (subject to their individual biases) by how energetic Wolf is feeling at the moment (see Section 7.6.2).

7.3 Composite Sensing Behaviors

In addition to the basic sensors provided by the domain (Chapter 3), Wolf has a number of composite sensing behaviors that work with and enable the action-producing behaviors described in the previous section.

A complete, detailed description of one of the more complex of these composite sensing behaviors, `recognize_threaten`, is given in Section 5.2, with the code for the behavior given in Figure 5.4. The version used in Wolf is an earlier version of this sensor that differs from this detailed description in that it takes two arguments, the potential victim and potential aggressor and only recognizes threats that by the aggressor to the victim. Thus Wolf must have a version of this recognizer for every pair of aggressor and victim that he wants to monitor.

The full list of Wolf's composite sensing behaviors is given in Figure 7.6. The function of each of these composite sensing behaviors is described here.

- <u>`recognize_threaten`</u> recognizes when a Woggle is threatened by another. Threatening is done by moving within the Woggle's physical space and performing multiple (3) fast movements or puffing at the Woggle.

  This sensor is used by Wolf to recognize threats to himself for the above `threaten_when_threatened` behavior and to recognize threats to others for the `gang_up_on_fight` behavior. As described below, it is also used to generate emotions in Wolf.

- <u>`monitor_follower`</u> is a composite sensor that is used by the `ltf_do_fun_stuff` of `lead_the_follower`. It is described in detail in the previous section.

- recognize_and_interpret_hey is the behavior that recognizes the stylized hey body motion (as performed by raw_hey). If this gesture is followed within two seconds by the Woggle jumping directly away (plus or minus .1 radians), the behavior categorizes the gesture as an invitation to play, and otherwise categorizes it as a greeting.

  In addition to recognizing the gesture, and as a product of the recognition process, recognize_and_interpret_hey causes certain actions to show Wolf's thinking as he is recognizing or partially recognizing these events. When the behavior recognizes a motion that looks like the first squash of a hey gesture, it issues a **StartTrackWoggleEyes** action causing Wolf to look at the potentially heying Woggle. When a hey is recognized, the behavior issues a raw_hey subgoal causing Wolf to acknowledge the hey by heying back.

  In addition to the social interaction of greeting each other, a recognized invitation enables the follow_the_leader behavior. Also, a greeting by the leader in a game of follow-the-leader to the follower is interpreted as the leader wanting to end the game.

- fear_being_hurt_distance_demon recognizes when an agent moves nearby that Wolf does not like (that Wolf has a negative valued **like** relationship with). It then causes raw **fear of** that agent because of the dont_be_hurt passive goal. (This fear is normally suppressed in Wolf, as described in Section 7.6.2, unless it becomes very intense.)

- recognize_falling_behind, recognize_and_remember_jumps, recognize_and_remember_puts and recognize_and_remember_squashes are all composite sensors used by the follow_the_leader behavior. Recognize_falling_behind is active whenever Wolf is following in a game, and it recognizes when he gets behind. This triggers the catch_up behavior and causes emotional reactions (see Section 7.6.1). Recognize_and_remember_jumps, recognize_and_remember_puts and recognize_and_remember_squashes are all used to remember the actions that the leader performs. The remembered **Jump**, **Put** and **Squash** actions are used by the copy_jumps and copy_squashes behaviors. Both **Put** and **Jump** actions are copied as **Jump** actions in follow_the_leader.

```
handle_goal_success
handle_goal_failure
em_decay
```

FIGURE 7.7: Emotion processing behaviors

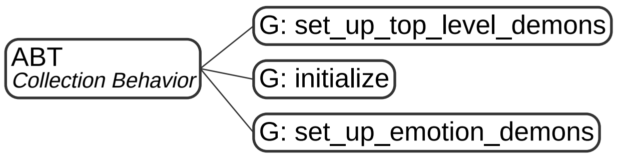


FIGURE 7.8: Wolf's Initial Active Behavior Tree

7.4 Emotion Support Behaviors

The last category of behaviors in Wolf are emotional processing behaviors. These behaviors perform much of the automatic processing of emotions in Wolf. They are listed in Figure 7.7, and are described in more detail in Section 6.6.

- `Handle_goal_success` and `handle_goal_failure` are behaviors that process goal successes and failures of Hap goals that have been annotated as *important*. `Handle_goal_success` creates **happiness** and **gratitude** emotional data structures, and `handle_goal_failure` creates structures for **sadness** and **anger**.
- `Em_decay` causes Wolf's emotional data structures to decay (back to neutral values) over time. Emotions that have fully decayed (to zero) are removed.

The automatic combination of emotions and mapping to behavior features, as described in Chapter 6, are performed as part of these three computations.

7.5 Organization of Behaviors

The behaviors described above are organized by the structure of Wolf's active behavior tree. Initially Wolf's ABT has three top level goals as shown in Figure 7.8. The `initialize` goal is highest priority of the three and is marked as conflicting with the other two, so it executes alone to completion before any other goals execute. It sets the initial physical parameters of the Woggle as well as initializing some needed internal state.

After `initialize` succeeds and the other two top-level goals are expanded, the structure of the ABT is as shown in Figure 7.9. The two behaviors are concurrent behaviors and all of their steps are marked persistent. Also, neither of the two root goals or their behaviors have reactive annotations. Thus, after the ABT is expanded to this point none of these nodes will be removed. The ABT will expand as Wolf pursues these goals, and contract back to this point as those goals succeed or fail. By grouping these goals under the otherwise

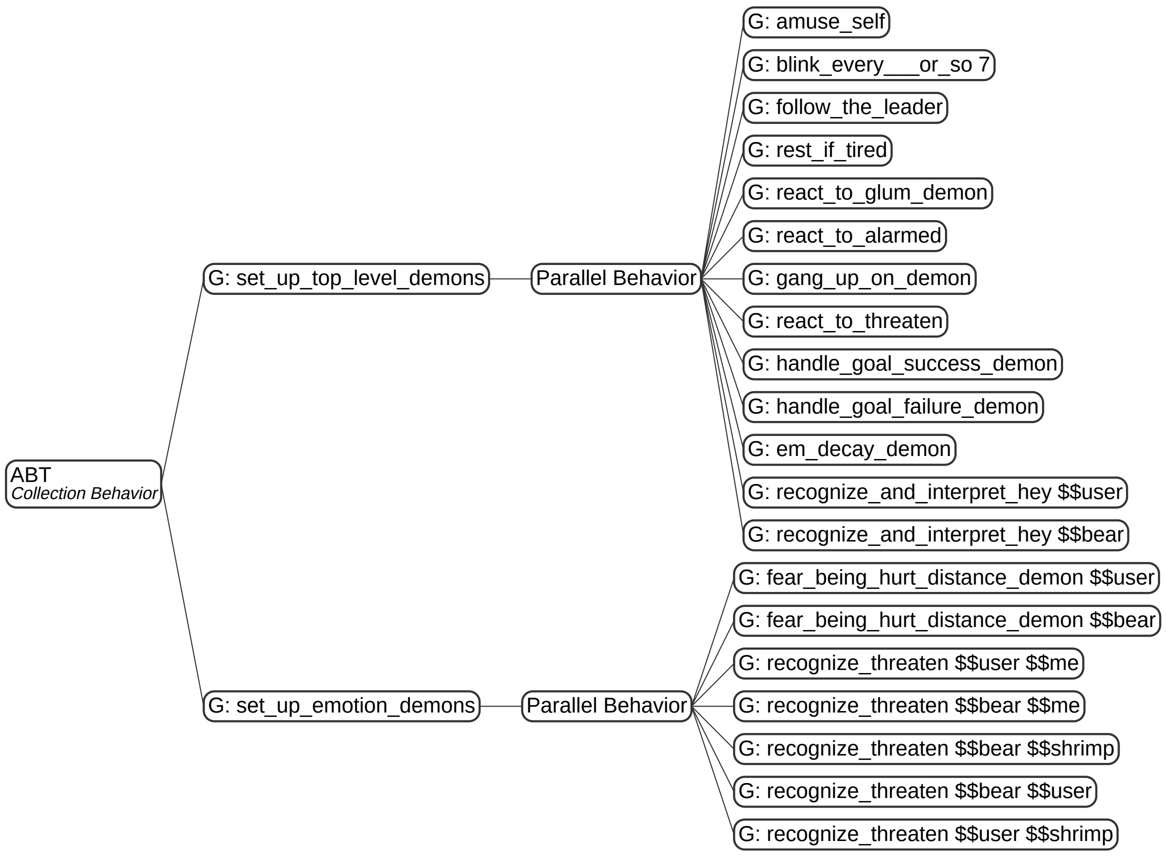

ABT
Collection Behavior
G: set_up_top_level_demons
Parallel Behavior
G: amuse_self
G: blink_every___or_so 7
G: follow_the_leader
G: rest_if_tired
G: react_to_glum_demon
G: react_to_alarmed
G: gang_up_on_demon
G: react_to_threaten
G: handle_goal_success_demon
G: handle_goal_failure_demon
G: em_decay_demon
G: recognize_and_interpret_hey $$user
G: recognize_and_interpret_hey $$bear
G: set_up_emotion_demons
Parallel Behavior
G: fear_being_hurt_distance_demon $$user
G: fear_being_hurt_distance_demon $$bear
G: recognize_threaten $$user $$me
G: recognize_threaten $$bear $$me
G: recognize_threaten $$bear $$shrimp
G: recognize_threaten $$bear $$user
G: recognize_threaten $$user $$shrimp

FIGURE 7.9: Example Active Behavior Tree

unnecessary root goals, rather than placing each of these persistent steps at the root, the conflicts between and relative priority orderings of the groups can be more easily specified, as is the case with these groups and the `initialize` goal described above.

As shown in Figure 7.9, Wolf's stable portion of the ABT has 20 constant leaf goals. These goals are the goals from which all of Wolf's activity arises. For the remainder of this chapter, I will refer to these goals as Wolf's *effective top-level goals*.

The bottom nine of these goals are all instances of three composite sensors described previously, `recognize_and_interpret_hey`, `fear_being_hurt_distance_demon` and `recognize_threaten`. Each of these composite sensing behaviors is instantiated with specific arguments that define what they sense. Wolf has a `recognize_and_interpret_hey` goal to recognize `hey` gestures from both the User and Bear. Similarly, he has a `fear_being_hurt_distance_demon` for both the User and Bear. He doesn't have one of either of these sensors for Shrimp, because the personality envisioned for Wolf doesn't think much of Shrimp; he simply is never afraid of Shrimp and never acknowledges any interaction that Shrimp initiates. He does, however, sometimes initiate interaction with Shrimp both by starting a game with Shrimp or by threatening him. Similarly he has two `recognize_threaten` goals to recognize threats to himself from the User or Bear. He doesn't recognize any threats to himself by Shrimp. He also has three goals to recognize

blink_every__or_so:	do_blink
follow_the_leader:	copy_squashes
	copy_jumps
	catch_up
	recognize_falling_behind
	recognize_and_remember_jumps
	recognize_and_remember_puts
	recognize_and_remember_squashes
rest_if_tired:	goto_bed
	sleep
react_to_glum_demon:	sigh
	mope
	sad_goto
react_to_alarmed:	run_away
	look_around_nervously
	scared_goto_region
gang_up_on_demon:	gang_up_on_fight
	threaten_other
	stay_in_view
	puff
	bash
react_to_threaten:	threaten_when_threatened
	threaten_other
	go_to_woggle
	stay_in_view
	puff
	bash
handle_goal_success_demon:	handle_goal_success
handle_goal_failure_demon:	handle_goal_failure
em_decay_demon:	em_decay

FIGURE 7.10: Behaviors organized by Wolf's effective top-level goals

threats to others (for potential ganging up on): threats to Shrimp by Bear or the User, and threats to the User by Bear. Again, his personality's image of Shrimp is that either Shrimp would never threaten anyone, or that such a fight isn't worthy of his attention, so he has no sensors to recognize threats by Shrimp to either the User or Bear.

While the bottom nine goals are directly instances of three of the behaviors described in the previous section, the other goals are all present in Wolf's ABT to organize the behaviors previously described. The behaviors organized by each goal are presented in Figure 7.10. The only main behaviors left out of this list are the general-purpose behaviors, `goto_region`, `go_to_woggle` and `hey`. These behaviors are used in multiple places when the Woggle needs to go somewhere or greet another Woggle.

One of these is chosen at a time for amuse_self

track_user_for_time:	goto_screen
	watch_person_behind_screen
watch_woggles_for_time:	goto_perch
	watch_a_woggle_until_its_boring
explore_wander_for_time:	wander_explore
lead_someone:	lead_for_time
	get_follower
	ltf_wait_to_catch_up
	monitor_follower
release_your_aggression:	threaten_for_fun
	threaten_other
	stay_in_view
	puff
	bash
	dance_on_pedestals
do_the_chute:	jump_through_chute
dance_on_pedestals_for_time:	dance_on_pedestals

FIGURE 7.11: Seven goals chosen among as Wolf's way of amusing himself and the behaviors that they each organize.

`Amuse_self` is the only goal in Wolf's effective top-level that has yet to be described. When `amuse_self` is chosen to be executed in Wolf, one of seven distinct behaviors is chosen as the method for Wolf to amuse himself now. Each of these seven behaviors gives rise to a single goal that then is similar to the effective top-level goals in that its purpose is to structure and organize the main behaviors described in the earlier sections. The set of seven goals that arise for `amuse_self` and the behaviors that they organize are listed in Figure 7.11.

7.5.1 Demons and Enabling Conditions

With the exception of `amuse_self`, the rest of the goals in Wolf's effective top-level are demons with various enabling conditions. (Composite sensors can be thought of as demons that when triggered give rise to some set of other demons to recognize/verify further, together with mental and physical actions that respond to the recognition or partial recognition thus far.) These demons remain dormant until their enabling condition becomes true, and then the rest of the behavior is available to be executed. The enabling conditions of each of the demons is given in Figure 7.12. This figure also gives the priority for each of Wolf's effective top-level goals. Remember that these goals are pursued concurrently because of the structure of the ABT.

Each enabling condition for the demons is encoded by building a sequential behavior

Effective Top-level Goal	Priority	Enabling condition (if demon)
amuse_self	1000	
blink_every___or_so	10000	2-8 seconds (chosen randomly) elapsed since last blink
follow_the_leader	2000	invited within last 2 seconds and $$if_invited_then_play is true
rest_if_tired	7000	**energy** <= 1 and $$if_tired_then_rest is true
react_to_glum_demon	3000	**act_glum** >= 3 and $$if_glum_then_mope is true
react_to_alarmed	9500	**act_alarmed_by X** > 3 and $$if_alarmed_then_run is true
gang_up_on_demon	6000	recognized a threat in the last 1.5 seconds that was not by me or toward me or toward bear and $$if_other_threat_then_gang_up is true and **act_aggressive** > 2
react_to_threaten	9000	recognized a threat in the last 1.5 seconds toward me, that was not by bear and if_threat_then_react is true
handle_goal_success_demon	11000	important Hap goal succeeded and has not been processed
handle_goal_failure_demon	11000	important Hap goal failed and has not been processed
em_decay_demon	1	one second elapsed since last decay
recognize_and_interpret_hey $$user	2000	*see Section 7.3*
recognize_and_interpret_hey $$bear	2000	*see Section 7.3*
fear_being_hurt_distance_demon $$user	4000	*see Section 7.3*
fear_being_hurt_distance_demon $$bear	4000	*see Section 7.3*
recognize_threaten $$user $$me	8000	*see Section 7.3*
recognize_threaten $$bear $$me	8000	*see Section 7.3*
recognize_threaten $$bear $$shrimp	5000	*see Section 7.3*
recognize_threaten $$bear $$user	5000	*see Section 7.3*
recognize_threaten $$user $$shrimp	5000	*see Section 7.3*

FIGURE 7.12: Priorities for all of Wolf's effective top-level goals, and enabling conditions for the demons.

for the goal with two or more steps. The first step is a `wait` step with an associated `success_test` that encodes the appropriate condition given in Figure 7.12. The second (and later) steps of the behavior encode the action(s) or behavior that Wolf engages in when the enabling condition is true. These steps and their subsidiary behaviors, goals and actions organize the parts for the goal that are listed in Figure 7.10. Each of the enabling conditions is described in turn here.

- `blink_every___or_so` The behavior for this goal randomly chooses a fixed number between 2 and 8 and records the current time. When the chosen number of seconds have elapsed from the recorded time, the condition becomes true. When the behavior finishes, the goal is reset (because it is marked `persistent`) and the same behavior executes resulting in a periodic blinking with between 2 and 8 seconds between blinks.

- `follow_the_leader` The behavior for this goal proceeds when Wolf has recognized an invitation to him within the last two seconds. Such an invitation is recognized and recorded by the `recognize_and_interpret_hey_sensor`. The global variable `$$if_invited_then_play` must also be true. This variable is true by default, and is temporarily set to false by `lead_the_follower` when it is executing. This prevents Wolf from accepting an invitation to play, when it would interrupt a perfectly good game he was already leading. It is also set to false for the duration of a `dance_on_pedestals` behavior.

- `rest_if_tired` is enabled when Wolf's **energy**[3] is 1 or lower.

- `react_to_glum_demon` is enabled when Wolf's **act_glum** behavioral feature has intensity three or greater. The variable `$$if_glum_then_mope` must also be true. This variable is temporarily set to false during `lead_the_follower` to prevent Wolf from moping during a game.

- `react_to_alarmed` is enabled when Wolf has an **act_alarmed toward** behavioral feature with a high intensity (intensity 4 or greater), and the variable `$$if_alarmed_then_run` is true. This variable is temporarily set to false by the `rest` behavior. The direction of the **act_alarmed toward** feature is used by the resulting behavior to know what to act afraid of.

- `gang_up_on_demon` is enabled when Wolf has an **act_aggressive** behavioral feature with (intensity 3 or more), and has recognized a threat within the last 1.5 seconds that was performed by another Woggle and was directed at someone other than Wolf or Bear. (The threat recognition is done concurrently by one of his `recognize_threaten` goals and behaviors.) The variable `$$if_other_threat_then_gang_up` must also be true. It is temporarily set to false during the `lead_the_follower` and `dance_on_pedestals` behaviors.

[3]**Energy** is described in Section 7.6.2.

- react_to_threaten is enabled whenever Wolf has recognized a threat within the last 1.5 seconds, directed at him, that was not performed by Bear. (The threat recognition is done by one of his recognize_threaten goals and behaviors.) The variable $$if_threat_then_react must also be true. It is temporarily set to false during Wolf's gang_up_on behavior so that he doesn't interrupt one good fight for another.

- handle_goal_success_demon is triggered whenever any Hap goal succeeds that has been annotated as important (as described in the emotion chapter). The goals in Wolf that are annotated as important are described in Section 7.6.1.

- handle_goal_failure_demon is triggered whenever any Hap goal fails that has been annotated as important.

- em_decay_demon is triggered when one second has elapsed since the last decay.

Global variables are consulted in a number of the enabling conditions to allow lower-priority behaviors to suspend higher-priority behaviors. This is complementary to Hap's conflict mechanism in which higher priority goals suspend conflicting lower-priority goals and actions. This use of global variables allows an artist to encode shades of behavior in Wolf. For example, normally when Wolf becomes really scared, running away seems like the only thing on his mind, but when he has already decided to go to sleep, and is perhaps trudging toward the bed, he doesn't run away no matter what agents do to try to make him run away; he is too groggy to pay attention to such things.

The amuse_self goal is the only one without an enabling condition. Thus, if none of the enabling conditions becomes true, this goal is the only one that would be pursued. When one of the enabling conditions becomes true, which is pursued first is dependent on the relative priorities of the amuse_self goal and the goals arising from the firing of the demon. (Remember that priorities are inherited, so the steps of the behavior will have the same priority as the effective top-level goal that gives rise to it unless it is annotated by a priority modifier. The priority modifiers in Wolf are described below.)

If the priority of the goals arising from the firing of the demon are lower than those arising from the amuse_self goal, then they will only be pursued when there is computation time left over in Hap, for example when the chosen amuse_self behavior has just issued an action or encountered a wait step, and doesn't have other concurrent activities to pursue. During these pauses, the enabled behavior will be executed. An exception to this is if the enabled goals *conflict* with the goals executing in pursuit of amuse_self. In that case the enabled behavior and goals will not execute until the conflicting goal finishes. In Wolf, only em_decay_demon is lower priority than amuse_self. (I discuss goal conflicts below.)

If the priority of the firing demon is higher than amuse_self and all of its expanded subgoals, then it is immediately pursued. If there is a conflict with an executing action or goal in the already executing, but lower priority, amuse_self behavior, then that action or goal is suspended until the conflicting goal in the demon finishes. In Wolf, all of the demons are higher priority than amuse_self except for em_decay_demon.

When one or more of the enabling conditions have fired, and its behavior is in the process of being executed, it is in the same position as `amuse_self` in deciding which is the main line of action and which is mixed in as time allows. If any of the executing behaviors or the actions they give rise to conflict, then the higher priority one has precedence.

7.5.2 Priorities

Wolf's top-level priority structure is given in Figure 7.12. This structure divides Wolf's activities into distinct levels with different priorities. The support behaviors that generate emotion from goal success and failure are at the highest level, followed by blinking and reacting to alarm, and so on. Thus when external or internal conditions cause one of these demons to fire, whether it has precedence depends on where it falls in this ordering relative to the other active behaviors.

Priority modifiers can change this ordering. This occurs in Wolf whenever he executes a `do_the_chute` behavior. This behavior adds a priority modifier of 20000 to its subgoal `jump_through_chute`. This goal causes the series of motions that move Wolf into the chute opening, down into the chute, to the output side of the chute and out into the world again. With a priority modifier of 20000 it becomes the highest priority action in Wolf regardless of whether it arises as part of `follow_the_leader` (priority 2000), as part of `react_to_alarmed` (priority 9500), or as part of some other behavior.

Most of the priority modifiers in Wolf make more local changes to priority. An example of such a local priority modifier is shown in the code for `ltf_do_fun_stuff` (in Figure 7.3), where `monitor_follower` is given a priority modifier of 5 allowing it to be given precedence over the other parts of the behavior, but not modifying how the behavior as a whole is scheduled relative to the other activities being pursued by Wolf.

7.5.3 Conflicts

Wolf has 49 pairs of conflicts. Each pair marks two goals or a goal and action as conflicting, and Hap only allows one of them to execute at a time. As described previously, these are in addition to action resource conflicts which are also automatically managed by Hap; two actions which use the same body resources cannot execute at the same time. Wolf's set of conflicts are best described in groups. As already mentioned, `initialize` conflicts with both `set_up_emotion_demons` and `set_up_top_level_demons` allowing it to run to completion before these are pursued.

There are 21 conflicts pairs that together cause only one of these seven activities to be pursued at a time: `amuse_self`, and the activities enabled by the firing of the demons `follow_the_leader`, `rest_if_tired`, `react_to_glum_demon`, `reaction_to_alarmed`, `gang_up_on_demon` and `react_to_threaten`. Each of these demons has a single goal after the wait step that encodes the enabling condition. The behavior for that goal organizes all of the activity when the demon fires, and all permutations of these goal names and `amuse_self` are included in Wolf's conflict list.

Another group of conflicts ensures that Wolf doesn't move while waiting for the follower to catch up when leading in a game of follow-the-leader. This is done by including conflict pairs of `ltf_wait_to_catch_up` with appropriate movement actions and goals. (Of course, if those actions or goals are in service to higher priority goals, they will have precedence and the `ltf_wait_to_catch_up` will be interrupted.)

Similarly there are conflict pairs to prevent Wolf from moving when waiting for the second half of a hey gesture. The goal that recognizes the second half is marked as conflicting with the movement actions. Thus, only high priority movement actions will take his attention away from a greeting once it is captured by the first half of the gesture. When the first part is recognized, Wolf also looks toward the Woggle that might be heying him. The restriction caused by the conflict is short-lived because to be a hey gesture the second part must quickly follow the first. After looking (and stopping movement, if the movement was by a lower priority goal), the recognizer will either complete the recognition, or abort and reset within a short time.

Finally, the long-term movement goals such as `go_to_woggle` conflict with themselves to prevent Wolf from trying to go to two places at once.

7.6 Wolf's Emotions

7.6.1 Important Goals and Other Causes of Emotion

Wolf has a number of goal instances that are emotionally important to him. These goals cause emotions when they arise and succeed, when they fail, and when they are judged by Wolf to be likely to fail.

Many of the goals that immediately arise due to `amuse_self` are important to Wolf. `Explore_wander_for_time` has an importance value of 3. Thus it gives rise to **happiness** of intensity 3 when it succeeds and **sadness** of intensity 3 when it fails (by being interrupted and timing out, for example). `Lead_woggle_for_time` has an importance of 5. In addition to causing **happiness** or **sadness** this goal also causes **gratitude** or **anger** because there is a behavior that places credit or blame on the follower when the goal succeeds or fails. `Dance_on_pedestals_for_time` has an importance of 3 both when it arises directly from `amuse_self` and when it arises in service to `release_your_aggression`. `Threaten_for_fun` has an importance of 2. `Do_the_chute` has an importance of 4. `Follow_the_leader` also has an importance of 5 and, like `lead_woggle_for_time`, credits or blames the woggle that is playing with Wolf for the success or failure, thus giving rise to **sadness** and **anger** or **happiness** and **gratitude**.

Each of these goals describes a "complete" activity of relatively long duration. In order for Wolf to feel emotions at partial success of these activities a number of the subgoals of these large behaviors are also annotated with importance values. For example, the goal `dance_a_bit`, which is not listed, is a subgoal of `dance_on_pedestals` that is repeatedly executed to perform the dance. (For a complete dance it executes about 3 times.) In Wolf, it is annotated with an importance of 1 so that Wolf feels a little bit of **happiness** each time he performs a part of the dance. In this way, he can feel a bit of **happiness** repeatedly, followed by more **happiness** if the dance completes as a whole or **sadness** if it fails after some successes.

Similarly, `copy_squashes` and `copy_jumps` of `follow_the_leader` are both annotated with importance 1. And a subgoal of `lead_woggle_for_time` is annotated with importance 1.

The final annotation of importance in Wolf's active goals is an importance value of 1 for the part of the `hey` behavior that recognizes the acknowledging hey. This is to capture the **happiness** and **gratitude** or **sadness** and **anger** that accompanies social niceties.

In addition to these active goals that are annotated as important, Wolf has four important passive goals. These are `dont_be_hurt_dist` with importance 2, `dont_be_hurt_threaten` with importance 3, `dont_be_threatened` with importance 5 and `dont_fall_behind_in_ftl` with importance 1. `Dont_be_hurt_dist` gives rise to **fear of** a Woggle when that Woggle is disliked and moves too close. This is recognized by Wolf's `fear_being_hurt_distance_demons`. `Dont_be_hurt_threaten` gives rise to **fear of** a Woggle that is threatening Wolf. `Dont_be_threatened` gives rise to **anger** and **sadness** when another Woggle threatens Wolf. Both of these are recognized by Wolf's `recognize_threaten` behaviors.

`Dont_fall_behind_in_ftl` gives rise to **anger** and **sadness** when Wolf falls behind as a follower in a game of follow-the-leader. It is recognized by Wolf's `recognize_falling_behind` behavior.

7.6.2 Behavioral Feature Map

Each of the generated emotions has an intensity that is the same as the importance of the goal that gives rise to it. These emotions are automatically combined as described in Section 6.4.3.

Wolf's behavioral feature mapping is slightly more complicated than those described in Section 6.4.3.

The method to compute them is to first compute temporary values for each. The **act_cheerful** feature is given the value of Wolf's **happiness** emotion summary. **Act_glum** is given the value of his **sadness** emotion summary minus 3. These are because Wolf envisioned personality tends to suppress his sadness and show happiness more. **Act_alarmed** is equal to his **fear** summary minus 2 unless the raw value is above 7 in which case it is the raw value. Again, this is because Wolf tries to suppress and not show his fear. The same computation is done for each of the **act_alarmed by** features with their respective **fear of** emotion summaries.

The envisioned personality for Wolf not only tries not to show fear, but also expresses his fear as aggression. This is done, in part, by the way the **act_aggressive** feature is computed. It takes the maximum value of Wolf's **anger** emotion summary and his **fear** emotion summary minus 1. Thus even if he is not feeling angry, he may have a high **act_aggressive** feature value due to his **fear**. **Act_aggressive toward** the User is computed in the same way using the **anger toward** the User emotion summary and **fear of** the User emotion summary.

Wolf treats Shrimp as a scapegoat, so his **act_aggressive toward** Shrimp feature is the maximum of all of the possible **act_aggressive toward** feature values for the User, Shrimp and Bear if they were all computed as the User's is. Because Wolf is wary of the much larger Bear, his **act_aggressive toward** Bear feature is always zero. Any value that it might get due to his **anger toward** Bear and **fear of** Bear can only show up transferred to his **act_aggressive toward** Shrimp (if it is the largest).

After each of these temporary values are computed, the final values are computed by choosing the feature with the largest value of the **act_cheerful**, **act_glum**, **act_alarmed** and **act_aggressive** features. This feature is chosen as Wolf's dominant emotion and the values of all other features are reduced by 2/3 (i.e., multiplied by 1/3). The dominant feature and any directed version of that feature (for example, **act_alarmed by** if **act_alarmed** is the dominant feature) are not reduced. This was intended to make Wolf be more clearly emotional by strongly showing one dominant emotion at a time. It also is intended to give him more mood swings than using the temporary values directly would give. When he has two strong emotions, one wins, with the other being greatly reduced in strength. Slight changes in the balance between these raw emotions can cause large swings in the expression of emotions as the dominant emotion switches between **act_aggressive** and **act_alarmed**, for example.

Finally, Wolf has a notion of *energy* that is similar to emotions and behavioral features, but different. **Physical energy** in Wolf is expressed as a number that is reduced by activity and the passage of time, and increased by sleeping. If **physical energy** were an emotion, then **energy** would be the behavioral feature correspondent to it. In Wolf, its base value is his **physical energy**, but it is increased by any **happiness**, **fear** and **anger** that he is feeling, and it is decreased by any **sadness** that he is feeling. **Physical energy** is reduced over time by the `em_decay_demon` that decays emotions. It is increased by Wolf's `sleep` behavior, and the **energy** feature is updated by Wolf's behavioral feature map.

7.6.3 Emotional Expression

All of the behavioral features and emotions are available to any of Wolf's conditions (preconditions, context conditions and success tests) to enable him vary his behavior based on his emotional state. As described previously, these features and emotions play a role in the enabling conditions of Wolf's effective top-level goals. They are also used to choose behaviors for goals that Wolf has already decided to pursue. For example, the behavior for `amuse_self` that gives rise to `release_your_aggression` is only chosen when Wolf has an **act_aggressive toward** feature with intensity 3 or greater. In that case, it chooses the Woggle that Wolf is feeling most aggressive toward to direct the goal at. The `threaten` behavior itself has as its precondition that Wolf's **angry at** emotion have an intensity greater than 3. The behavior arises due to his **act_aggressive toward** feature, so it is possible that he will start the preparation to threaten another Woggle for fun or gang up on another Woggle, and by the time he gets there he might decide that he doesn't have it in him, because he wasn't feeling angry enough (his aggression might have come from fear instead of anger, or his anger may have decayed since the behavior was triggered). The behavior and goal would then fail, and Wolf would go on to something else.

Just as emotions can help trigger activities, they also are used to signal the end of activities. When moping, for example, there is a success test around the main moping goal that causes it to end if Wolf is no longer sad (if **act_glum** drops to zero).

Also, Wolf's emotional state can have several indirect effects. For example, as described in the detailed description of Wolf's `lead_the_follower` behavior (Section 7.2), when leading, the amount of attention Wolf pays to his follower in the game is determined continuously by how aggressive he is feeling toward him. Also, the **energy** feature, which is a product of both **physical energy** and Wolf's emotional state, influences many aspects of his behavior, from how fast and high he jumps when moving around the world, to when he sleeps and when he wakes up. The emotional influence on this feature allows the User (or other Woggles) to make Wolf too excited to sleep by threatening him on his way to bed. The **anger** and **fear** temporarily raise his **energy** level, and it takes time for him to "settle down" enough to again decide to go to bed.

7.7 Where's the Personality? — Everywhere

At the beginning of this chapter, I described the personality we were trying to capture when building Wolf. In the above description there have been many explicit choices that were made to help capture this personality in interactive form, and I hope that these descriptions give some idea of the process of building an interactive personality. Nevertheless, I have not yet directly addressed the question of where we encoded Wolf's personality in the constructed agent.

The answer is — everywhere.

The content described in each of the areas above was all tailored for the expression of Wolf's personality. In the following sections I describe how the different areas were tailored to express Wolf's personality. This description is illustrative rather than exhaustive, because to describe all of the decisions would result in an oppressively large description, and would duplicate much of the information already described.

7.7.1 Activities

The activities the Woggles engage in partially define the social context in which they exist. Nevertheless, for a particular personality, the particular activities it engages in must only be those which are appropriate to the personality. Wolf, being an aggressive, self-centered character, does not have the activity of "cheer-up other". Wolf participates in all of the other activities, but how and when he participates depends on and helps to reveal his personality.

7.7.2 Existence of Behaviors for Action and Composite Sensing

While there is much overlap in the set of behaviors among all of the Woggles, Wolf both includes behaviors appropriate to his personality that others don't include, and omits behaviors of others that are not appropriate for his personality. For example, Wolf is the only agent that has a `threaten_for_fun` or `gang_up_on_fight` behavior. The other agents never express their amusement by picking a fight, and their reactions to fights in the world is quite different than Wolf's. Where Wolf sees a fight as an exciting opportunity, a fight causes Bear to become sad and perhaps try to stop it. Observing a fight for Shrimp is a cause of fear, and he will never become involved unless he is the victim. Similarly, Wolf does not include certain behaviors that would be foreign to his personality. For example, he does not have behaviors to recognize when a Woggle is moping, to cheer up a sad friend, or to invite the human user to participate in the world.[4] All of these would be too actively friendly for Wolf's personality. Where Bear or Shrimp might use their view of the world outside the screen[5] for a dance to attract the attention of a person who is just watching and not participating, Wolf only uses this information to stoically observe the person.

7.7.3 Expression of Behaviors

Even when Wolf has the same behaviors as the other Woggles there are often differences in the details of his behaviors to express his personality. For example, in the `goto_bed` behavior, Wolf randomly chooses which of the two resting rocks to sleep on without any regard to other Woggles that might be sleeping there. Shrimp, in contrast, prefers resting rocks on which his friends are sleeping, and avoids those that Woggles he doesn't like are on. These details can appear at all levels of Wolf's behavior. For example, in the low-level details of `lead_the_follower`, the time between runs of the `monitor_follower` behavior that is based on how aggressive Wolf is feeling toward the follower was designed to express Wolf's personality. It allows him to callously lead faster than his follower can keep up, and then become angry when his follower cannot catch up.

7.7.4 Organization of Behaviors

Wolf's effective top-level goals were chosen to express his personality. As already mentioned, the specific sensing goals that he has in his effective top-level are those that are appropriate for his world view, namely that moping by a Woggle, threats from Shrimp, greetings from Shrimp, etc. don't need to be recognized. Similarly, the choices of which action-producing or emotion-producing effective top-level goals to include is based on his personality. Wolf both has top-level goals that other Woggles don't have and explicitly excludes top-level goals that would be inappropriate for his personality. For exam-

[4]Using the sonar sensor, Shrimp and Bear can tell when there is a person observing the world but not controlling the User Woggle. They then might engage in behavior to greet the person and entice him or her to participate.

[5]Sonar sensors are used to give the location of the human observer or participant.

ple, he is the only Woggle with `gang_up_on_demon` as a top-level goal, and he doesn't have the top-level goals of `console_glum_friend_demon`, `greet_the_user_demon`, or `save_threatened_friend_demon`.

The behaviors organized by Wolf's effective top-level goals (listed Figure 7.10) were also determined by the personality we were trying to build. For example, both Wolf and Bear have the top-level goal `react_to_threaten`, but in Wolf this goal organizes the aggressive behaviors `threaten_when_threatened`, `threaten_other`, etc. In Bear, this goal organizes a behavior that causes Bear to roll his eyes at the one threatening him. (This behavior is not present in Wolf.) Similarly, the particular goals and behaviors that were chosen as ways for Wolf to pursue his `amuse_self` goal were guided by his personality.

The priorities and enabling conditions listed in Figure 7.12 were also chosen to express Wolf's personality. Since he is an aggressive personality that doesn't like to show fear or sadness, his behaviors that recognize and express both fear and sadness have relatively lower priorities. For example, his goal to `react_to_glum` has priority 3000 while Shrimp's has priority 6000. The enabling conditions were also similarly crafted. For example, his threshold for running away when alarmed is much higher than Shrimp's, and he doesn't ever threaten Bear back (because he is intimidated by him) while all of the others react to threats by any Woggle.

7.7.5 Wolf's Emotions

All of the components for specifying what Wolf feels emotional about and how he expresses those emotions are part of his personality. In general, goals that primarily generate **fear** (because of behaviors in Wolf that infer likelihood of failure for them) were given lower importances in Wolf than in other Woggles. Goals that generate **anger** were given higher priorities by Wolf's authors. As another example, because Wolf likes to show off, the goals for directly showing off, `dance_on_pedestals` and `do_the_chute`, are more important to him than to the other Woggles. Wolf also has fewer goals to infer the likelihood of failure to cause **fear** than Shrimp and more than Bear.

Wolf's behavioral feature map is tailored to his personality as well. He is the only Woggle that suppresses **fear**, maps **fear** to **act_aggressive**, and that tends to **act_aggressive toward Shrimp** whenever he is feeling **angry toward** other Woggles.

7.8 Four Minutes with Wolf

In Section 4.15 we've seen a description of Wolf's execution that focuses on the details of Hap's execution. Because of the detail, that description is necessarily of a very short excerpt of behavior. In this section I want to convey some of the feeling of interacting with Wolf for a longer period of time. This excerpt was taken from a run of the system with a human user interacting with the Woggles and paying particular attention to Wolf. The description is less detailed than that of Section 4.15, but focuses on how the structures described in this chapter combine to make a whole agent over time in an interaction.

Action	Frequency	Action	Frequency
Jump	109	**StartFaceWoggleEyes**	19
Put	18	**StartLookPoint**	86
SquashDir	0	**StartLookWoggleEyes**	0
Squash	66	**StartTrackWoggleEyes**	187
SquashHold	46	**StopFace**	80
SquashDirHold	22	**StopLook**	137
SquashRelax	68	**Say**	0
Spin	35	**ChangeBodyRadii**	60
SpinTo	87	**ChangeColor**	0
SpinEyes	18	**CloseEyes**	36
SpinEyesTo	18	**OpenEyes**	66
ElevateEyes	18	**StartTremble**	0
ElevateEyesTo	18	**StopTremble**	0
StartFacePoint	27		

FIGURE 7.13: Frequency of primitive actions performed by Wolf in four minute excerpt of interaction.

The trace of the system for the four minutes of interaction is more than three megabytes of data. (Tracing only Wolf's Hap execution, but the actions and emotions of all four Woggles: Wolf, Shrimp, Bear and the user-controlled Woggle.) Wolf executes 5042 goals, actions and mental actions in those four minutes, many of them concurrently. There are 3423 actions executed by the four agents during this time. Wolf executes 1221 actions. Figure 7.13 shows how many of each action is executed by Wolf. The actions are listed in the order in which they are described in Chapter 3. The action **Say** is not used at all because this version of Wolf uses no natural language. The use of natural language in an agent is described in the next chapter.

With this much activity, it is impossible to give the same detail as given in the previous short example in Section 4.15. At the start I will give more detail (although still less than in the trace or previous detailed example). Toward the end I will give less detail, trusting that the reader can fill in some of the details from the understanding of the previous trace and the introduction of this trace.

At the start of this excerpt Wolf is pursuing his `amuse_self` goal by jumping through the chute. As he is flying through the air toward the chute, the time-based enabling condition for `em_decay_demon` fires, his emotions are decayed and new values for his behavioral features are computed. The **Jump** action is followed by a **Put** to the exit of the chute, and another **Jump** out of the exit chute. The goal that causes these three actions in sequence has a priority modifier of 20000 that causes it to be higher priority than any other goal in Wolf. It also conflicts with all of the movement-causing actions. Thus it will not be interrupted.

When he lands from the chute exit, his `amuse_self` goal is again chosen to be executed.

This time he randomly chooses to pursue it by dancing on the pedestals. As he is dancing on the pedestals, the User moves to where he is and tries to threaten him. Dancing on the pedestals is a quick behavior with a lot of large movements, so it is hard to get close to Wolf for long enough to perform a puff movement or three quick movements. Wolf jumps away right before the User's first attempt, and User threatens the air. The User repositions to where he thinks Wolf will jump next and puffs as Wolf is jumping through the air. This threat is recognized by Wolf, and causes the enabling condition of his `react_to_threaten` goal to fire. This behavior interrupts the previous dance behavior and is marked to conflict with it, so Wolf stops dancing and starts exchanging threats with the User. As he lands he has already started threatening gestures in the User's direction. He steps up the attack when he is on the ground and can move closer and threaten directly in the User's physical space.

As Wolf is threatening the User, Bear notices that one of his friends (the User) is being threatened and responds by trying to save his friend and break up the fight. (These are Bear's `recognize_threaten` and `save` behaviors.) He does this by positioning himself between the User and Wolf and threatening Wolf. The User continues to attack for a while and then stops to watch.

The initial threaten by the User and subsequent threatens by the User and Bear are recognized by Wolf's `recognize_threaten` sensors and generate **fear**, **sadness** and **anger**. **Sadness** and **anger** are caused by the failure of Wolf's passive goal `dont_be_threatened`, and the **anger** is directed at the Woggle doing the threatening. The **anger** is directly mapped into **act_aggressive toward** that Woggle. **Fear** is caused by Wolf's believing that he might be hurt; the evidence that he has is that he is being threatened currently. This is expressed by Wolf inferring that his passive goal `dont_be_hurt_threaten` is likely to fail. This causes two **fear of** emotion that are directed toward the two Woggles that are threatening him. At this point, the combined **fear** (summary) from the threats is less intense than 7, so it is mapped to **act_aggressive** by Wolf's behavioral feature map, and he is expressing mostly aggression at this point. Wolf continues to threaten the User, and Bear continues to threaten Wolf and position himself between Wolf and the User.

When Wolf's `react_to_threaten` completes, the demon that enabled it completes, and its parent goal resets. The demon behavior is again placed in the tree as the child of this goal, but it doesn't fire again because the only one who is currently threatening Wolf is Bear, and Wolf's enabling condition for this behavior explicitly excludes Bear. (Wolf's envisioned personality is intimidated by Bear and doesn't try to confront him unless necessary.)

This allows Wolf's `amuse_self` goal to be pursued. Because his **act_aggressive** behavioral feature has a large value, he chooses to pursue `amuse_self` with the goal and behavior `release_aggression`. This behavior is pursued with an argument directing it toward one of the Woggles that Wolf is feeling most aggressive toward. At this moment, Wolf is most angry at Bear and Shrimp, Bear directly because he has been threatening him, and Shrimp because of the way Wolf's behavioral feature map works. He always feels as aggressive toward Shrimp as the maximum level of aggression he feels toward any other Woggle. The behavior randomly chooses Shrimp from the two choices, and Wolf heads off to do a `threaten_for_fun` toward Shrimp. The processing time between finishing his `react_to_threaten` goal and heading off to threaten Shrimp for fun is 100 milliseconds, so it looks nearly instantaneous.

Bear continues to position himself between Wolf and the User until Wolf moves away, and his `save_friend` goal succeeds. Wolf quickly (a couple of seconds) arrives at Shrimp and starts threatening him. Bear notices the threatening, and moves to save Shrimp, who is also his friend. Wolf continues the `threaten_for_fun` behavior, trying to stay close to Shrimp and perform threatening movements as Shrimp runs away and Bear gets between Shrimp and Wolf.

As this continues, Wolf's level of total **fear** is rising due to the continued threats from Bear. Eventually the intensity of this summary is greater than 7. Wolf's behavioral feature map then makes a large change. **Fear** is no longer reduced by 2 in the mapping, and it is directly mapped to **act_alarmed by** instead of **act_aggressive toward**. Suddenly Wolf has a large intensity **act_alarmed** behavioral feature and a much less intense **act_aggressive** behavioral feature. (The **act_aggressive** feature is lower because **act_alarmed** now dominates and all other behavioral features are reduced by 2/3.) This large **act_alarmed** feature causes the enabling condition for `react_to_alarmed` to fire. This behavior causes Wolf to pursue his `run_away` behavior.

As Wolf runs away, there are no more causes of **fear** and the automatic decay of emotions continue decreasing his **fear**, **anger** and **sadness**. The `run_away` behavior takes about 25 seconds to execute. Wolf runs away from Bear during this time, occasionally pausing to execute his `look_around_nervously` behavior as he goes. When he finishes his `run_away` behavior, his **fear** intensity is below 7 and the normal suppression and mapping of **fear** to **act_aggressive** takes place in his behavioral feature map. `Amuse_self` is again chosen, and because he is still feeling sufficiently aggressive, `release_your_aggression` is again chosen as the method for amusing himself. In this case, however, he randomly chooses `dance_on_pedestals` instead of `threaten_for_fun`. He jumps over to the pedestals and starts to dance.

During all of this time, Wolf's **physical energy** has been decaying. (As mentioned above, it is reduced over time at the same time emotions are decayed, and it is increased by the sleeping behavior.) His behavioral feature for **energy** is a combination of `physical_energy` and the intensity of his emotions, which have also been decaying. While he is dancing, his **energy** behavioral feature drops below the threshold recognized by the enabling condition of `rest_if_tired`. This behavior starts, and Wolf starts jumping toward the resting rocks. He chooses an empty rock (the other one holding a sleeping Bear), and goes to sleep by closing his eyes and "breathing" slowly (executing appropriate slow **Squash** actions). In addition to the "breathing" movements, his `sleep` behavior increases his **physical energy** periodically as it executes. If his sleep is interrupted he only gets the increase in **physical energy** gained up to that point.

At this point about two and a half minutes have elapsed since the start of the excerpt.

When Wolf wakes up, none of his demons fires and he pursues his `amuse_self` goal twice in a row. First he jumps through the chute, which is a large and energetic movement since he is on one side of the world and jumping to the chute entrance on the other side. He happens to have the right amount of speed when he goes into the chute, so when he comes out flying through the air, he hits the opening again, going through the chute twice on the same jump.[6]

After completing his jumps through the chute, he chooses to `watch_woggles_for_time` for his `amuse_self` goal. He jumps toward the hills on the upper left of the world and turns to watch the Woggles. As he is sitting and watching various Woggles, the User comes up to him and invites him to play a game by performing a `hey` gesture and then jumping away. This is recognized by Wolf's `recognize_and_interpret_hey` behavior and the decision about whether to play is made by the enabling condition for `follow_the_leader`. The enabling condition fires, and Wolf starts playing. The composite sensing behaviors `recognize_and_remember_puts`, `recognize_and_remember_jumps`, `recognize_and_remember_squashes` and `recognize_falling_behind` are all instantiated and start observing and recording the activity of the User. The behaviors `copy_squashes`, `copy_jumps` and `catch_up` are all demons that respond the the information recognized by the other behaviors and follow the activity of the User as best as Wolf can. The User leads at a moderate pace and Wolf keeps up without ever falling behind, until the User jumps through the chute. Because the User lands far away from Wolf when exiting the chute, Wolf is far behind, and has to jump to where the User is to continue the game. This is recognized by Wolf's `recognize_falling_behind`, and he attempts to catch up while following by using his `catch_up` behavior. They play for a while more, and the User ends the game by performing a `hey` gesture again toward Wolf. This gesture is recognized by Wolf's `recognize_and_interpret_hey` behavior, and the recognized gesture causes a success test on his `follow_the_leader` goal to fire. This success test causes the `follow` goal to end successfully.

The successful completion of the `follow_the_leader` goal (caused by the success of the `follow` goal) causes **happiness** and **gratitude** toward the User because it is an important goal, and Wolf has a behavior that attributes credit for a successful `follow_the_leader` goal to the leader. Wolf again chooses to pursue his `amuse_self` goal and his and the other Woggles' activities continue.

In this excerpt there are necessarily many details that have been left out. These details are important for making Wolf seem believable. For example, the many times that Wolf blinks and the movements of his eyes that he makes before doing something or when recognizing something are all critically important to him seeming alive, but they would both bog down the above description and make it too lengthy were they to be included. The timing and movement of the activity is also important for believability, but difficult to convey in a description such as this. Some idea of these details can be gotten by looking at the detailed description in Section 4.15. Nevertheless, I hope the above description gives some idea of how all of the pieces of a complete agent fit together during execution.

[6]The chute exit is pointed at the opening so this is possible. The computation of whether the amount of energy is correct is based on the length and duration of the jump into the chute. The computation and series of jumps and puts to perform the jumping through the chute is done by the `do_the_chute` behavior that is shared by all of the agents including the user-controlled agent.

Chapter 8

Natural Language Production for Believable Agents

This chapter presents the results of my efforts to allow Hap-based agents to communicate using natural language while addressing the needs of believability. It includes attempts to understand the needs of believability for natural language generation, extensions to Hap to directly support natural language generation, and tests in two example agents that use this as part of their behavior.

This work is preliminary. More work needs to be done to see if the approach can be used for complete believable agents, but the results here suggest that this may be a fruitful path to continue to pursue for believable agents that use language.

8.1 Introduction

Talking is not a requirement for believable agents; believable agents can be built without the ability to use natural language. This is clear if one looks at characters in the arts that don't use language. The characters in the award-winning short films by Pixar [Lasseter 1986; 1987b; 1988; 1989], the character of Grommit from the popular Wallace and Grommit films [Park 1992; 1993; 1995], and the flying carpet from Disney's film *Aladdin* [Clements and Musker 1992] are all rich examples of characters with strong, vivid personalities that don't use language.

Even though believable agents can be built that don't use language, it is a simple truth that most characters in the arts talk. To be able to build autonomous agents like these characters, any believable agent architecture and theory must address the issue of language use.

There are two additional reasons one might want to address language production for believable agents. First, it is a powerful avenue of expression. What a character says, how he says it, and how that combines with everything else he is doing does much to tell us who that character is. This is not to say that language is *the* avenue of expression. In fact, experience in the arts suggests that often strong expression can be had by using little language and allowing the other avenues of the character's expression, face, body movements, eyes, to show through [Giannetti 1987, p. 178] [Thomas and Johnston 1981, p. 471]. Nevertheless, language can be a powerful tool among those available to an author for building a believable agent.

Second, and perhaps especially important in interactive agents, language can aid in the accessibility and understandability of created personalities. Many types of interactions involving people or characters in the arts naturally involve language. Inviting someone to play a game, greeting a friend as a prelude to other interaction, cheering up a sad friend, all are naturally expressed using some language. It is possible to perform them all without language; mimes and animals do it all of the time, and, in fact, as described in Chapter 7, Wolf and the other Woggles do all of these without language. When doing these behaviors without language, however, body motions, gestures and context must be used for the necessary communication. In my experience with the Woggles, this makes it more difficult for people to immediately understand and interact with the agents. They first have to understand something about the culture of the agents, for example, that to invite one to play a game of follow-the-leader one greets the other and jumps away. Such social conventions seem natural after they are known (and we tried to make them as natural as possible), but they are not as natural as using language to say "let's play". With language, it is easier to build agents that follow the social conventions of *our* human world, and are therefore immediately accessible to people. Without language it is sometimes necessary to invent artificial conventions, an artificial social context, that must be learned before people can understand and fully interact with the agents.[1]

Language production has been widely studied in computer science in both natural language text generation and speech synthesis. The majority of research in these areas does not take believability as a primary goal. (There has been notable work that focuses on sub-problems of believability for example [Hovy 1988; Cassell *et al.* 1994; Walker *et al.* 1997]. Most of these works, however, focus on realism and general-purpose language and not on the task of expressing a particular autonomous personality that seems alive.)

When believability is taken as a primary goal, the task of language production is affected in wide-ranging ways. To understand this impact, let us consider what requirements believability places on language production. To ground these requirements, let us examine four seconds from the film Casablanca [Curtiz 1942], which are transcribed in Figure 8.1. In this scene, Ugarti (Peter Lorre), a dealer in the black market, is being arrested for stealing two letters of transit. Just before the police haul him away, he seeks help from Rick (Humphrey Bogart), the seemingly cynical owner of the Café Américain. Speech and action occur simultaneously, and they are transcribed into two columns to show the parallelism.

In these moments, Ugarti is a very believable and engaging character. If we wish to build autonomous agents that can produce similarly believable language use and behavior, we must understand and capture the important qualities of his behavior. The following

[1]To fully gain this advantage of accessibility, the agents must understand natural language as well producing it. Such full understanding is beyond the scope of this thesis. In this work, the agents use keyword matching along with behavior sensors as described in Section 5.2 for limited understanding.

Speech	Action
... Ugarti enters yelling "Rick! Rick! Help me!", puts his hands on Rick's forearms. Rick pushes Ugarti against a column saying "Don't be a fool, you can't get away."	
But Rick, hide me!	U's eyes are wide, focused on R, U has facial expression of extreme desperation and fear.
Do	U's eyes and then head turn left to see approaching police, mouth tight, face tense.
something,	Head, eyes back on R, intense gaze, "something" emphasized.
you	Eyes then head turn a bit left toward police as they grab him.
must	U's face compresses in pain.
help	Shrinks down, looks further away from R.
me	Twists to get free.
Rick!	Looks back at R, but eyes pressed shut, looks away as police pull at him.
Do something!	U looks toward R as he speaks, then away in pain as he is dragged from scene yelling.

FIGURE 8.1: Transcript of moment from Casablanca.

seem to be important to the power of this excerpt, and they echo the requirements for believability described in Chapter 2. After each observation, I note the general requirements for believability that it relates to.

1. In general, production (and understanding) of language and action appear very tightly integrated. I feel this probably is not the result of distinct sensing, acting, understanding, and generating modules communicating through narrow channels.

   (This appearance of tight integration is similar to the general requirements for believability that the agents be **broadly capable** *and* **well integrated***.)*

2. Action and language are used together to accomplish communication goals. An example is pleading language with a wide eyed facial expression.

   (This is an instance of **concurrent pursuit of goals and parallel action***. In this case, the parallel goals are coordinated for a unified purpose.)*

3. Language generation occurs in parallel with other independent goals. Parallel behaviors producing streams of control signals to multiple channels (eyes, body, voice) help bring the character to life. Ugarti is generating language while watching and struggling with the police.

   (This is another example of **concurrent pursuit of goals and parallel action***.)*

4. All of the channels of expression work together to present a coherent, consistent message. Ugarti's eyes, body, voice all express his fear and plea for help.

   (This is exactly the same as the requirement of **consistency***.)*

5. Perception and action occur as subgoals of generation. For instance, as the transcript begins, Ugarti yells "Rick" because he perceives that Rick is not attending to him. He acts by putting his hands on Rick's arms to signify an embrace of friends, presumably to increase the persuasiveness of his words. I believe both of these arise most naturally as consequences of generation.

   (Language like the rest of the agent should appear **situated** *by including sensing in decisions. That action can be used as subgoals of generation is an example of the needed appearance of* **tight integration** *for believability.)*

6. Generation does not always reduce the agent's responsiveness to events in the world. (Some reduction to correspond to the fact that generation takes thought is appropriate.) Ugarti notices and responds to the police approaching, grabbing him, etc. all while producing one short sentence.

   (Believable agents are **responsive***.)*

7. Generation is reactive to changes in the world. If the police ran away to pursue some more important criminal, it would be strange if Ugarti did not react to this important change and continued to plead.

   (Believable agents need to be appropriately **reactive***.)*

8. Pauses, restarts, and other breakdowns are desirable when they reflect the personality and situation of the agent. In the fine scale timing of the transcript, the actions of the police absorb some of Ugarti's attention and noticeably vary his rate of speech production.

   (Believable agents need to appear appropriately **resource-bounded***.)*

9. Generation is incremental. Word choice and other generation activities seem to be influenced by the real-time flow of events just as other action production is.

   (This is another instance of **situated** *language use.)*

10. Language varies by social relationship. The social relationship between the speaker and those spoken to should influence language production as it influences the rest of the agent's behavior. That Ugarti looks up to Rick and considers him somewhat as a friend is evident both in the fact that he goes to him for help in the first place and by the way he talks and moves even when the police are descending on him.

    (This is a one-sided version of the requirement of **social relationships.***)*

11. Language production is influenced by the goals of the agent. Ugarti's words are clearly the result of his goal to get away from the police.

    (This is an instance of the believability requirement, **appearance of goals***.)*

12. Language production generally follows the social conventions of the characters' world.

    (This is the same requirement as **exist in a social context***.)*

13. Language generation, like other action, varies with emotional state. Ugarti pleading with Rick is in accord with his emotions upon being arrested.

    (This is half of the requirement of **emotion:** *believable agents must show emotions in some way.)*

14. Language generation is also specific to the personality using it. It is unlikely that Rick would express himself in the same way as Ugarti even were he to be in the same situation. If this happened, it would greatly change the audiences perception of who Rick is.

    (This is the same as the most important requirement for believable agents: **personality***.)*

15. Emotion is produced from the success and failure of communication as well as of other action. For instance, Ugarti is upset about not escaping the police, but this failure is partly a consequence of not having enough time to convince Rick to help him. So he is angry and sad about his inability to achieve a communication goal, as that was his means to achieve an important parent goal. Anger in response to being constantly interrupted is another example of this phenomenon.

    (This is the other half of the requirement of **emotion***: believable agents must appear to have emotional reactions.)*

These observations from the brief excerpt with Ugarti illustrate instances, specialized for language production, of all of the requirements for believability from Chapter 2 except for *self motivation* and *change*. While it is easy to see why these two might not appear in such a short excerpt, it is also clear that they are just as important to the language aspect of an agent as they are to the whole agent. If an agent has internal motivation, language is as natural a place as any to show it, and an author should be able to build agents with this property. Likewise, as a believable agent changes over time, it may be desirable to show that change through language.

8.2 Text or Speech

The most obvious way to approach this problem is to build agents that have speech as one of their capabilities. Speech, however, introduces multiple technical challenges in the context of believable agents that are beyond the scope of this thesis. These include natural, human-quality speech that is in a "voice" appropriate to the character being built, and emotion-based influence on the sound stream produced. Most current speech synthesis is still easily recognizable as computer generated. Of those systems that approach human quality, the "voice" of the speech is limited to one or a few voices. Variation in the voice due to how relaxed, hurried, tired, energetic, etc. the agent is at a given moment is very limited.

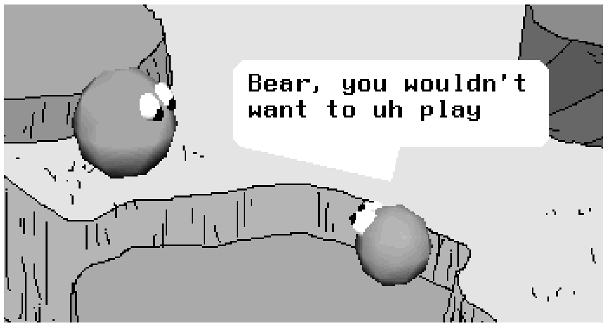


FIGURE 8.2: Woggles that "speak" text bubbles.

There has been some work on emotion-based variation of the speech stream, notably the work of Janet Cahn [Cahn 1989], but this is only now reaching a point where it might be possible to include in an autonomous agent [Walker *et al.* 1997].

Without speech the agents are limited to text communication. Nevertheless, in order to address the requirements raised above, it is important to maintain a conversational, speech-like, aspect to this communication. I want to allow the agents and people interacting with these agents to use language in the spontaneous, fluid manner of verbal conversations rather than the formal style of much of written language.

To allow this I extended the original Woggles domain (which is described in [Loyall and Bates 1993]) with text communication through voice bubbles. A snapshot of that world displaying a voice bubble is shown in Figure 8.2. My goal in this extension was to allow, as much as possible, a conversational metaphor to be used in the text communication of the agents (and human user). The full extended domain was described formally in Chapter 3. Here I describe the properties of conversational communication this domain allows.

Each agent produces language in its own voice bubble. These bubbles are kept close to the agent's physical body to allow, as much as possible, a person to see both the physical motion of the agent and what the agent is saying at the same time.

At any given moment, an agent can produce a text string of any length between 1 and 15 characters[2] by using a **Say** action as described in Section 3.1. These strings are added to the voice bubble in turn. Once a language fragment is added to the voice bubble there is no way for an agent to remove it, just as in speech there is no way to retract an utterance.

The timing of an agent's communication can be observed by watching the timing of the language added to the voice bubble. Because people cannot read instantly, as they can and do absorb verbal speech, the text added to a voice bubble persists for a few seconds. This persistence introduces a potential misunderstanding that does not exist in normal communication: if the interactor does not see pauses in the text as they occur, he

[2]The limitation to 15 characters per act is not significant. Multiple **Say** actions can be rapidly issued in sequence. Practically, the **Say** actions issued as a result of people typing in real-time rarely have more than two characters per act. Similarly the agents typically have only a few characters per act. Words and phrases are constructed over time by a sequence of actions, preserving the timing information present in the typing or generation process.

would miss pauses and other timing information entirely. To partially address this, the voice bubble automatically introduces a "." character whenever a significant pause occurs (currently .7 seconds). This allows one to perceive some of the timing of the utterance when looking back on the text as it remains in the voice bubble.

The bubble is automatically scrolled as necessary, and text is removed from the bubble after it persists for a few seconds.

The human interactor can say whatever he wants by typing. All of the characters (normally one or two) typed during each video display frame (normally a tenth of a second) are added into the voice bubble of the User Woggle.[3]

This test-bed preserves many of the informal properties of real-time conversations that I want. One can see the timing of the language as it is produced, along with whatever the agent is doing at the same time. Agents can interrupt each other, talk at the same time, or take turns as in normal conversation. And language, once produced, cannot be undone.

There are definite limitations to this approach as a substitute for verbal communication, however. People aren't as competent at reading text and observing action simultaneously as they are at hearing speech and observing action. Also, the persistence of an utterance beyond the moment of utterance is a somewhat unnatural convention compared to speech. The time limit on persistence and automatic introduction of one or more dots when pauses occur is an attempt to preserve some of the conversational properties in this framework.

Even with these limitations, this framework seems to capture some of the flavor of conversational communication and is a useful test-bed for real-time, embodied language without dealing with the additional complications of speech. Text may also have some additional benefits to the pursuit of believable agents that use language. Text can allow people to imagine the voice of the agent they are interacting with, along with appropriate intonations. It remains to be seen what artistic power this approach to conversation can have when properly used.

8.3 Expressing Language Generation in Hap

In this section I describe how I have extended Hap to allow the direct expression of natural language generation.

I chose to base my language approach on the Glinda generator developed by Kantrowitz [Kantrowitz 1990; Kantrowitz and Bates 1992] for three reasons. First, Glinda is an *integrated* generator, with the same generation engine used for both text planning and realization, and there appeared to be similarities between the Hap execution engine and Glinda's generation engine that suggested that a fundamental merging might be possible. Second, Glinda outputs text incrementally, which is an important part of allowing reactive, responsive and situated language production. And third, Kantrowitz's current research

[3] Again, this communication is currently only understood by keyword matching. The user can communicate using a combination of what is understood by keyword matching, combined with context and other activities that are recognized by the agent's sensing behaviors (Section 5.2). The same mechanisms are used by each Woggles when it is spoken to by another Woggle.

[Kantrowitz in press] is concerned with exploring pragmatic variation for natural referring expressions. In the future I may want to incorporate this work to allow agents to generate more natural text.

There are two main challenges to expressing text generation in an action architecture such as Hap. The first challenge is a knowledge representation issue. Glinda's generation goals are significantly more structured than those normally used by Hap. In order to pursue generation in this action architecture I had to extend it to handle the central properties of Glinda's structured goals in a way that helps the integration of generation with the rest of a Hap agent. The second challenge is how to allow the encoding of the grammar. Glinda includes four different types of processing knowledge as rules that fire in stages. In an action architecture such as Hap one must express this knowledge in terms of goals and behaviors in some way that does not make encoding this knowledge onerous. In addition, it is important to allow this direct expression without compromising properties of Hap that are important for believability, like Hap's reactivity, and real-time properties.

8.3.1 Example and Overview of Expression of NLG in Hap

Before going into detail about how I have extended and used Hap to directly express natural language generation, I want to give a preview of how language generation is done in Hap. I first present a simple example of generation in Hap, and I then present an overview of how Hap's mechanisms are used for the main parts of the generation process. This introduction only describes the basics of natural language generation in Hap. Sections 8.3.2 and 8.4.1 give a detailed description, and Sections 8.4.2, 8.5 and 8.6 describe the advantages of this approach to believable agents.

Simple Generation Example in Hap

Imagine that Bear wanted to describe to Wolf Shrimp's **like** relationship with Wolf, which Bear thinks has a value of -3 (indicating that Shrimp dislikes Wolf with a good bit of intensity).

In the approach to generation presented here, the natural language generation processing proceeds as follows. (Note that this example only includes the natural language generation elements, as if they were the only part of an agent. To see a more detailed example that shows how NLG is incorporated with the rest of an agent, see Section 8.5.)

First, Bear would issue a generate goal with this concept as an argument of the goal. The concept would have various parts, in particular: that the concept is a like relationship; that the value is -3; that the object of the like is Wolf; and that the actor of the like is Shrimp. It might also include other information, such as a modifier to downplay the intensity, or the information that Wolf is the person being spoken to.

In this case, a sequential behavior for the `generate` goal is chosen that generates the concept as a declarative sentence. It does this by issuing three `generate` subgoals, each with an argument that is one of the parts of this concept, in the order: the actor concept, the like concept, and the object concept. Other behaviors that could have been chosen include ones to generate this as a sentence in passive voice, as a question or as a subordinate clause. The first two are chosen when appropriate modifying information is included, and the last might be chosen when a the concept is part of another concept. (An example of this is given in Section 8.5.) Each of these might use other orderings of the parts, as well as possibly including other modifications.

These subgoals are executed by Hap in order, starting with the actor concept. The actor concept is generated by other behaviors for the `generate` goal, eventually resulting in the execution of a **Say** action with the argument "Shrimp", which causes this to appear in Bear's voice bubble. The behavior then returns the number and person of the generated noun, in this case `singular` and `third`.

The next goal is a `generate` goal with the **like** concept as its argument. Because the **like** concept is not a linguistic data structure that can be directly generated, the behavior that is chosen to pursue this goal first issues a subgoal to infer a linguistic expression of this concept. After that goal returns, the behavior's second step causes a recursive generation of the chosen linguistic expression by issuing a `generate` subgoal with the chosen linguistic concept as its argument. The third step of this behavior is to return the chosen linguistic concept so that the choice that was made, and has been "said" by the agent at this point because of the recursive generation in the second step, will persist for later steps in the generation process. This is important if the like concept were to be generated twice in a way in which the same linguistic choice should be used.

The behaviors for this special-purpose inference goal can choose among linguistic concepts such as `<negated like>`, `<negated is fond of>`, `<hate>` and `<dislike>`. In this case, because of the modifier to downplay the intensity, `<negated is fond of>` is chosen.

This recursive generation, using this linguistic choice and the number and person information returned by the generation of "Shrimp", causes "isn't fond of" to be produced over time. This is the result of two more levels of recursive generation calls. "Isn't" is produced because of a global modifier in Bear to use contractions. (This modifier can be overruled by modifiers that are included as part of the generation process, if desirable.)

Next, the object concept is generated. Because it is equal to the person being spoken to, a behavior is chosen that realizes it as "you".

This short example leaves out many important details. Those details are included in the sections that follow.

Overview of How Hap Performs Parts of NLG Processing

The above example illustrates how the process of generation is performed in Hap, and suggests how an author can express linguistic knowledge in Hap goals and behaviors. Each of these is described in detail in the remainder of the chapter, along with how this approach addresses the needs of believable agents. Before describing those details, however, I want to summarize the ways in which Hap supports the main steps of the natural language generation process.

When an agent decides to communicate something, it issues a `generate` goal whose arguments contain a representation of the concept that is to be generated. This concept is a nested data structure with appropriate information. In addition, there may be modifying information provided as part of this goal or in other parts of the agent. Allowing uniform access to these various sources of information is one of the problems that the later sections address. Such uniform access seems important to me both to aid in the expression of linguistic knowledge by an author, and to aid in integration of natural language generation with the other parts of an agent.

The knowledge about how to hierarchically decompose and order the subparts of the concept to be generated is encoded in normal Hap behaviors for `generate` goals. These behaviors are normally sequential behaviors, with the order of the subgoals encoding the order of generation. The preconditions of these behaviors encode the conditions under which each ordering and decomposition is appropriate. This hierarchical decomposition eventually results in strings output over time by the execution of **Say** actions.

Communication between generation subgoals is done by Hap's normal mechanisms of return values and parameter passing. In addition, Hap allows generation subgoals to communicate by writing and reading from shared dynamic variables in a common parent. This alternate mechanism has some advantages that are described in Section 8.3.2.

Inference where needed can be expressed through the use of normal Hap behaviors and goals. Such behaviors can be used to infer an appropriate linguistic construct from a nonlinguistic concept to be generated, for example choosing `<want>` or `<would like>` lexical items to express a `desire` concept. They can also be used to infer other information needed in the generation process, for example determining the `voice` to express a sentence when `voice` is not provided in the concept to be generated. Goals for these behaviors are included in the appropriate `generation` behaviors where the information is needed.

Each of these issues, other details necessary to support generation, and the relevance of the approach to the needs of believability are described in the following sections.

8.3.2 Details of Direct Expression of NLG in Hap

As I mentioned previously, this approach is based on the Glinda generator [Kantrowitz 1990; Kantrowitz and Bates 1992]. The general approach I take is to use Hap to directly perform natural language generation processing similar to that performed by Glinda, and modify that processing to meet the requirements of believability. In the process, Hap was changed to support needs of language, and the generation process was changed to integrate well with Hap and support the needs of generation.

In the rest of this section, I first describe the extensions to Hap for nested data structures, of the sort used by Glinda, for the expression of concepts in the generation process. I then describe how the processing knowledge for natural language generation is expressed in Hap.

```
      group: (receiver (type 'relation)
                       (time 'present)
                       (agent (type 'agent) (value 'bear))
                       (predicate (type 'action) (value 'play)))
   features: ((case 'objective) (hearer 'bear))
```

FIGURE 8.3: Example Concept to Generate

Data Structure Support

As mentioned in the overview in Section 8.3.1, my generation approach uses nested data structures to express concepts at various stages of the generation process. Originally Hap provided no special support for nested data structures; Hap goals are a flat list of a name and zero or more values. This does not prevent Hap from supporting generation, because a goal's values can be any standard C data structure. So, in fact, one could use them without modification to support structured concepts. This would require explicit accessing functions and other support to be spread throughout the grammar, and would make encoding it tedious. It would also serve to separate generation from the rest of action because the ways of representing and accessing the arguments and other knowledge in generation would be different than those used by other parts of a Hap agent.

It is my goal to support a tighter integration, because I think this is needed for believability. I want to adapt Hap to support the nested structures of generation in a manner that is most compatible with its existing ways of expressing behavior knowledge, while still supporting the needs of generation. The first step for such an integration is to understand how the data structures of generation are used. For this purpose, I analyzed Glinda's use of its data structures.

A concept to be generated in Glinda is expressed as a structured *group* and set of *features*. Groups and features are expressively the same as frames or association lists. A feature is a name/value pair, and a group is a *role* followed by an unordered collection of groups or features. An example group and set of features is shown in Figure 8.3. This concept represents a play relation with the agent Bear as the actor. Its role is the receiver of the action in an enclosing relation. When generated, it produces the dependent clause "that you play" or "to play", depending on the verb of the enclosing relation. Pronominalization or elision of the agent occurs in this case because the agent of the relation is the same as the one being spoken to (as indicated by the `hearer` feature).

When looking at how Glinda uses these data structures for generation, there seem to be four key properties of the representation:

1. Groups and features need to be easy to create, and must allow easy access to component subgroups and features. In this NLG model, nearly all accesses to the group are to its immediate subparts, but see item 3 below.

2. The group and features set are independently specified when invoking a generation goal, but features in the feature set are treated as if they are included in the group. Any references to features must look in both the group and features set. This property allows the subgoaling process of generation to choose to generate a subcomponent of the current group and pass separately any modifying information through features. With the features in both the group and features set treated identically, modifying features can be embedded in the group or subgroups if they are important to the meaning and known ahead of time, or they can be included easily in the feature set that is created in the process of generation.

3. The only deeply nested portion of a group that is accessed is its *projector*. Conceptually, the projector is the core concept of the group, and along with the type and role of the group, it is central in choosing how the group is generated. It is found by a recursive traversal of the group, choosing the head of the current group at each level. The head is the most central subgroup or feature at each level. For example, the head of a group of type relation is the subgroup with role "predicate", and the head of a group of type word is the "root" component, that is, the subgroup with role "root". When the recursive traversal results in a feature, the value is returned as the projector of the group.

4. The final property of the representation is a global abstraction hierarchy for matching symbols within a group. It is often useful to write general rules that apply to all groups with a projector that is a verb, for example, with more specialized rules applying to groups with a projector that is an auxiliary verb or even more specialized rules applying to specific verbs. With an abstraction hierarchy applying to all of these matches, these general and more specific rules can be easily written.

In Hap, I supported these properties by adding first-class environments to Hap to support easy creation of groups, features and sets of features as well as the uniform access to elements of a group and set of features (properties (1) and (2)). An environment is a set of name/value bindings. As first-class entities, environments can be passed and stored as values. When an environment is passed as an actual parameter, it can be bound to a behavior's corresponding formal parameter and accessed in the usual ways. However, an environment also can be *imported* (by using the particular formals named `group` and `features`). If imported, all of the bindings in the passed environment are added to the behavior's lexical environment and can be accessed like normal bindings.

In Hap, the values of parameters and variables are accessed by the form $$<variable>. I extend this to allow a variable's *binding*, to be accessed by the form $$&<variable>. A binding can also be constructed using the form `(name value)`, where `value` is a value denoting expression. An environment is zero or more bindings. An environment value can be created by enclosing a sequence of bindings in parentheses. For example, the expression `((a 1) (b 2) (c ((d 3) $$&wolf (e 4))))` creates an environment with three bindings. The variable `a` has value `1`; the variable `b` has value `2` and the variable `c` has an environment as its value. This environment has three bindings: the variable `d` has value

3; the variable e has value 4; and the variable wolf has whatever value that variable has in the current scope.

A feature is now represented as a binding, and roughly speaking, both the group and set of features that comprise a Glinda goal are represented as environments. When these two environments are passed as values to a generate behavior, they are both *imported* so that all of their features and subgroups can be accessed by name or role as normal Hap variables. More precisely, in order to be accessed by role, a group is represented as a binding whose value is an environment. The name of the binding is the role of the group and the environment contains all of the other information. When a group is imported, the bindings inside the environment are imported. In addition, a new binding is created and imported with name role and value the role (that is, the binding name) of the group. If the above example group were passed into a behavior for a formal named features, then the whole environment could be referenced by the form $$features, and the components of the environment could also be referenced as normal variables: $$a, $$b and $$c.

Groups and features can now merge seamlessly with Hap goals and parameter passing. For instance, the following behavior shows how a natural language generate goal can be created:

```
(sequential_behavior request_play (who object)
  (subgoal generate
          (sentence ((type sentence) (hearer $$who)
                     (relation ((agent $$who) (predicate desire)
                                (object ((agent $$who)
                                         (predicate play)
                                         $$&object))))))
          ((focus play))))
```

This behavior tries to accomplish a request_play goal by generating natural language. The group and features are constructed using the binding and environment creating operators and include embedded variable accesses. The value of the who formal is included in the group in three places; the binding of the object formal appears once. Each of these could be either an data structure in an internal format or a group or feature. In this case $$who is an internal data structure used by Shrimp to point to one of the other Woggles for its normal action-producing behaviors, and $$&object is a group that represents a particular object concept.

Thus properties (1) and (2) are supported using environments as values. I support properties (3) and (4) by extending the matching language that all Hap agents use for preconditions and reactive annotations. Property (4) is provided by introducing a primitive is_a operator to the match language that allows items to match any of their abstractions. (The abstraction hierarchy is written by the author.) Projectors of a group can be matched by using a projector operator. For example, combining these one would write (is_a (projector $$group) auxiliary) as part of a condition (in a precondition, success test or context condition) that applies to groups with an auxiliary verb as their core concept.

Hierarchical Decomposition

With these data structures supported, we can turn our attention to the second challenge: allowing the encoding of the four types of generation process knowledge without compromising Hap's properties.

As mentioned in Section 8.3.1, the core of my generation processing is hierarchical decomposition: given a structured concept to be generated, break it into subconcepts to be generated and issue each of those in turn as subgoals of a sequential behavior. This is the same basic approach Hap uses for action generation[4], and that many natural language generation systems, including Glinda, use.

Normal Hap sequential[5] behaviors are used to express this generation knowledge in Hap. Each behavior expresses a particular order, and the behavior's precondition expresses the situations in which this ordering is appropriate. In this way, multiple, competing behaviors with appropriate preconditions encode the linguistic knowledge to decompose and order concepts appropriately. Examples of this expression are given in Section 8.4.

These behaviors operate at multiple levels in the generation process. At the text planning level, they select which concepts to convey and what order to present them in. At the realization level they follow appropriate precedence relations, and at the word level they order morphemes.

Smoothing of Stream of Text

The eventual result of this expansion is a sequence of strings over time. Local modifications are often necessary between pairs of these strings, for example to introduce spaces between words and longer spaces after sentences, introduce no spaces between roots of words and their prefixes or suffixes, and perform character deletions, additions and substitution (such as "i" for "y" in "happiness").

Combination rules perform these functions in Glinda. A buffer is used to hold the most recent string generated, and all applicable combination rules fire whenever a new string is generated. The (possibly modified) buffered string is then printed, and the new (possibly modified) string is placed in the buffer.

I support this knowledge in Hap with a special `generate_string` behavior (shown in Figure 8.4) that is invoked only when a string is being generated. This behavior keeps the buffered string and new string in dynamically scoped local variables, creates a `combine` goal with annotations that force all applicable behaviors to run (these are Hap's `(persistent when_succeeds)` and `ignore_failure` annotations), and then prints the possibly changed old string and buffers the new string. Rules for these local modifications

[4]With other embellishments such as parallelism, situated variation, and reactive annotations.

[5]Other types of Hap behaviors can also be used when that is appropriate to the desired expression. An example of such a situation is given in Figure 8.8.

```
(sequential_production generate_string (string)
       (dynamic_locals (old $$buffer)
                       (new $$string)
                       (common_parent  find common parent from last
                                       generate_string goal and this one)
  (with ignore_failure
    (subgoal run_all_combination_rules))
  (subgoal output_string $$old)
  (mental_act  copy $$new to $$buffer and mark all parents))

(sequential_production run_all_combination_rules ()
  (with (persistent when_succeeds)
    (subgoal combine)))
```

FIGURE 8.4: Generate String support behavior to allow string combination information to be encoded.

```
(sequential_behavior combine ()
       (precondition  type of $$common_parent is not word and
                      $$old does not end with ‘‘, or a space and
                      $$new doesn’t begin with any of the characters ,’.!?)
  (mental_act  add a space at the end of $$old))

(sequential_behavior combine ()
       (precondition  $$common_parent is type  word and
                      $$old ends with a consonant followed by a ‘y’, and
                      $$new doesn’t begin with an ‘i’ or ‘a’)
  (mental_act  change last ‘y’ to ‘i’ in $$old))
```

FIGURE 8.5: Combine behavior to put spaces between words and substitute ‘i’ for ‘y’ when appropriate.

between strings can then be written as normal Hap behaviors that match the buffered string and new string variables using preconditions and modify them through side-effects as desired.

Two such behaviors are shown in Figure 8.5. The first adds spaces between words in the sequence of strings output by generation behaviors. It uses the information about what type the common parent is to prevent inserting spaces between parts of a word such as the root and suffix. It also doesn’t insert spaces after a beginning quotation or a space, or before a comma, ending quotation, or any end of sentence punctuation. The second behavior causes substitution of ‘i’ for ‘y’ when appropriate. It does this when the ‘y’ is preceded by a consonant and the suffix does not begin with an ‘i’ or ‘a’. This provides correct substitution for words such as “happiness” and “spied”, and no substitution for words such as “played” and “spying”.

Information Flow

A number of effects in generation require communication between generation subgoals, for example subject-verb agreement and the propagation of verb tenses. In Glinda this knowledge is encoded in a third category of rules called *flow rules*. When a generation goal is about to be pursued all rules of this type are evaluated to provide the necessary data to the generation goal. In my approach to text generation in Hap I chose not to manage information flow using rules for two reasons. First, Hap already has mechanisms for information flow using normal parameter passing and value returns. And second, using rules for information flow introduces potential responsiveness problems for the agent. If used excessively the time required to evaluate these rules might compromise responsiveness of Hap agents by introducing arbitrary delays between goal expansions.

For these reasons, I facilitate such information flow through two mechanisms. The first is normal parameter passing and value returns from subgoals. The second mechanism uses dynamically-scoped variables and side-effects. I should point out that dynamic variables are not needed to encode the information flow necessary for generation; this information flow can be expressed using Hap’s normal lexically-scoped parameter passing and value returns. Nevertheless, I have found that dynamic scoping can provide certain advantages when carefully used. One can create dynamic variables in a common parent and allow both communicating goals to write and read from these variables. This eliminates the need for intermediate goals to explicitly pass along values. Dynamic scoping can also be used to allow behaviors running in parallel to communicate through locally shared variables.

Both of these mechanisms have the advantage over rules for information flow in that minimal processing is necessary at runtime to execute them, and thus far they have been expressive enough to encode the information flow necessary for generation.

Inference

The final type of processing needed for generation is inference. Glinda allows these inferences to be encoded as a fourth type of rule. All rules of this type are evaluated after flow rules and before a generation goal is pursued. Inference in Hap is done using normal Hap goals and behaviors as described in Chapter 5. Appropriate inference goals can be included in generation behaviors wherever they are needed.

These behaviors perform normal special-purpose inferences such as determining `voice` from `focus` and the content of the `actor` and `receiver` parts of a relation, or choosing lexical items to express a concept.

```
(sequential_behavior generate (group features)
       (precondition $$type is a relation and
                     $$mood is declarative or imperative)
  (subgoal generate $$&subject ((case nominative) $$&hearer))
  (subgoal generate $$&predicate ((sv_order sv) $$&r_time $$&voice
                                 $$&modal $$&number $$&person))
  (subgoal generate $$&object ((case objective) $$&voice $$&hearer))

(sequential_behavior generate (group features)
       (precondition $$type is relation and $$mood is interrogative_yn)
  (subgoal get_number_person $$&subject ((case nominative) $$&hearer))
  (subgoal generate $$&predicate ((aux only) (sv_order vs)
                                 $$&r_time $$&voice $$&modal
                                 $$&number $$&person))
  (subgoal generate $$&subject ((case nominative) $$&hearer))
  (subgoal generate $$&predicate ((aux skip_first) (sv_order vs)
                                 $$&r_time $$&voice $$&modal
                                 $$&number $$&person))
  (subgoal generate $$&object ((case objective) $$&voice $$&hearer))
```

FIGURE 8.6: Portion of standard language generation expressed in Hap

8.4 Building a Grammar in Hap

With the extensions described, one can express a traditional grammar in Hap directly. Generation then automatically inherits properties from Hap I believe to be important for believable agents. In addition, this combined architecture allows the agent builder to express additional knowledge in the grammar such as: how sensing affects linguistic choices, how emotions influence generation, or how the production of language should react to changes in the world. Later sections illustrate and describe these implications of the integration in more detail, in this section I show how one expresses a grammar, both traditional and extended, in this framework.

8.4.1 Traditional Grammar

Figure 8.6 shows, in simplified form, a portion of the traditional language knowledge that is encoded in the grammar I have built in Hap. The two behaviors in the figure encode two methods for generating a `relation` group. The first behavior produces language such as “I want to play”, and the second produces language like “Would you like for Shrimp to play”. Additional behaviors encode the knowledge necessary to generate other types of clauses. For example there is a behavior in the grammar that produces the subordinate clauses “to play” and “for Shrimp to play” in both of the above sentences.

Each behavior has a precondition that describes when it is applicable. The first behavior is appropriate when the type of the group is `relation` and the mood feature is `declarative` or `imperative`. In this situation the behavior causes three subgoals to be issued in turn, one each to generate the subject, predicate and object. Features are passed into each of these subgoals to cause the correct generation. For the subject and object subgoals, an appropriate `case` feature is included. The `hearer` feature is included to allow pronominalization if the person being spoken to is referred to. A feature to specify the speaker for pronominalization is not needed because the identity of the agent is stored in a global variable, `self`.[6] The `voice` feature is passed in to the generation subgoals for both the object and predicate subgoals to allow the behaviors that generate those groups to handle passive voice correctly. The `voice` feature and `subject` and `object` subgroups are inferred at an earlier stage in the generation from the `actor`, `receiver` subgroups and the `focus` feature, if present.

In addition to `voice`, five other features are included in the subgoal that generates the predicate. `Number` and `person` are passed in for subject-verb agreement. They are returned by the subgoal that generates the `subject` subgroup.[7] The `sv_order` feature is included to allow the correct realization of the predicate when the verb follows the subject (value `sv`) or when that order is inverted (value `vs`). `R_time` and `modal` are features that are inferred from the time feature in the concept to be generated, and are part of the information needed to generate sentences with different tenses.

The second behavior is appropriate when a group of type `relation` is being generated as a yes/no question (`mood` is `interrogative_yn`). It does this generation by causing subgoals to generate the first auxiliary verb of the predicate subgroup, the subject subgroup, the rest of the predicate, and the object subgroup. The subject and object are generated exactly as above. The first subgoal determines the `number` and `person` of the subject to allow the predicate to be generated. In addition to the features already described for the generation of the predicate is the additional feature `(aux only)` or `(aux skip_first)`. These features allow the predicate to be generated normally, but only output the first auxiliary verb produced or output all of the verb after the first auxiliary verb produced. In the course of generating the predicate, linguistic decisions are made that need to be consistent between these two invocations. In particular, lexical items can be chosen to express a concept; for example, “would like” or “want” could be chosen for the concept “desire”. This coordination is accomplished by the behavior that makes the lexical choice. In the process of making that choice it returns a new predicate group that replaces the concept (e.g. “desire”) with the lexical choices that express that concept (e.g. “would like”). This predicate group effectively replaces the old predicate group, ensuring that the same linguistic choices are made in the second invocation. An example of this is given in Section 8.5.

[6]If it is desirable to override that (as in the case of mocking another agent, for example), a dynamic variable can temporarily change the value of the `self` variable for the goals that generate the mocking utterance, or a `self` feature can be passed in to the appropriate `generate` goals. Remember that all of these variables are accessed in the same way regardless of whether they are passed in as part of the group, the accompanying set of features, Hap’s normal parameter passing or are otherwise in the scope of the behavior.

[7]Annotations to bind the return values are included in the running code for these behaviors. They were left out of the figure for clarity.

```
(sequential_behavior generate (group features)
       (precondition $$type is relation and
                     $$mood is interrogative_yn and
                     feeling glum with intensity > 2 and
                     $$desired_answer is yes)
       (specificity 1)
       (context_condition not happy with intensity > 1 and
                          not angry with intensity > 1)
  (subgoal generate $$&subject ((case nominative) $$&hearer)))
  (subgoal generate $$&predicate ((negative t) (sv_order sv)
                                  $$&r_time $$&voice $$&modal
                                  $$&number $$&person)))
  (subgoal generate $$&object ((case objective) $$&voice $$&hearer))
  (subgoal generate $$&location ())
  (subgoal generate_string ",")
  (subgoal generate $$&predicate ((aux only) (sv_order vs)
                                  $$&r_time $$&voice $$&modal
                                  $$&number $$&person))
  (subgoal generate $$&subject ((case nominative) $$&hearer
                                (pronominalize t))))
```

FIGURE 8.7: Grammar behavior that includes emotion and reactivity.

8.4.2 Grammar with Non-Traditional Knowledge

The ability to encode traditional grammar is useful, and in fact allows the generation process to inherit some properties of Hap automatically, as is shown in the example of processing in Section 8.5 and discussed in Section 8.6.1. But, in addition to these properties, for believable agents it is important to be able to express aspects of the detailed personality that are not normally expressed in traditional grammars.

Figure 8.7 shows an example of a grammar behavior that includes emotion and reactivity. Like the second example above, this is a generation behavior for the expression of a relation group as a yes/no question. It generates questions of the form "You don't want to play, do you?" or "Bear isn't quitting, is he?".

This body of the behavior is similar in structure to the other two behaviors above. The first three subgoals are the same as in the declarative sentence behavior, except that a `(negative t)` feature is included in the subgoal that generates the predicate. This feature causes a negative version of the predicate to be produced. (Whether contractions are used is determined by a `use_contractions` feature. In the personality of Wolf, it defaults to `true` and only has the value `false` when Wolf is very angry.) The last three subgoals cause the second half of the sentence to be produced. A comma is output by the generate_string goal, and the last two parts are generated by generating the predicate subgroup with `(aux only)` and `(sv_order vs)` features and the subject subgroup with a `(pronominalize t)` feature.

For some linguistic constructs, such as the normal phrasing of a yes/no question, the conditions under which they arise are straightforward: the agent is expressing a concept for which this behavior applies. Other phrasings might be chosen for a reason. This reason might be politeness, social convention or, as is the case of this behavior, because of the emotional state of the agent and content being expressed. For this personality, it uses a negative phrasing of a yes/no question when it is particularly sad and wants a "yes" answer to the question. This is intended to capture self-deprecating questions such as "you wouldn't want to help, would you?" or "you don't like me, do you?". (This is not the only situation in which it uses a negative phrasing. Other situations are encoded in separate behaviors.) The conditions in which this behavior is chosen are in the precondition for the behavior. The type of group it applies to is the same as in the previous example: that it is a relation group with mood `interrogative_yn`. In addition the desired answer of the question must be yes (as encoded in another feature of the group), and the agent must be showing its sadness (with an **act_glum** behavioral feature of appropriate intensity). Emotional references are easily included in the grammar because of its expression in Hap.

The personality of which this grammar behavior is a part, Wolf, is one that doesn't normally like to show its sadness. It is much more comfortable showing anger or happiness. So, if while this behavior is running, the agent becomes angry or happy (due to any of the concurrent things it is continuously doing), it would not want to continue this sad language. It would rather interrupt this and start over with another behavior choice. This is encoded by adding a context condition to this behavior. This behavior is only pursued when the context condition is true: that the agent is not significantly happy and not significantly angry. If this condition every becomes false (in this case the agent becomes happy or angry), then the behavior is aborted. This causes any subordinate goals and behaviors to also be aborted, and another behavior can be chosen to generate the concept (perhaps in an angry or happy way). This causes language such as: "You wouldn't want to ... You want to play, don't you?" when Wolf becomes angry in the middle of a sad phrasing.

Figure 8.8 shows another example of how one builds into the grammar variation based on sensing, reactivity and mixing action and language. This behavior encodes one possible way to reference a location: by simultaneously gesturing and using a pronoun reference. Other methods to generate references to locations (when this behavior doesn't apply or is undesirable) are encoded in separate behaviors.[8] The precondition states the conditions under which it is applicable: the group being generated must be of type location, the agent being spoken to must be looking at the speaker, and the location must be visible. The last two of these conditions are sensor queries that actively sense the world at the time the precondition is evaluated.[9] Action and language are mixed in this behavior simply by

[8]This work doesn't include contributions in traditional grammar itself. The referring expressions I use are bounded by the simple world in which these agents exist. For example, there is only one hill, and each agent can be referred to by name.

[9]Primitive sensors can be embedded directly in match expressions, as in this precondition. More complex sensing is performed by sensing behaviors written in Hap which can be included as subgoals of the behavior as described in Chapter 5.

```
(concurrent_behavior generate (group features)
       (precondition $$type is a location and
                     $$hearer is looking at me and
                     $$location is visible)
       (context_condition $$hearer is looking at me until
                          subgoal 2 is done)
  (subgoal generate $$&location
                    ((pronominalize t)))
  (subgoal glance $$location))
```

FIGURE 8.8: Grammar behavior that includes sensing, reactivity and mixing action and language.

including both goals as subgoals of the behavior. These goals are pursued concurrently because the behavior type is concurrent rather than sequential. To react to changes that might occur while this behavior is executing, appropriate annotations can be added. In this case it is important that the hearer doesn't look away before the gesture is performed. This is captured by adding a context condition with this information to the behavior. This condition causes sensing to be performed while this behavior is active. If the condition becomes false, this behavior is aborted, and another (perhaps referring to the location by name) will be chosen.

8.5 Example of Processing

To illustrate how my approach responds to the challenges for believability posed in the introduction to this chapter, let us now consider a detailed example of an interaction with an agent that includes generation.

As the example begins, Shrimp has approached Bear who is looking away. Shrimp is sad, but wants to play a game with Bear at a nearby hill. He decides to invite Bear to play, by invoking a `generate` goal for the group and accompanying feature set:

```
(sentence (type sentence) (hearer $$follower)
          (relation (type relation)
                    (agent (type agent) (value $$follower))
                    (predicate (type action) (value desire))
                    (receiver (type relation)
                              (agent (type agent) (value $$follower))
                              (location $$where)
                              (predicate (type action) (value play)))))
((mood interrogative_yn) (desired_ans yes))
```

The variables `$$where` and `$$follower` reference internal data structures for Shrimp. `$$where` references one of Shrimp's internal pointers for his concept of a location in the world, in this case the hill in the upper left hand corner of the world as shown in Figure 3.1. `$$follower` is one of Shrimp's internal references to one of the agents in the world, in this case Bear. Both of these data structures are the same ones used in Shrimp's other behaviors.

This group represents a relation concept with Bear as the agent. The predicate of the relationship is the action `desire`. The receiver is a relation group with agent Bear, predicate the action `play`, and a location with a reference to the hill as its value. The group also includes the information of who is being spoken to `(hearer Bear)`, that it should be expressed as a yes/no question, `(mood interrogative_yn)`, and that the desired answer is "yes", `(desired_ans yes)`.

As I will explain, this goal generates "Bear, you wouldn't want [pause] uh [pause] to play over there, would you?", while causing parallel action, sensing, etc.

The `generate` goal invokes a sequential behavior to perform the following subgoals:

```
(generate (punctuation beginning_of_sentence))
(generate $$&hearer)
(generate $$&relation)
(generate (punctuation end_of_sentence))
```

The first subgoal, when expanded, places a special symbol in the output buffer to mark the beginning of the sentence. This symbol cannot occur elsewhere in generation, and aids the capitalization and spacing combination behaviors.

The generation of the hearer feature has a number of behaviors from which to choose. If the generation of the example sentence is part of a larger on-going conversation between Bear and Shrimp, a behavior would fire that would result in the empty string being generated for the hearer feature. Since that is not the case, a behavior is chosen to decide how best to get Bear's attention. This is accomplished by sensing the world, for instance by making the first subgoal of this behavior be a sensing behavior to determine where Bear is and where he is looking. Since he is nearby but not looking at Shrimp, the behavior chooses to generate from the group `(name (object $$hearer))` followed by `(generate (punctuation end_of_pref))`, and issue the action to look at Bear in parallel. The first results in "Bear" being generated. When this happens the `combine` goal is posted with the buffer contents (beginning of sentence symbol) and the newly generated string "Bear". The `combine` goal persists until no behaviors apply for it. In this case there is only one behavior that applies. It removes the beginning of sentence symbol and capitalizes the first character of the newly generated string, which has no effect in this case because "Bear" is already capitalized. The `(punctuation end_of_pref)` group results in a comma being generated. No combination behaviors fire for these two, so at this point "Bear" is printed, "," is in the buffer, and Shrimp is looking at Bear.

If Bear had been across the room, this generation behavior would have resulted in Shrimp looking at Bear and generating "Hey Bear!". Alternatively, if Bear had noticed Shrimp's approach and was watching Shrimp attentively, a behavior would have been chosen to generate the empty string. Thus, sensing is being used to affect generation on a fine time scale.

The next goal to be executed is `(generate $$&relation)`. The behavior that is chosen first infers a `voice` feature and `subject` and `object` groups from the information provided in the group. In this case active voice is chosen, resulting in the content of the `actor` subgroup being placed as the content of the `subject` group, and the content of the `receiver` subgroup being placed as the content of the `object` group. The behavior then issues a new `generate` goal with this newly augmented group as its argument.

Now that the `subject` and `object` parts are known, a behavior is chosen to give an ordering to the parts of the group. There are several ways to order the subgroups of a relation to be generated as a sentence. Some of these are shown in Figures 8.6 and 8.7. Because Shrimp is currently feeling very sad, is generating a yes/no question, and desires a yes answer to his question, (the `(desired_ans yes)` feature), a negative phrasing of the question is chosen. This is the behavior shown in Figure 8.7. This results in the following sequence of subgroups:

```
(subgoal generate $$&subject ((case nominative) $$&hearer))
(subgoal generate $$&predicate ((negative t) (sv_order sv)
                                $$&r_time $$&voice $$&modal
                                $$&number $$&person)))
(subgoal generate $$&object ((case objective) $$&voice $$&hearer))
(subgoal generate $$&location ())
(subgoal generate_string ",")
(subgoal generate $$&predicate ((aux only) (sv_order vs)
                                $$&r_time $$&voice $$&modal
                                $$&number $$&person))
(subgoal generate $$&subject ((case nominative) $$&hearer
                              (pronominalize t)))
```

Because the `actor` is also the `hearer`, the first subgoal is generated as "you". The behavior that does this generation also returns two features: `(number singular)` and `(person second)`. (The annotation for binding return values are not shown. Each of these subgoals includes a `bind_to` annotation that expects an environment and *imports* all of the bindings of that environment for later reference.)

The second subgoal uses the `number` and `person` information in the generation of the `predicate` subgroup. This group is of type `action`, and so the behavior that is chosen for it has three steps: choose a linguistic expression of this concept; recursively generate using the chosen linguistic expression; and return any features that result from the recursive generation along with the group from the linguistic expression of the action concept. Because the `action` has value `desire`, the first step chooses among "want", "would like" and "would want". In this case it chooses "would want" by returning the group:

```
(predicate (type parameter) (modal conditional)
           (arg (type arg)
                (word (type word) (root want))))
```

This group has role `predicate`, so when it is returned it effectively replaces the previous `predicate` group with the choice made, by creating a more local binding with the same name. This causes this linguistic choice to persist to later steps of the generation process. In particular, in this example the sixth subgoal that generates the predicate again will use this group instead of the previous group with the `action desire`. This prevents the generation of sentences such as "you wouldn't like to play, do you?", which would result if "would like" were chosen when generating the first predicate, and "want" were chosen when generating the second predicate.

The recursive generation uses this group to incrementally produce “wouldn’t want” as its output.

At this point, Shrimp notices that the aggressive creature, Wolf, is coming toward him at high speed. He notices it via the firing of a higher level sensing goal. This knowledge gives Shrimp a good bit to think about, and the resulting processing elsewhere slows the Hap thread that is running this generation task. He becomes afraid. He actively looks to decide if he should run or get out of the way. Observers can notice that something is going on because Shrimp stops generating words. In addition, part of the behavior to communicate is a concurrently executing goal that watches for pauses in output and inserts stuttering “uh”s at intervals during these pauses. This goal is only active when a communication goal is active. As Shrimp’s pause becomes longer, this goal is triggered, and Shrimp says “uh”. Shrimp continues to watch Wolf, and decides to move slightly to let him pass more easily. As Wolf goes by, Shrimp continues to generate, producing “to play”.

Shrimp now generates the relation’s `location` subgroup. There are several potential behaviors. Since Bear is looking at him, a behavior is chosen to gesture and concurrently generate a pronoun referring to the location. So, Shrimp says “over there” as he glances toward the mountain.

Finally, the trace ends as the last three subgoals generate “, would you?”.

To summarize the behavior that is observed in this example, Shrimp has just come over to Bear who has not noticed him. Shrimp starts looking at Bear at the same time he says “Bear, ”. He goes on to say “you wouldn’t want” one word at a time, when he pauses and looks over at Wolf racing toward him. He looks around, says “uh”, moves slightly to get out of Wolf’s way and continues to say “to play a game”. As he says the next two words “over there” he glances toward the mountain. Looking at Bear again, he concludes with “, would you?”.

8.6 Properties and Opportunities of Expression for Natural Language Generation in Hap

With these extensions to Hap one can directly express a traditional grammar in Hap’s normal language constructs. Such a system generates text, just as the same grammar expressed in a traditional hierarchical natural language generator would. By being expressed in Hap, the natural language generation system automatically inherits properties of Hap that are central to its support of believability.

Here I discuss how this approach addresses the requirements for believable language use raised in Section 8.1. First, I describe properties that arise automatically (or nearly so) as a result of the architecture. Then, I describe opportunities of expression that this approach to natural language generation provides. For example, if one wants the agent’s language use to be reactive to changes in the world, an author must specify what changes this personality cares about and reacts to. The ability to specify such information is one of the contributions of my approach to believable agents.

8.6.1 (Nearly) Automatic Properties

By directly expressing natural language generation in Hap, it inherits four properties automatically or with limited work on the part of the author.

Incremental Generation

Incremental generation is a property of Glinda that I have preserved in my approach. Language is generated as a stream of morphemes. For each morpheme produced, decisions are made at various levels by conditions in the grammar behaviors. For this system to respond to the flow of changes in the world or in the internal state of the agent as it generates, the decision-making conditions in the grammar need to be sensitive to the appropriate changes. The ability to include such knowledge in the grammar is described in the other sections, and is one of the contributions of this integration. Including external sensing and internal sensing of emotions in the grammar is illustrated in Section 8.4.2, and is discussed in Section 8.6.2.

Pauses, Restarts, Other Breakdowns Visible

Pauses, restarts and other breakdowns due to the difficulty of the generation task itself are visible in Glinda and in this integrated system. However, with the generation process expressed as Hap goals and behaviors in an agent with other goals and behaviors, pauses or other breakdowns due to other aspects of the agent can also arise and be visible. These include pauses caused by the agent attending to goals activated by events in the external world (e.g. Wolf's approach in the example), as well as goals triggered by internal events. For example, generation could infer a piece of information, which when placed in memory awakens an otherwise independent goal. This goal might then perform more inference, generate emotions, generate action or language, etc. This might be similar to the types of pauses people make when realizing something while talking. The pause caused by this thinking would be visible in the timing of text output. Any actions or language produced by this digression would cause the mental digression to be even more visible.

Concurrent Pursuit of Goals and Parallel Action

Pursuing other independent goals (possibly resulting in parallel action) at the same time as the agent generates language happens without any additional work on the part of the author. All of the top-level goals of an agent are part of a collection behavior, so they are available to be pursued concurrently. When Hap pursues a generation goal, other goals in parallel threads (such as those from other top-level goals) are mixed in as time allows. Time is available to mix in other parallel goals when the body is executing a **Say** action, or when the generation is paused for other reasons (through the use of a `wait` step or conflict with another higher-priority goal or action, for example).

While some concurrent goal pursuit and parallel action occurs without extra work by the author, an author gets more out of the system if he takes advantage of the avenues of expression available to him.

Parallelism arises due to the initial structure of the ABT (all top-level goals are part of a collection behavior), but it can also arise because of the behaviors an author writes. At any level in the language producing behaviors being written, an author can write a concurrent or collection behavior as a way to satisfy a goal. When such a behavior is chosen, it can lead to the agent doing multiple things at once. This shows up in the example when Shrimp gestures at the mountain while generating language, and again when "uh" is said by Shrimp during the pause of his language generation. The first of these is the result of a concurrent behavior with two steps: one that results in saying "over there" and the other that results in the gesture. The second is a result of a concurrent behavior that has the main line of generation as one of its steps, and a demon that recognizes pauses as its second step. Parallelism of both of these forms are powerful for language and other behaviors.

An author needs control not just of when to introduce parallelism in the agent's behavior, but also of how to limit this parallelism. For example, if one was writing a behavior to forcefully threaten another agent in a style like Clint Eastwood's Dirty Harry character, it would destroy the forcefulness and mood of the behavior if Hap decided to automatically mix in a glance to his partner during one of the stand-off type pauses. It might even be undesirable for the character to blink during one of these pauses (especially if the personality is a child imitating Dirty Harry). This type of control is one of the needs that Hap's conflict mechanism was created to address. By adding appropriate conflicts to the character, one can control which behaviors or actions are never pursued at the same time. In this case, the author might add the conflict pairs `(clint-eastwood-threat blink)` and `(clint-eastwood-threat glance)`. In addition, by specifying the relative priorities of goals, an author controls the architecture's allotment of limited computational resources and indirectly determines which concurrent goals are pursued in parallel.

Not Reducing Responsiveness

With the generation behavior expressed in Hap, other parts of the agent can respond to emergencies or other higher-priority events regardless of how much processing language needs. As described in more detail in Chapter 5, this responsiveness would be compromised if generation is not built following the policy of building large computations in Hap rather than expressing them in their own mental actions.

For the language generation itself, responsiveness is a matter of how one builds the grammar and cannot be guaranteed by the architecture. Hap, and the approach to language described above, allow one to write grammars that have short paths from the `generate` goal to the issuing of the first **Say** that puts text in the voice bubble. The grammar that I built in this framework, while small, is very fast. It uses only a portion of the agent's processing time each frame to produce its text in a system running at 10 frames per second. But there is nothing to prevent an author from building arbitrary computation in the grammar. Hap does nothing to prevent complex inference, or even an infinite loop, from introducing delays in the completion of a behavior. The agent builder must be aware of such delays when building the behaviors and only introduce them when the pauses they introduce are appropriate for the personality being built, as they might be in a reflective, thoughtful personality. Higher-priority behaviors can interrupt such a behavior, allowing the agent to respond appropriately to other more important concerns.

Resource Bounded

Hap automatically enforces resource bounds both computationally and physically. The computational resource limitations are the result of Hap's single computational bottleneck: Hap expands a single goal at a time, and always chooses greedily among the available goals. This results in the highest priority threads getting full attention and other parallel threads getting mixed in as time allows. This arose in the example when Shrimp becomes afraid and moves out of Wolf's way. The computation time taken by this processing caused the generate behaviors to get less computation, and Shrimp's language production was slowed.

Hap also enforces physical resource limitations, by not issuing actions that use the same resources or that have been marked as conflicting (see Section 4.10). The higher priority (or first chosen if equal priorities) gets to execute, while the other is temporarily suspended.

8.6.2 Opportunities for Expression

In addition to those automatic properties, because generation is expressed directly in Hap, there are opportunities for an author to express language variation and other knowledge that are not normally included in natural language generation systems. These include the ways emotion, social relationships and personality affect the generation process, the way the current situation affects the generation process, the ways action are mixed with language for any purpose, and the types of changes this personality reacts to in its language production. Each of these is described in turn.

Varying for Emotional State, Social Relationships, and Personality

Because language generation is expressed in Hap, in addition to the concept that is being generated, the generation system has access, at all decision-making points, to the current emotions, behavioral features and social relationships of the agent. Variation based on this information is easy to include by simply including additional references to these features in the preconditions, success tests or context conditions of the grammar.

Thus, when writing a grammar for a particular believable agent, just as when writing other behaviors, the range of emotions and social relationships that might be present should be kept in mind. This variation can occur at all stages of the generation process. At the top level, the aspects of the concept to be expressed or suppressed might be affected. Decisions about what sentence level structure to use, or even what word choice to use might depend on this emotional and social state.

All of these decisions have to be made appropriate to the particular personality being constructed. Some characters vary their language widely based on their emotions. Others only allow subtle variation to appear between different emotional moods. With a firm personality in mind, and some visualization of different emotional situations this agent might find itself in, I believe one can decide on the the choice of emotional effects that should be spread throughout the grammar.

This is not necessarily as onerous a task as it might at first appear. There is normally many more than a single way to express any given concept. Using the personality, emotions and social relationships of the agent to choose among them can be natural and more satisfying than arbitrarily choosing a default way of talking.

Of course, the question remains of how to build a large, robust grammar that includes variation based on emotions, social relationships and that is tailored to express a particular personality. No one currently knows how to build such a grammar, I believe.

On the other hand, it is also not clear how large and robust a grammar is needed to be able to build a specific believable agent that will exist within a particular bounded context. It may be possible to make progress toward building believable agents with modest progress on the language issues.

Including Sensing

By directly expressing the grammar in Hap, all of the decision points also can use external sensing, just as other behaviors routinely include such sensing. In the small grammar I have created it has thus far seemed natural to include such sensing.

As above, no one has yet built a large, robust situated grammar. Building a robust grammar that includes and uses sensing well is an important research direction. For believable agents, progress may be possible with smaller grammars.

Including Action

Similarly, because generation is expressed as Hap behaviors it is easy to include action-generating goals as part of those behaviors. In fact, Hap doesn't have a distinction between "action-producing" and "language-producing" behaviors; many of the behaviors written in the "grammar" and other behaviors for Shrimp blur that distinction. It is one of the points of this work to make crossing that boundary easy.

As in the above two sections, because this is preliminary work, I do not yet know if building behaviors for a complete agent that substantially mixes action and language is possible. One of the most obvious questions is how large the cognitive load on the author is when building such an agent, and whether an author can handle that load when creating an agent. In the work I have done, this mixing seemed possible, but the final test is to construct a full agent that uses this approach.

Reactive Language

Each behavior in a grammar takes time to execute. A behavior to generate a sentence might take several seconds, and a behavior to generate the root of a particular word might take a fraction of a second. During this time the world and the agent's own internal state are continually changing. If they change in ways that the personality being built should react to, then this knowledge needs to be captured when building the agent.

Many of these types of reactions are captured in other parts of the agent. Events might cause parallel sensing behaviors, such as those described in Section 5.2, to trigger. Parallel goals of higher priority might take control and cause the agent to perform other action or thinking.

Some changes that can occur while a given behavior or goal is executing should affect the language generation behavior itself. Because the grammar is expressed in Hap, these situations can be locally recognized and reacted to. This type of reactivity is easy to add using success tests and context conditions. Examples of this are given in Figures 8.8 and 8.7, and are described in the surrounding text.

8.7 Summary

This chapter has presented an approach to natural language for believable agents. This includes suggestions for the requirements for believable language production, technology to allow natural language generation to be directly expressed in Hap, an example of an agent generating language as part of its activity, and an analysis of the properties and opportunities of expression this approach brings to natural language generation.

This work is preliminary. Only a small grammar and portions of agents have been built. Whether this approach can work for a larger grammar and agent is a topic of future work.

Chapter 9

Evidence of Success

In the preceding chapters I have presented my work toward believable agents, and I have attempted to present an understanding of the technical details used to construct believable agents. The question remains, however, of how well does it work? This chapter addresses this question by first presenting an analysis of progress towards the goals, and then presenting empirical evidence of success.

9.1 Analysis of Progress Toward Goals

At the beginning of Chapter 4 I described the three goals that guided the design of Hap: that it be a language for expressing detailed, interactive, personality-rich behavior; that it directly address many of the requirements for believability described in Chapter 2; and that it be a unified architecture in which to express all of the processing aspects of the agent. Now that Hap and its various uses have been described in detail, I want to revisit these goals to see how well they have been achieved.

9.1.1 Language for Expressing Personality

The first goal, that Hap be a language to express detailed, interactive, personality-rich behavior, has guided much of the design of Hap. How Hap was influenced by this goal, and how it can be used for this expression is the topic of much of Chapter 4 and later chapters. See Sections 4.3.1, 4.4.1, 6.3.2, the end of Section 4.5 and Section 7.7 for discussions that directly address this expression.

Although these sections discuss the expression of personality in Hap, the issue is spread throughout the dissertation because it is one of the central points of the thesis. It has impacted nearly everything from the initial decision that goals get their meaning from author-written behaviors to the details of how a particular agent expresses its fear in all of its behaviors, and I have tried to convey that impact and opportunities for expression in the previous chapters.

9.1.2 Unified Architecture

Chapters 5, 6 and 8 describe how all of the aspects of an agent's mind are implemented directly in Hap. Chapter 5 describes how arbitrary computation and composite sensing are expressed in Hap. Chapter 6 describes how the basic mechanisms for creating and combining emotions as well as the personality specific expression of what a particular agent becomes emotional about and how it expresses those emotions are expressed in Hap. And Chapter 8 describes how natural language generation is directly expressed in Hap. Thus, I believe I have presented evidence that Hap can serve as the basis of a unified architecture.

9.1.3 Architectural Support for Requirements of Believability

The remaining goal of the three goals for Hap was that it provide architectural support for the requirements of believability presented in Chapter 2. Figure 9.1 gives a summary of the support for these requirements given by Hap. Each of them is describe in more detail here.

- **Personality**: Hap provides architectural support for the permeation of personality through all aspects of the agent by being a language for the detailed expression of personality-specific behavior. This is described in Section 9.1.1. By implementing all aspects of the agent's mind in Hap, this expression of personality is not limited to action-producing behaviors, but also can include all aspects of the agent's mind.

- **Emotion**: Hap provides architectural support for Emotion through its integration with Em. Emotions are generated when important goals (annotated by the author) succeed, fail or are expected to fail. Some of this process is automatic, and other aspects require information about the personality being created, for example what it cares about and who it blames when an important goal fails. The generated emotions are automatically summarized, decayed over time, and mapped to behavioral features by an author-specified personality-specific mapping. The resulting features, emotion summaries, and raw emotions are available to all aspects of the agent's behavior to express the emotions in the full range of its activity. (See Chapter 6 for more information.)

- **Self Motivation**: Hap provides some support for the appearance of self motivation in that it requires an author to specify top level goals for the agent's active behavior tree. The author is encouraged to create top-level goals that require no external stimuli to arise. The detailed description of an agent, Wolf, in Chapter 7 shows how one top-level motivation structure is constructed, with goals that arise from external stimuli, internal stimuli and without any such stimuli. The requirement of *self motivation* is that the agent *appears* to have self motivation. Such a top-level structure allows that appearance to be achieved, but the behaviors that are enabled by the structure must *display* the self motivation in a way that it is visible to people interacting with the agent. Doing this well depends on the personality being constructed, the world being constructed, and the goals of the author.

- **Personality**: yes — language for expressing detailed, personality-specific behaviors.
- **Emotion**: yes — integration with Em.
- **Self Motivation**: some — top-level goals, motivation structure of agents including top-level goals that require no external stimuli to arise.
- **Change**: no architectural support
- **Social relationships**: yes — integration with Em.
- **Consistency**: no architectural support
- **Illusion of Life**:
  - **Appearance of Goals**: yes — explicit goals.
  - **Concurrent pursuit of Goals**: yes — automatic, greedy, author managed by conflict, priorities and structure of behaviors.
  - **Parallel Action**: yes — same as above.
  - **Reactive and Responsive**: yes — reactive annotations, parallelism.
  - **Situated**: yes — behaviors chosen in the moment, reactive annotations.
  - **Appear Resource Bounded – body and mind**: yes — action resource conflicts, author-specified conflicts, and single processing bottleneck
  - **Exist in a Social context**: no architectural support
  - **Broadly Capable** yes — unified architecture.
  - **Well integrated (capabilities and behaviors)** some — unified architecture.

FIGURE 9.1: Architectural support for Requirements for Believability.

- **Change**: Hap provides no particular support for the requirement that agents grow and change. It is important that any such support not cause change that is wrong for a given personality.
- **Social relationships**: Hap is integrated with Em which provides explicit support for the social relationships of `like`, which captures the notions of like and dislike. In addition, Neal Reilly's work on Em has included advances in building social agents, and I believe Hap supports these advances.
- **Consistency**: Hap provides no architectural support for the consistency of personality requirement.
- **Illusion of Life**:
  - **Appearance of Goals**: Hap has explicit goals, that derive their meaning and activity from the behaviors written for them. These behaviors often show, through the activities they structure, that the goal is present.
  - **Concurrent pursuit of Goals**: Hap automatically pursues multiple goals concurrently. It does this in a greedy manner, pursuing as many goals as time allows given the flow of events in the real-time world. The pursuit of concurrent goals is managed by physical constraints of the domains — actions that have resource conflicts cannot be executed at the same time. It is also managed by author-specified information: conceptual conflicts between goals or between goals and actions, the structure of behaviors (where parallelism is allowed), and the priorities of goals.
  - **Parallel Action**: When Hap automatically pursues multiple goals concurrently, often resulting in parallel action execution. This concurrent execution of goals is subject to the same management as above (by action resource conflicts, author-specified conflicts, structure of behaviors and priorities).
  - **Reactive and Responsive**: Hap provides reactive annotations to any goal or behavior. This tight syntactic coupling of reactive annotations to the concepts they affect is meant to encourage and suggest reactive annotations in the authoring process. Hap also allows easy expression of demons at any level including the top level, and supports parallelism to take advantage of these demons. Such demons, when they are relatively high priority, can be used to respond to events that require quick response time. In addition, Hap is compiled and uses selective sensing to allow its processing to be responsive in practice.
  - **Situated**: All behaviors are interpreted with respect to the moment in which they are instantiated. Preconditions for behaviors can be used by an author to encode how a particular personality varies its behaviors based on the situation. Reactive annotations can recognize changes in the situation for which the personality should respond.
  - **Appear Resource Bounded – body and mind**: Hap enforces physical resource limits by enforcing action resource conflicts. Hap also has a single processing

bottleneck: it expands a single goal or executes a single action at a time. It manages this bottleneck in response to the flow of activity in the world by always attending to the highest priority goals first and mixing in other activity only as time allows. The author controls this greedy management of the mental resource bound by specifying the priorities for an agent's goals as appropriate for its personality.

- **Exist in a Social context**: Hap provides no architectural support for following the social conventions of a particular world.
- **Broadly Capable**: Hap provides support for the agents appearing broadly capable by being a unified architecture for all aspects of the agents mind. It includes support for action, sensing (both primitive and patterns over time), emotion, arbitrary computation and natural language generation.[1]
- **Well integrated (capabilities and behaviors)**: Hap provides support for the integration of **capabilities** by allowing the expression of all capabilities of the agent's mind in the same language of goals and behaviors. In this way activities of different capabilities can be mixed by including the goals for these activities in a single behavior. Multiple examples of this mixing are given in Chapters 7 and 8. For the integration of **behaviors**, Hap provides some support because of the authoring nature of behaviors and the reflection capabilities in Hap. Behaviors can use reflection or sensing to know what behaviors have recently executed to make appropriate author-specified transitions. The need for appropriate transitions between behaviors is the subject of Phoebe Sengers's Ph.D. thesis research [Sengers 1996].

9.2 Empirical Evidence

The above analysis suggests the progress that this work makes to addressing the needs of believability. The question remains, however, of does it really work? What *evidence* do I have for how well the approaches presented in this thesis work? In this section I present empirical evidence for the level of believable agent that has been built using this technology, and evidence that others besides myself can use this technology for constructing believable agents.

9.2.1 How Believable an Agent Can Be Made in Hap; What Evidence Do I Have?

Evaluating believability in agents, as in traditional non-interactive characters, is a fundamentally subjective pursuit. This is the process that has been used in the arts since they began. People go to a movie or play and say whether so-and-so was "good" in the role of Macbeth (usually along with how they liked the play). The most formal evaluation we have

[1]The natural language generation work is preliminary.

evolved for characters are critics who give their reviews. These reviews, although more informed, are still subjective opinions. To complicate this subjective process is the fact that no two people, especially critics, agree in these subjective judgements. Nevertheless, building believable agents and somehow getting these subjective judgements seems the only way to obtain evidence for how good a believable agent can be built.

One way of measuring such subjective judgements to judge how believable traditional characters are is to look at how many people choose to go to see the work. This metric doesn't work very well if the work doesn't focus on character. People pay to see the movies with good special-effects and strong adventure with less concern for how good the characters are. But the more the work focuses on character, the more the success of the work can be attributed to the success of the characters. Movies like *Terms of Endearment* (1983) or most of the movies with Meryl Streep or Sophia Loren simply are not successful unless the characters are good. In these movies the box office gives some indication of how good the characters are.

To see what evidence there is for the level of believable agent that can be built using my technology, we must look at the agents that have been built for evidence in one of these forms. A number of agents have been built in the Hap architecture. The ones that most fully use the features of the architecture and have been most widely seen and interacted with by normal people are the Woggles. The Woggles were originally shown at the AAAI-92 AI-based Arts Exhibition and then subsequently at SIGGRAPH-93, at Ars Electronica '93, and in the Boston Computer Museum. At the Boston Computer Museum they were exhibited for about a year.

The Woggles are a character-based world. They use the physical world and bodies described in Chapter 3 without the **Say** action. This world was designed to be simple with the complexity coming from the characters. There are no objects in the world beyond the characters, and the only mechanism is the "chute" which functions similar to a playground slide. There are three computer controlled agents in the world, including Wolf who is described in detail in Chapter 7, and the fourth body is controlled by a human user using the mouse. Thus, like character-based films, there is little in the world to keep the attention of people except the agents.

The Boston Computer Museum doesn't keep detailed statistics about their exhibits, but the Director of Exhibits and the Executive Director of the Museum had these observations:[2]

> It was there for about a year. (We don't have written records in inventory since the machine was a loaner.)
>
> The computer museum gets 150,000 people a year; probably about 30,000 people used it.
>
> People needed a little help to get started. And so we had signage up for them. But then they used it on their own, and it was a big hit.
>
> It was one of those exhibits that people either get or don't. Some people spent time and got into it, and others just saw bouncing balls and moved on.

[2]These quotations were given by personal communication.

> If they did get it, then they enjoyed it. They laughed, and usually played with it for a while.
>
> The ones who got it, usually played with it for 5 to 10 minutes.
>
> 10 minutes is a long time for a public exhibit. We have only a few exhibits that people interact with for that long. They sense other's presence, and want to keep moving.
>
> It was extremely popular. People were visibly laughing and having fun.
>
> The response was overwhelmingly positive because it was colorful, engaging and because of the endearing aspects of the creatures.

These observations include anecdotal support that suggests that the Woggles achieve some level of believability. The hard evidence they give is similar to the second form of evidence above: they are believable enough to engage people for 5 to 10 minutes.

9.2.2 Can Anyone but Me Build Them?

An approach and technology for building believable agents is of limited use if the creator is the only one that can use them. So, a natural question in evaluating this work is: can anyone but me use the technology to build believable agents?

One piece of evidence to address this question comes from the Woggles, the same system that was exhibited at the Boston Computer Museum as described in the previous section. The behaviors of the Woggles were built in six weeks by five people. Three of these authors did not have previous experience with the architecture. (The other two were myself and Scott Neal Reilly.) Over the course of building the Woggles, these three others learned the architecture and built the majority of the Woggles' behaviors. I built less than a fifth of the behaviors because I was also concentrating on building the Hap architecture.

Other evidence that this technology can be used by others are the other researchers who have chosen to use it for their own work in believable agents or related areas.

Scott Neal Reilly used Hap as the agent architecture for his doctoral thesis research on emotion and social knowledge for believable agents. This work is reported in his dissertation, *Believable Social and Emotional Agents* [Neal Reilly 1996]. In this work Neal Reilly developed a model of emotion and an approach for social knowledge for believable agents as well as building several agents.

Phoebe Sengers is currently using Hap to pursue her doctoral thesis research on transitions between behaviors for believable agents. She is working to build agents that combine many behaviors while remaining coherent and understandable as a whole [Sengers 1996].

Barbara Hayes-Roth at Stanford University has used Hap as part of her architecture for agents in the Virtual Theater Project [Hayes-Roth and van Gent 1997]. In this work Hayes-Roth is creating computer-based "improvisational actors" that can be directed by children to construct stories. The actors must be able to take direction and generate improvised activity that is appropriate to their personality.

Charles Rich and others at Mitsubishi Electric Research Laboratories have used Hap as part of agents they built for a real-time, interactive, animated world [Rich *et al.* 1994].

A version of my technology is also being used in a commercial product. Fujitsu, Ltd. is currently selling a product in Japan and the USA called Teo, that showcases a believable agent called Fin Fin. The system allows one to interact with Fin Fin, and build a relationship with him over time. Fin Fin uses Fujitsu's implementation of Hap for part of its control.

9.2.3 Evidence for NLG Portion

The natural language generation portion of the work is at an earlier stage than the other parts described in this dissertation. There is some evidence that the fundamental merging of NLG with the rest of the architecture works, and there are examples of the combined system responding to many of the challenges raised in the introduction to Chapter 8. But there is no evidence yet for being able to build complete agents using this system that are believable. There is also no evidence that others besides myself can use the system. These are both natural areas for future work.

Chapter 10

Issues and Guidelines for Building Believable Agents

The previous chapters have described the task of creating believable agents and technology that I have created to address that task. At this point I want to take a few pages to address issues in building believable agents that are not addressed by the previous chapters. This collection of issues and guidelines are eclectic, with the unifying theme that I think they are the most important and under-represented (in the rest of the dissertation) of a larger set of such issues for working in this area.

I will not be proving any of these statements. You may choose to not believe some (or all) of these. But these are beliefs that I have come to after eight years of working and thinking about building believable agents. In many cases, they represent reversals of deeply-held positions that I feel I have been forced to accept as I have struggled to understand and face the problem squarely.

10.1 **Study and Build the Unique:** Generality is Good Computer Science but Bad Believability

Computer Science, and Artificial Intelligence as a part of it, rewards, values and pursues *general* solutions to problems. This is appropriate for much of computer science because general solutions are possible, and are more useful than narrow solutions.

In believable agents, however, the nature of the problem is in conflict with generality. Believable agents are largely about the unique. No one goes to see Inspector Clouseau in the famous *Pink Panther* movies because he is a general solution to the "believable character" problem. They go to see him because of the individual personality and character that is brought to life on the screen.

To make progress toward building interactive versions of such powerful characters we need to embrace this aspect of the problem. As discussed earlier, this seems to require that the envisioned personality permeate the decisions made when developing all aspects of an agent, abstract or concrete, low-level or high-level. If that means that particular solutions have less generality than we would like, that is infinitely better progress than general solutions that don't address the problem.

10.2 **Embrace Integration:** The Hazards of Modularity

Similarly, modularity is one of the marks of good computer science. And like generality, this is for good reason. Modularity often allows for code re-use, more understandable systems, the ability to make independent progress on components of the system and plug them in when they are ready, and several other advantages.

But, like generality, modularity is in conflict with the needs of believable agents. Tight, seamless integration is needed for believable agents. When creating a personality-specific behavior to express a portion of a particular believable agent, an artist needs to be able to freely use and combine any of the agent's avenues of expression. Any hampering of this expression due to modularity boundaries will be destructive to the needed expression.

In addition, it is important for believability that people interacting with the agents not be able to see module boundaries. A viewer can't tell when Humphrey Bogart is in "language generation mode" versus "acting mode". He and all characters in the arts seamlessly blend these capabilities.

In many ways this observation goes beyond just modularity. It impacts various aspects of "neat" artificial intelligence and computer science. In my experience building believable agents and working with others to build them, I have found that when working to express a personality, authors always want to modify any part of the architecture that gets in the way of what they are trying to express. This frequently has meant breaking abstraction barriers, working around "defaults" intended to make life easier for the author, and changing aspects of the architecture that get in the way of their expression. This is why my work has gradually moved toward a *language* in which the artist can express a personality.

Artists are breakers of rules, including those imposed by modularity boundaries. This is for good reason; the goal is to express an envisioned unique, quirky, rule-breaking personality in interactive form, not to make reusable code. Anything that enables or facilitates flexible expression by the author is precisely what is needed to advance the field.

10.3 **Build Expressive Tools:** Primitive Tools Might Be Better Than High-Level Tools

When I describe my research, computer scientists often assume that what I am working toward is something like the believable agent building machine. The vision is normally a machine that allows one to set values for the personality you want, for example a value for the introvert/extrovert scale, another for the body size, etc. From this set of values (usually envisioned as a small set), the machine produces the believable agent that is defined by these properties.

Character-based artists never make this assumption. Their first concern tends to be whether they will have enough control to create the personality they envision. How easy the tools are to use is always secondary to this concern of being able to express their vision.

This difference, I think, reflects deep properties of the problem of believability that character-based artists understand by virtue of working relatively longer on the problem. They understand the underlying fallacy of the "agent building machine" described above: that is that for any set of values set on such a machine there is not just *one* character that fits those values, but rather an infinite set of characters that fit them.[1] And they want to choose and create a particular character. The details are everything.

This may be why the tools of artists in general tend to be relatively low-level; expressiveness is paramount. Painters put every drop of paint on the canvas with brushes even though machines that automatically paint between the lines could be built. Directors carefully stage and craft every shot for their films. Writers choose every word for a novel. This is because the details are crucially important in the work they are creating.

For artistic, high quality believable agents, the details are just as important. And so the tools must be just as expressive. Any attempts to create labor-saving tools (higher level languages, believable agent toolkits, etc.) that undermine this expressiveness are misguided.

10.4 How to Make Progress Without Modularity, Code Reuse and High-Level Tools

One of the reasons generality, modularity and high-level tools are so valued in computer science is that they make progress in the field possible. If the nature of the problem forces researchers in believable agents to give up these aids to progress, then how can progress be made in this new art and science? This is an important issue, and one which I believe researchers in this field will have to address. As a start, I would like to suggest three possible approaches.

10.4.1 Inspiration from Previous Art/Agents

One of the best ways to make progress in the arts (and I would argue in computer science) is by inspiration from previous work. When a painter sees a painting in a gallery that inspires him, he doesn't go home and try to duplicate it. Instead, the striking painting is in some way inspirational to him. It might suggest a new way of doing things, or simply a level of expression that the artist didn't know was possible, or perhaps something less easily described. In any case, this inspiration, rather than leading to direct use in the painter's work, causes various forms of indirect use depending on the nature of the inspiration. He

[1]How many values would be needed to uniquely determine only a single character is an interesting question. I wouldn't want to count them.

might think about his work in new ways, or try new techniques to get as powerful an effect, but the resulting work will still be his own. The change is one of inspiration.

Artists of all media do this all of the time. The Disney animators in the beginning of their studio thought this form of progress was so important that they made it almost a formal part of their work. Whenever Walt Disney or a director thought a piece of animation was particularly good, or had captured some important quality, he called all of the other animators over to see it. These animators then went back to their own work, which was usually quite different. The value they gained was not animation reuse, but learning through inspiration.

The same inspiration can take place in believable agents. Some might argue that this is a dominant path of progress in computer science research already.

10.4.2 Copy and Modify

Code reuse, in general, may be against the requirements of believable agents. It goes against the maxim that no two characters ever do something in the same way. But it is possible that a reasonable approach might be copying and *modifying* portions of existing agents. The modifying is crucial, without it there is little hope that the existing behavior would work in the new personality. The process must still be one of authoring.

Again, one might argue that this is already a part of computer science research culture. It is rare to incorporate someone else's work in your research without modification. In this case, because of the nature of believability, the modification is all the more crucial. The greater the modification, the closer this method approximates progress by inspiration above.

10.4.3 Agent Reuse

Piecemeal code reuse might not be compatible with believability, but a different view of the method might be fruitful. One could reuse whole agents rather than portions of agents. Given a developed framework for an agent and a particular believable agent built in that framework, an author could continue to develop and extend that agent for a series of worlds, interactions, etc. Over the course of this development the agent could be used in a series of interactive story worlds, interfaces, or other situations. This would, of course, involve new challenges and problems, but it might be a promising path for progress that is not in conflict with believability.

Many existing systems have benefited from such a long development life. The Soar architecture [Newell 1990; Rosenbloom *et al.* 1993] is a particularly rich example. By being actively pursued for well over a decade now, Soar researchers have a different perspective than other artificial intelligence researchers who have more freely changed platforms.

10.5 **Irrationality Is OK:** The Hazards of the Assumption of Rationality

Much work in artificial intelligence assumes that we want to build rational agents. This is probably a correct assumption if we want to build robots to assist us, or information agents to manage our mail. This is not an appropriate assumption if one is building believable agents. Many characters are irrational. Some might argue that those are the interesting ones. Authors need to be able to build believable agents that are irrational. If believable agent researchers adopt the assumption of rationality in their research, they run the risk of blocking the construction of believable agent versions of these rich characters. It is not clear to me how far irrationality reaches in believability. It is possible that no believable agent could be built in an architecture that enforced the rationality of its agents.

10.6 Study the Arts (Believable Characters)

The final issue I want to address for how to build believable agents is to study the character-based arts. The space of knowledge in this area is large and nearly completely uncharted by computer scientists. The nearest cousins to believable agents are believable characters. This domain is immense, but artists have been working to understand it for millennia. What they know may be foreign. It may undermine long held beliefs for computer scientists (as it has for me). But when they are right, researchers in believable agents ignore them at their own risk.

Chapter 11

Conclusion

In this chapter I conclude by first summarizing the results of the dissertation. I then compare and contrast to recent related work. Finally, I give some thoughts on promising future directions.

11.1 Summary of Results

The goal of the research reported in this dissertation is to make progress toward the creation of believable agents. In directly attacking this goal, I have addressed a number of areas.

11.1.1 A Study of Believability

The first result of this dissertation is a study of the problem to be solved. This study has two parts: a study of believability in traditional character-based arts, and a study of believability for agents, primarily drawing on experience in multiple attempts to construct such agents. Both are necessary. Studying existing art is necessary because artists have struggled to create believable characters for decades, or millennia depending on the art form, and much of what they know applies equally well to the creation of believable agents. Attempting to build believable agents and using that experience to study believability is also necessary because the nature of autonomous agents brings new challenges to believability that are not focused on by traditional artists.

The results of this study show up in multiple places within the thesis. First, they are the basis upon which the requirements from Chapter 2 and Chapter 8 are drawn. Second they are, in large part, the basis of the ideas presented on issues and guidelines for building believable agents in Chapter 10. And third, the results of this study are spread throughout the thesis as methodology, ways of thinking, examples, and incidental remarks.

David Chapman argues well in his thesis [Chapman 1990] that the most important result of many works in artificial intelligence are *approaches*, ways of thinking about a problem. I hope if one takes any one thing from this dissertation, it is an understanding of believability and ways of thinking about the problem.

11.1.2 An Architecture for Believable Agents

The second result of this thesis is an architecture for creating believable agents. This architecture includes Hap and the technical components built upon it. Hap was specifically designed to support the creation of believable agents in three ways. It is a formal language of expression for conveying detailed, personality-rich, situated behavior. Such a language is needed to think about the details of personality beyond the initial generalized descriptions of a character. It is not enough to know that the character is a villain who hates children. To describe a personality in the detail needed for building a believable agent, you must also know how the villain walks, what she does for fun, what she cares about, what she reacts to and when, what she does and doesn't do at the same time, etc. Hap is a language for expressing these details.

The second way in which Hap supports the creation of believable agents is as an architecture that provides support for many of the requirements of believability. This support is described in detail in Chapter 9.

Third, Hap is a unified architecture for the expression of all aspects of an agent's mind. By implementing all of these aspects in Hap they inherit Hap's architectural support for believability.

11.1.3 Steps Toward Natural Language Generation for Believable Agents

The third result of this thesis is an approach for natural language generation for believable agents. This result includes a description of the requirements for natural language in believable agents, an architecture for supporting these requirements, and an example grammar and behaviors that use this architecture as part of an agent's behavior. The resulting behaviors address some of the key requirements for language generation in believable agents.

This aspect of the work does not yet include a complete agent that uses language, or evidence that others can use it to build their own believable agents.

11.1.4 Agents

A number of agents have been built using this framework. I built the behaviors for a simulated cat called Lyotard which has been reported previously [Bates *et al.* 1994; 1992]. I helped build the Woggles, and I built behaviors that use language for part of the behaviors of a complete agent.

These constructed agents show how the technologies described in the dissertation can be used together to build complete believable agents. In addition to appearing as examples throughout the thesis, a complete description of one of these agents appears in Chapter 7.

11.1.5 Evidence of Success

I have presented evidence that this work makes progress toward building believable agents. This includes results of a year of the public's interaction with agents built in this framework when the Woggles were exhibited at the Boston Computer Museum. The empirical evidence shows that some level of believability can be achieved using the approach described in this thesis. It also shows that others can learn the framework and use it to build believable agents.

In addition, there have been a number of other researchers using this architecture for their own work. There are two Ph.D. theses focusing on aspects of believable agents besides my own that use this architecture. Researchers at Stanford and Mitsubishi Electric Research Labs have used the architecture in their work, and there is a commercial believable agent product based in part on this architecture.

11.2 Related Work

In Chapter 1, I discussed work up until about 1990 that influenced the work presented in this thesis. In this section I discuss more recent work that relates in some way. I focus on works that relate to believable agents as a whole first and include works that relate in more specialized ways toward the end of the section.

11.2.1 Virtual Theater

Hayes-Roth and her colleagues are building systems for improvisational theater [Hayes-Roth and van Gent 1997]. These systems include semi-autonomous agents that can be directed by children to build stories or improvisational experiences. Her agents use BB1 [Hayes-Roth 1995] to interpret the situation, provide choices to the children and cause the agents to perform the chosen behaviors. Her system uses Hap as described in this thesis to realize the behaviors. (Another version of Hayes-Roth's system uses the Improv system described below.) Her Hap agents have no motivational layer, and in fact no autonomy[1]; instead, they are controlled by a script provided by the children or by the children's direct control using the interactive interface the system provides. In addition to the behaviors provided by Hap and the combinations of those behaviors provided by BB1, the system also allows speech actions by being able to play one of 250 prerecorded phrases. This combination makes for a powerful "stage" in which children can direct and improvise, but since the agents are not autonomous, the component that is directly related to believable agents is the behaviors that locally carry out the children's direction.

[1]A previous version included limited autonomy by the agent actors, but this was found to confuse the children who were directing them.

11.2.2 Improv

Similar to Hayes-Roth's system, in that it emphasizes believable movement but is not intended as autonomous believable agents, is the work of Ken Perlin and Athomas Goldberg. Perlin has built an animation system based on ideas from his work in procedural texture synthesis [Perlin 1995]. This approach produces compelling movement in human figures; the movement is especially compelling in his dancer [Perlin 1995]. Originally this work created "puppets" that were controlled by a person choosing which behavior the puppet performed at any given time. More recently this work has been progressing toward autonomy. Together Perlin and Goldberg have extended this animation system to support scripted agents [Perlin and Goldberg 1996]. These agents are still not complete believable agents. The focus, instead, is on providing tools that allow artists to create powerful scripted scenes with these agents.

11.2.3 ALIVE

For his thesis work, Bruce Blumberg has created an autonomous agent architecture [Blumberg 1996] and a number of agents, most notably a simulated dog and hamster. Blumberg's architecture draws on ethology, the study of animal behavior, for much of its structure. Behaviors are created as independent, competing entities that continuously vie for control of the agent. He provides mechanisms to aid in the relevance of an agent's resulting behavior to its goals, and for limiting the amount of switching between the goals to give an appropriate amount of persistence to the agent's behavior. The system also includes an interesting framework for mixing primitive actions from multiple behaviors. In addition, the architecture includes an ethologically-motivated model of learning, and a promising approach to robust navigation and obstacle avoidance for virtual agents in its use of synthetic vision for these pursuits.

Blumberg's work is usually presented as part of the ALIVE system developed by Maes and colleagues [Maes *et al.* 1995]. ALIVE is an impressive combination of technologies that creates a visual "magic mirror" in which a user can step in front of a large monitor and see himself in a fantasy world on the screen. It uses computer vision and graphics technology to merge the video image of the person with the computer generated scene, as well as recognizing position and gestures made by the person to allow interaction with the computer-created world.

This approach seems a promising one for the creation of agents with realistic animal behavior, which has some commonalities with the goals of believability. However, as argued in this thesis, there remain significant additional requirements of believability not addressed by the ethological approach.

11.2.4 Herman

James Lester at North Carolina State University has built an architecture to increase believability in animated pedagogical agents [Lester and Stone 1997]. His system includes an animated bug, called Herman, that teaches kids about plants. Herman is controlled by a pedagogical sequencing engine [Stone and Lester 1996] that chooses appropriate speech and animation segments to perform its teaching role. During the pauses between these segments "believability-enhancing behaviors" are chosen and executed by a complementary part of the architecture [Lester and Stone 1997]. There are 15 believability-enhancing behaviors of durations between 1 and 3 seconds. They are activities such as scratching his head, twiddling his thumbs, sitting down, etc. The choice of which believability-enhancing behavior to perform is based on its appropriateness to the learning situation.

Lester's system is an interesting mixing of the needs of a pedagogical agent with approaches toward believability. Teaching is Lester's primary goal in this work, and to avoid damaging the ongoing learning experience, believability takes a secondary role. The personality of Herman primarily comes across in the canned animation sequences, with the autonomous portions adding some illusion of life, but not constituting a full believable agent. There is no notion of the agent having self-motivation, emotions, appearance of goals (beyond the goal to teach), or interactive personality. At a low level, the system executes one behavior (pedagogical or believability-enhancing) at a time, and often has moments in which the agent is motionless for multiple seconds. Yet, Lester seems to clearly understand and value the goal of believability, and this makes his work rare among the work to date.

11.2.5 Julia

Michael Mauldin has created an autonomous agent called Julia [Mauldin 1994; Foner 1997] that regularly interacts with people in multi-person text-interface worlds (MUDs), and has participated in the modified Turing Test of the Loebner prize competition. Julia is an agent that interacts primarily through text communication (although she also acts in MUDs). The task Mauldin faces with Julia is different from the task of creating a believable agent. First, in both the MUDs and in the Loebner prize competition Mauldin is often going up against sceptical interactors who are actively trying to find out if Julia is computer controlled or not, rather than the willing suspension of disbelief in believable agents. Second, in both of these domains interactions are primarily through text alone. The architecture of Julia is engineered for these differences. She uses a simple parser, primarily prewritten text responses, a simple discourse model and an elaborate arbitration mechanism to match the parse (or lack of a parse) and discourse state to a prewritten response. The personality of Julia is due to the crafting of these prewritten responses and the arbitration mechanism that chooses which response to give at any moment. These two elements have been developed over many years to give Julia competence in carrying on a conversation for a remarkable amount of time.

11.2.6 NL-Soar

Jill Fain Lehman and colleagues have built a real-time, responsive language generation system that is mixed with action in their NL-Soar system [Lehman *et al.* 1995; Rubinoff and Lehman 1994; Lonsdale in press]. Some of the work being done is in the context of the Soar IFOR systems in which Soar agents participate in military training simulations. These simulations are run in real-time with soldiers and computer agents both participating. The Soar agents control aircraft in scenarios in which their plane is outside of visual range of opponent aircraft. The goal of the work is for the Soar-controlled aircraft to be indistinguishable from those controlled by human pilots. The NL-Soar component has been used for radio communication in an experimental version of the system in this domain.

Because of the needs of their task, this work addresses many of the same problems as my work in natural language generation for believable agents described in Chapter 8. One high-level difference is that rather than striving for believability with a willing human participant, they necessarily have the goal of being indistinguishable from human pilots in the face of potentially sceptical human interactors. They are also working with the goal of cognitive plausibility because of Soar's focus as a theory of human cognition. Nevertheless the goal of supporting the types of communication that people really use, including all of the timing variations, mixtures with action, etc. is similar to what is needed for believability and to what I addressed in that portion of my work. My understanding is that because their task domain involves communication by radio outside of visual range, they have not pursued coordinated action and language to accomplish a single communication goal. Also, their agents as yet have no emotion model, so they have not explored this aspect of integration.

11.2.7 Gandalf

Kristinn Thórisson in his thesis work [Thórisson 1996] has built an interesting system that addresses many issues in face-to-face communication. His agent is rendered as a computer-animated face and associated hand. People interacting with the system must wear sensors and a close microphone to enable his agent to sense their position, sense what they are looking at, sense their hand position over time, and perform speech recognition. Using this system he has tested his theory for psychologically motivated, natural, high-bandwidth communication using speech, eye contact, and gesture. His agent participates in conversations by attempting to produce all of these modalities at moments appropriate to the ongoing conversation. Because of the focus of his research, his system does not attempt to express a personality, have an emotion model, have agency outside of the conversation, or other aspects needed for believable agents. It also uses prewritten text rather than performing natural language generation. Nevertheless, the techniques used in his system address a subset of the requirements for language use in believable agents, and could clearly be useful in such agents.

11.2.8 Other Work

"Teo — The Other Earth" is a commercial product by Fujitsu Limited. Teo features a believable agent, called Fin Fin, that people can interact with and build a relationship with over time. The system uses a version of Hap along with other proprietary technology.

The Persona Project [Ball *et al.* in press] at Microsoft Research has built a "conversational assistant" for which believability is a part. Their system uses speech recognition and natural language understanding to drive a conversational state machine. This machine causes their animated bird to play prerecorded segments of audio and large-grained segments of behavior to participate in the conversation.

For his master's thesis, Brad Rhodes built an agent architecture that focuses on action selection for believable agents [Rhodes 1996]. The system addresses some ways of expressing an interactive personality through action-selection.

Clark Elliott's work on the Affective Reasoner [Elliott 1992; 1997] focuses on models of emotion, and how emotions are expressed. In the pursuit of this work he has built some autonomous agents, for example poker playing agents. The agents from his affective reasoner express themselves using synthesized speech with emotional variation, music chosen to express emotion, and through 60 schematic faces. The schematic faces have each been drawn to express a particular emotion, and the system morphs between faces when the internal emotion changes.

Rich and colleagues [Rich *et al.* 1994] have integrated a remarkably wide range of capabilities in their system, including language understanding, speech synthesis and recognition, low-level motor control, and simple cognition and emotion. Their broad integration takes a traditional approach in that there are distinct components for each of the capabilities and they communicate through relatively narrow channels. This results in latencies that degrade believability, but the work nonetheless represents a high-water mark in integration.

Cassell and colleagues have built autonomous agents that interact with each other using speech synthesis and gesture [Cassell *et al.* 1994] using the Jack human simulation system of Badler, Webber and colleagues [Badler *et al.* 1990]. Webber and Badler have more recently started exploring social behavior and personality-based variation [Badler *et al.* 1997]. Nagao and Takeuchi [Nagao and Takeuchi 1994] have a system for speech synthesis and facial animation. Nadia Magnenat Thalmann and Daniel Thalmann [Thalmann and Volino 1997; Thalmann *et al.* 1997] have been working toward a wide range of issues needed for realistic computer-animated human actors.

Related to my work in natural language generation is the work of Walker, Cahn and Whittaker [Walker *et al.* 1997]. They have built a system that makes linguistic choices based on perceived position within a given social situation. The system includes Cahn's approach for emotionally varied speech synthesis [Cahn 1989].

Scott Neal Reilly and Phoebe Sengers have been long-time collaborators with me and researchers in the area believable agents. The model of emotion presented in Chapter 6 is Neal Reilly's work, and for his thesis work he created and integrated with Hap a richer and more complete model of emotion for believable agents. As part of that work, he also created an approach to building social knowledge for believable agents, and several agents to test his approach [Neal Reilly 1996]. Sengers is working to build agents that combine many behaviors while remaining coherent and understandable as a whole [Sengers 1996]. Both of their work has included the construction of believable agents using the framework described in this thesis.

11.3 Future Directions

There are a number of directions that I think would be promising avenues for future work. In this section, I present several of these ordered from most specific and concrete to most speculative and far reaching.

11.3.1 Build a Complete Agent to Test the NLG Approach

The approach described in Chapter 8 to natural language generation for believable agents is preliminary. It has only been used for a small grammar and in portions of an agent. A natural next step in this work is to use this approach for the creation of complete agents and a grammar large enough for their needs.

11.3.2 Natural Language Understanding for Believable Agents

As mentioned in Chapter 8, to fully gain the benefits of natural language for believable agents, the agents need to understand as well as generate natural language. Such understanding needs to take the requirements of believable agents seriously, just as is necessary for generation and the rest of a believable agent. Previously, I have discussed some of these requirements as they affect NLU [Loyall 1997], but part of the process of building such a capability would be to fully understand how the believability requirements should affect natural language understanding.

11.3.3 Richer Physical Bodies

The physical bodies that my architecture has controlled have been very simple. This is not necessarily a limiting factor for believability. Much can be done with simple physical bodies. The Disney animators found that they could draw a simple half-filled four sack, without even eyes, so that it conveyed a range of emotions and actions [Thomas and Johnston 1981, p. 49]. Using the simple bodies of the Woggles, I do not think we have even begun to push the expressiveness that they are capable of. Nevertheless, I think an interesting avenue of research is use this technology to control richer, more complicated physical bodies. Such exploration could give insight into the complexity that can be controlled using this technology, and would likely raise new technological and artistic research issues in the process.

11.3.4 Better Tools for the Authoring Process

Authoring a believable agent is not an easy task. There are many details to create, issues of consistency, and the normal problems of creating an autonomous agent. Given this, a promising direction for work would be to create tools that make the authoring process easier. This must be undertaken with care, because as I argue earlier, tools that make the process easier at the expense of expressiveness or undermining believability are not very useful tools.

Also, it is not necessary that such tools be easy for existing artists to use. Historically, artists have had to adapt to the demands and opportunities of each new medium as it has appeared. Those who did not learn and adapt to the new medium did not fare as well as those who did. Film is a good example of this. The advantages and expressiveness of this medium — cuts, filmic montage, zooms, close ups, two shots, etc. — were unknown before film, and yet one would be hard pressed to find a film today that doesn't use them. They are a language of expression that is unique to film, and that has to be learned by aspiring film students. The same is true in believable agents. This is a new medium that is truly a combination of art and technology. The tools should respect the strengths of the medium, and augment them, perhaps requiring training to be used well.

11.3.5 Deep, Fundamental Merging with Traditional Artificial Intelligence Technology

This dissertation has presented technology that was specifically created for believable agents. The technology draws from particular portions of traditional artificial intelligence. In the process of responding to the requirements of believability, the AI technology has been adapted for the needs of believability, as well as being used differently than in traditional artificial intelligence.

This dissertation only touches on a subset of the areas of artificial intelligence, however. Areas such as planning, learning, speech synthesis and vision have not been addressed. Many of these could be usefully applied to the needs of believable agents. In the process, it is likely that richer believable agents could be made, and that these research areas would be faced with new, interesting challenges, that they could be fruitfully adapted to meet. This, I think, is a promising avenue of research for many subareas of AI, but I want to mention two in particular.

Intelligence, Reasoning and Planning

As I mention in Chapter 1, intelligence is not the main focus of believability. There are many other problems and qualities that are more centrally important. Nevertheless, reasoning, planning and in general the goal of making agents intelligent is a part of many characters in the arts, and could be a powerful part of believable agents.

Traditionally, these areas of research have used as their metric of success optimality, satisficing solutions, appropriate utility, or cognitive plausibility. For these technologies to be used in a believable agent, there are other, different challenges. For example, how should a planner or reasoning system vary its processing for different emotional states of the agent? How can planners or general-purpose reasoners perform their activities so that they are appropriate to the social situation? How can one make a planner that plans in a style appropriate to an artistically chosen personality? These issues and others that arise from the requirements of believable agents are important if such systems are to be used in believable agents.

I, personally, think they will have deep ramifications for the future of planning and reasoning research, but such a conjecture can only be tested by time.

Learning

Similarly, learning has historically been driven by utilitarian goals. As described in Section 2.4, for believable agents the requirements are clear: change should be appropriate to the personality in which it takes place. Creating learning approaches that are true to the requirements of believability is a challenging, interesting, and important area of research for believable agents.

11.3.6 Continually Attempt to Build Better Believable Agents

In addition to those specific goals, there are two ongoing goals that I believe are important for progress in believable agents. The first is to continually attempt to build better believable agents. This is the fundamental driving force of this area of research. It is necessary to guide the technology, and it is necessary to judge progress in the field because such judgements are necessarily subjective.

11.3.7 Continually Study Existing Art Forms

I believe it is also important to continually study existing character-based art forms. They have knowledge about the problem that we are only beginning to understand, and that can guide researchers in this field to understand how to achieve the dream of creating believable agents.

Appendix A

Hap Grammar

```
<hap-character>    --> (hap_ral_character <number>
                          (global_variables <var>*)
                          (include_C <RAL-code>)
                          (include_rules <RAL-rules>)
                          (production_memory
                             <production>*)
                          (initial_active_plan_tree
                             (persistent <step>*)
                             (one_shot <step>*)))

<production>      --> (<type> <name> (<formals>)
                              [<precondition>]
                              [<context-cond>]
                              [(specificity <number>)]
                              [(locals (<var> <value>)*)]
                              [(dynamic_locals (<var> <value>)*)]

                         <step>*
                         [(return_values <value>*)])
                      (include <filename>)
                      (link <filename>)

<type>            --> sequential_behavior
                      parallel_behavior
                      collection_behavior
                      sequential_production
                      parallel_production
                      collection_production

<name>            --> any legal C identifier optionally in a string.

<formals>         --> <var>*
```

```
<precondition>    --> (precondition <test>
                                  <sensor-form>*
                                  [(:code <statement>*)])

<context-cond>    --> (fail_when <test>
                               <sensor-form>*
                               [(:code <statement>*)])

<var>             --> any legal C identifier optionally in a string.

<test>            --> (<LHS>+)

<LHS>             --> a string containing any RAL LHS clause.

<number>          --> any integer

<sensor-form>     --> "<sensor-name>,arg1,...,argn"
                      (e.g. "ESColor,1,156,-1,-1")

<C-args>          --> <expression> | <expression>,<C-args>
                      where <expression> is any valid RAL expression.

<sensor-name>     --> The valid sensor names are given in event.h in
                      the #define for ES_TYPES

<step>            --> (with [<success-test>] [<priority-mod>]
                            [(importance <number>)]
                            [ignore_failure]
                            [persistent | (persistent <persistent-type>)]
                            [effect_only]
                            [(bind_to <formals>)]
                            [(dbind_to <formals>)]
                            [(bind_to_no_shadowing <formals>)]
                            [(dbind_to_no_shadowing <formals>)]
                            [(set_to <formals>)]
                            [in_lexical_scope]
                         <abstract-act>)
                      <abstract-act>

<persistent-type> --> when_fails
                      when_succeeds

<success-test>    --> (success_test <test>
                                    <sensor-form>*
                                    [(:code <statement>*)])
```

```
<priority-mod>    --> (priority_modifier <number>)
                      (priority <number>)

<abstract-act>    --> (act <act-name> <value>*)
                      (mental_act <statement>)
                      (subgoal <name> <value>*)
                      (wait)
                      (breed_goal <value> <name> <value>*)

<value>           --> a string containing a C form that returns a
                      value (e.g. "a+3", "sqrt(3)", etc.)

<statement>       --> any valid RAL or C statement in quotes.
```

Appendix B

Natural Language Generation Grammar

This appendix describes a small example natural language generation grammar that is supported by Hap. The main description is in the form of an annotated context free grammar in Section B.1. Because the generation process in Hap is not fully captured by a context free grammar, annotations are added to note information flow, conditions for choosing between competing clauses, physical and mental actions, and parallelism in the grammar. The grammar description is also divided into sections that provide additional information about the underlying process being described.

The later sections describe supporting information for this grammar. Section B.2 describes the basic group data structure whose content drives the generation process along with the grammar. Section B.3 presents the abstraction hierarchy used for matching in the conditional annotations of the grammar, and Section B.4 describes the rules that smooth the sequence of strings that results from the generation process.

B.1 Annotated Context Free Grammar

This grammar description uses an annotated context free grammar. To illustrate the notational conventions used for the base CFG and the annotated information, consider the following clause.

Relation1	→	Subject ((`case nominative`) hearer)
(feeling sad and desired_ans is 'yes')		<infer time>
		Predicate (number person (`negative t`) (`sv_order sv`) r_time modal voice)
		ObjectPhrase
		`","`
		Predicate (number person (`aux only`) (`sv_order vs`) r_time modal voice)

Subject `((case nominative)` hearer `(pronominalize t))`
<return `(presumed_ans no)`>

The base CFG is composed of uppercase non-terminal terms such as Relation1 and Subject and strings as terminals (e.g. `"to"`). The pure context free grammar portion of the above clause is:

Relation1	$\rightarrow$	Subject
		Predicate
		ObjectPhrase
		`","`
		Predicate
		Subject

Adjacent terms (on the same line or in consecutive lines) are interpreted as sequential. Thus, this clause specifies that a relation is a non-terminal Subject followed by Predicate, ObjectPhrase, the terminal string `","`, and the non-terminals Predicate and Subject. Alternatives are expressed in the normal way using the symbol | or by using multiple clauses for the same non-terminal.

To this context-free grammar notation, I add four annotations. These annotations describe conditions for choosing between alternatives, physical or mental actions in the grammar, data flow in the grammar, and parallelism in the grammar. Each of these notations is described in turn below.

- Under the left-hand non-terminal is information about when this clause is applicable. The above example includes the annotation (feeling sad and desired_ans is 'yes') to the clause. Only key preconditions are listed. Others are left implied, or are described in Chapter 8.

- Physical and mental actions are expressed as terms of the grammar using the notation <...>. The above example includes two such notations: <infer time> and <return `(presumed_ans no)`>. There are three types of these actions in the grammar. Actions of the form <infer ...> are mental actions whose function are described in the clauses given in Section B.1.4. Terms of the form <return ...> describe information flow to the later parts of the generation process. Other <...> terms are self descriptive. Like the rest of the grammar, only key return values are included to avoid obscuring the description with a large number of them.

- Downward information flow is noted by the argument annotations often included after non-terminals on the right-hand side of a clause. They are typeset in a small font to distinguish them from the associated non-terminal. In the above annotated clause, number and person (along with other arguments) are arguments to the Predicate non-terminals. (As described in Chapter 8, `number` and `person` are returned by the generation of the Subject. These return values are noted in the grammar with appropriate <return ...> forms as above.

- The fourth annotation is not illustrated in the above example. The symbol ⊕can be included to note parallelism in the grammar.

Terms in sans-serif font, for example Subject or hearer above, represent references to parts of the group structure being generated. Terms in fixed-width font, for example `"to"`, `","` or `(negative t)`, are literals. Terms in normal font such as Relation1 are intermediate parts of the grammar description with no special meaning. Each sans-serif term on the left-hand side of one of the productions in this section are entry points in the generation process. Thus an agent can use this grammar to generate, for instance, a stand-alone Sentence, a Greeting, or Predicate.

Each non-terminal also has an implicit clause that allows it to result in nothing. These clauses have been omitted for brevity.

The grammar is divided into four sections. The first, and main, section (Section B.1.1) describes how different `types` of groups can be generated. Each clause with a sans-serif left-hand side term expresses how a group of that type can be generated. The non-terminals on the right-hand side of these clauses each references subgroups or features of the given group that is being generated by role. What type of group each role reference results in depends on the structure of the group being generated. A particular concept being expressed could have a group of type action to express the predicate role of a relation group or a group of type parameter that specifies the particular lexical item to express the predicate. These possible mappings (extendible by an author) are described in Section B.1.2. Section B.1.3 describes constant elements of the generation grammar, and finally, Section B.1.4 describes the inference portions of the generation process.

B.1.1 Type-Based Clauses

Sentence	→	`"#"`
		Greeting (hearer)
		Matrix
		Punctuation (`(value end_of_sentence)` mood)
Greeting (hearer is looking at me)	→	nothing

Greeting → Agent `((value` hearer`))`
Punctuation `((value end_of_pref))`

(hearer is not looking at me and is nearby)
(Note: hearer feature is not passed in to generation of Agent.)

Greeting → `"hey"`
Agent `((value` hearer`))`
Punctuation `((value end_of_pref) (mood imperative))`

(hearer is not looking at me and is not nearby)

Relations

Relation → <infer voice>
<infer subject/object>
Relation1

Relation1 → Subject `((case nominative)` hearer)
<infer time>
Predicate `((sv_order sv)` r_time modal voice number person)
ObjectPhrase

Relation1 → Predicate `((sv_order sv) (r_time infinitive)` voice)
ObjectPhrase

(parent_predicate is an infinite verb and hearer is the same as enclosing clause's subject)

Relation1 → `"for"`
Subject `((case objective)` hearer)
Predicate `((sv_order sv) (r_time infinitive)` voice)
ObjectPhrase

(parent_predicate is an infinite verb)

Relation1 → Subject `((suppress_output t) (case nominative)` hearer)
<infer time>
Predicate `((aux only) (sv_order vs)` r_time modal voice number person)
Subject `((case nominative)` hearer)
Predicate `((aux skip_first) (sv_order vs)` r_time modal voice number person)
ObjectPhrase

Relation1	$\longrightarrow$	Subject `((case nominative)` hearer) <infer time> Predicate `((negative t) (sv_order sv)` r_time modal voice number person) ObjectPhrase `","` Predicate `((aux only) (sv_order vs)` r_time modal voice number person) Subject `((case nominative)` hearer `(pronominalize t))` <return `(presumed_ans no)`>
Relation1	$\longrightarrow$	Subject `((case nominative)` hearer) <infer time> Predicate `((sv_order sv) (direct t)` r_time modal voice number person) ObjectPhrase `","` Predicate `((aux only) (sv_order vs) (negative t) (use_contractions t)` r_time modal voice number person) Subject `((case nominative)` hearer `(pronominalize t))` <return `(presumed_ans yes)`>

Objects

Obj_Parameter	$\longrightarrow$	Role (relation_role) Obj_Arg (features) <return number person>
Role (value is object, and relation_role is actor) *(Actor is in object position.)*	$\longrightarrow$	`"by"`
Agent (value is me)	$\longrightarrow$	Agentme <return `(number singular) (person 1)`>
Agent (value is hearer)	$\longrightarrow$	Agentyou <return `(number singular) (person 2)`>

Agent (value is last_agent_referenced or pronominalize is true) *(Woggles don't have a gender, but "it" is too impersonal when they refer to each other.))*	→	Agentshe <return `(number singular) (person 3)`>
Agent	→	Agentbyname <return `(number singular) (person 3)`>
Agentme (case is nominative, objective, genitive or reflexive respectively)	→	`"I"` | `"me"` | `"my"` | `"myself"`
Agentyou (case is nominative, objective, genitive or reflexive respectively)	→	`"you"` | `"you"` | `"your"` | `"yourself"`
Agentshe (case is nominative, objective, genitive or reflexive respectively)	→	`"she"` | `"she"` | `"her"` | `"herself"`
Agentbyname	→	`"Shrimp"` | `"Bear"` | `"Wolf"` | `"User"`
ObjectPhrase	→	Object `((case objective)` voice hearer `(parent_subject projector(`subject`)) (parent_predicate projector(`predicate`)))` Location Manner

Verb/Action Support

Action (role = `predicate`)	→	<infer predicate> Predicate (features) <return predicate>
Parameter (projector is a verb)	→	<setup dynamic locals aux_generated and suppress_output> <set suppress_output to 1 if (aux skip_first)> Paramverb1

Paramverb1	→	<demon: if suppress_output = 1 and aux_generated = 1, then set suppress_output = 0>
	⊕	Paramverb2
Paramverb2	→	R_time
		<infer modal>
		Modal (inflection negative number person)
		Perfect (inflection negative number person)
		Progressive (inflection negative number person)
		Voice (inflection negative number person)
		Arg (inflection negative number person)
R_Time	→	"to"
(value = infinitive)		
(value = past)	|	<return (inflection past)>
(value = not_past)	|	<return (inflection not_past)>
Modal	→	Will ((inflection past) negative number person)
(value = conditional)		
(value = emphatic)	|	Do (features)
(value = intent)	|	Will (features)
(value = possibility)	|	May (features)
(value = ability)	|	Can (features)
(value = obligation)	|	Must (features)
Perfect	→	Have (features)
(value = t)		<return (inflection past_part)>
Progressive	→	Progressive1
		<return (inflection pres_part)>
Progressive1	→	Be (features)
(value = t or region)		
(value = begin)	|	Begin (features)
(value = finish)	|	Finish (features)
(value = start)	|	Start (features)
(value = stop)	|	Stop (features)
(value = continue)	|	Continue (features)
Voice	→	Be (features)
(value = passive)		<return (inflection past_part)>

Arg	$\longrightarrow$	Word (number person inflection negative)
(projector is a verb)		Particle
		Adverb
Word	$\longrightarrow$	Root
(projector is a verb)		Inflection (number person)
Word	$\longrightarrow$	<infer contraction>
(projector is an auxiliary verb)		Root (inflection number person contraction)
		Negative (contraction)
		<add one to aux_generated>

Regular Verbs

Root	$\longrightarrow$	`"want"`
(value = want)		
(value = like)	|	`"like"`
(value = play)	|	`"play"`
(value = finish)	|	`"finish"`
(value = start)	|	`"start"`
(value = stop)	|	`"stop"`
(value = continue)	|	`"continue"`
(value = cause)	|	`"cause"`
(value = lead)	|	`"lead"`
(value = follow)	|	`"follow"`
Inflection	$\longrightarrow$	`"s"`
(value = not_past, person = 3, and number = singular)		
(value = past or past_part)	|	`"ed"`
(value = pres_part)	|	`"ing"`
Negative	$\longrightarrow$	`"n't"`
(contraction =aux)		
Negative	$\longrightarrow$	`" " "not"`

Irregular and Auxiliary Verbs

Root (value = be) *(Choices based on inflection, number, person and contraction values.)*	→	`"been"` | `"being"` | `"’m"` | `"am"` | `"’re"` | `"are"` | `"’s"` | `"is"` | `"’re"` | `"are"` | `"was"` | `"were"` | `"was"` | `"were"` | `"be"`
Word (projector is be, inflection = not_past, person = 1, number = singular, and negative = t) *(Needed to override "amn’t".)*	→	`"ain’t"` | `"am not"` <add one to aux_generated>
Root (value = can) *("Ca" is used when inflection = not_past and contraction = aux)*	→	`"could"` | `"can"` | `"ca"` | `"been able"` | `"able"`
Word (projector is can, negative = t, and contraction is not aux)	→	`"cannot"` <add one to aux_generated>
Root (value = will) *(Abbreviated forms are generated when contraction = subj or aux.)*	→	`"would"` | `"’d"` | `"wo"` | `"will"` | `"’ll"`
Root (value = do)	→	`"done"` | `"doing"` | `"did"` | `"does"` | `"do"`
Root (value = have)	→	`"had"` | `"having"` | `"has"` | `"’ve"` | `"have"`
Root (value = may)	→	`"might"` | `"may"`
Word (projector is may and negative = t) *(Override "mightn’t" and "mayn’t".)*	→	`"might not"` | `"may not"` <add one to aux_generated>
Root (value = must)	→	`"must"`

Miscellaneous

Location	$\rightarrow$	<glance at location>
(language_only $\neq$ t)	$\oplus$	Location `((pronominalize t) (language_only t))`
Location	$\rightarrow$	`"over" "there"`
Location	$\rightarrow$	`"on" "the" "hill"`
(value = `hill`)		
Manner	$\rightarrow$	`"quickly"` | `"slowly"`
(value = quickly or slowly)		
Punctuation	$\rightarrow$	`"."` | `"!"` | `"?"`
(value = end_of_sentence) *(mood is declarative, imperative, or interrogative)*		
Punctuation	$\rightarrow$	`","`
(value = end_of_pref)		
Punctuation	$\rightarrow$	`"!"`
(value = end_of_pref and mood is imperative)		

B.1.2 Role-based Clauses

Matrix	$\rightarrow$	Relation
Subject	$\rightarrow$	Obj_parameter
Object	$\rightarrow$	Obj_parameter
Obj_Arg	$\rightarrow$	Agent | Relation
Predicate	$\rightarrow$	Action | Parameter

B.1.3 Constant Clauses

Will	$\rightarrow$	Word
Do	$\rightarrow$	Word
May	$\rightarrow$	Word
Can	$\rightarrow$	Word

Must	→	Word
Have	→	Word
Be	→	Word
Begin	→	Word
Finish	→	Word
Start	→	Word
Stop	→	Word
Continue	→	Word

Constant Groups

Will	→	`(word (type word) (root will))`
Do	→	`(word (type word) (root do))`
May	→	`(word (type word) (root may))`
Can	→	`(word (type word) (root can))`
Must	→	`(word (type word) (root must))`
Have	→	`(word (type word) (root have))`
Be	→	`(word (type word) (root be))`
Begin	→	`(word (type word) (root begin))`
Finish	→	`(word (type word) (root finish))`
Start	→	`(word (type word) (root start))`
Stop	→	`(word (type word) (root stop))`
Continue	→	`(word (type word) (root continue))`

B.1.4 Inference Clauses

<infer voice> *(uses focus, defocus, agent and receiver subparts for decision.)*	→	<return `(voice active)`> | <return `(voice passive)`>

```
<infer                  →    <return ((subject (type obj_parameter)
  subject/object>                               (relation_role actor)
                                                (obj_arg (actor)))
                                       (object (type obj_parameter)
                                               (relation_role receiver)
                                               (obj_arg (receiver))))>
                        |    <return ((subject (type obj_parameter)
                                                (relation_role receiver)
                                                (obj_arg (receiver)))
                                       (object (type obj_parameter)
                                               (relation_role actor)
                                               (obj_arg (actor))))>

<infer time>            →    <return ((r_time not_past) (modal intent))>
   (time = future)
   (time = present)     |    <return (r_time not_past)>
   (time = past)        |    <return (r_time past)>
   (time = infinitive)  |    <return (r_time infinitive)>
   (default)            |    <return (r_time not_past)>

<infer contraction>     →    <return (contraction aux)>
   (use_contractions = t and
    negative = t)

<infer contraction>     →    <return (contraction subj)>
   (use_contractions = t, sv_order
    = sv, aux_generated = 0)
   (need to make sure that this
    isn't the only verb that will
    be generated, if that would
    be inappropriate for this
    verb type, and that prev word
    is subject.)

<infer modal>           →    <return (modal emphatic)>
   (negative = t, voice = active,
    there is no modal or perfect
    feature, and there is no
    progressive feature or
    progressive = t or region)

<infer modal>           →    <return (modal emphatic)>
   (sv_order = vs, voice = active,
    there is no modal or perfect
    feature, and there is no
    progressive feature or
    progressive = t or region)
```

```
<infer predicate>     →    <return (predicate (type parameter)
   (action = desire)                  (modal conditional)
                                      (arg (type arg)
                                           (word (type word)
                                                 (root like))))>
                      |    <return (predicate (type parameter)
                                      (arg (type arg)
                                           (word (type word)
                                                 (root want))))>

<infer predicate>     |    <return (predicate (type parameter)
   (action = play)                    (arg (type arg)
                                           (word (type word)
                                                 (root play))))>

<infer predicate>     |    <return (predicate (type parameter)
   (action = cause)                   (arg (type arg)
                                           (word (type word)
                                                 (root cause))))>

<infer predicate>     |    <return (predicate (type parameter)
   (action = lead)                    (arg (type arg)
                                           (word (type word)
                                                 (root lead))))>

<infer predicate>     |    <return (predicate (type parameter)
   (action = follow)                  (arg (type arg)
                                           (word (type word)
                                                 (root follow))))>
```

B.2 Typical Group Structure

Groups are general-purpose data structures, and can be therefore used with many structures. Typically, they follow structure similar to the following.

```
(sentence (type sentence) (hearer $$follower)
          (matrix (type relation)
                  (actor (type agent) (value $$follower))
                  (predicate (type action) (value desire))
                  (receiver (type relation)
                            (actor (type agent) (value $$follower))
                            (location $$where)
                            (predicate (type action) (value play)))))
((mood interrogative_yn) (desired_ans yes))
```

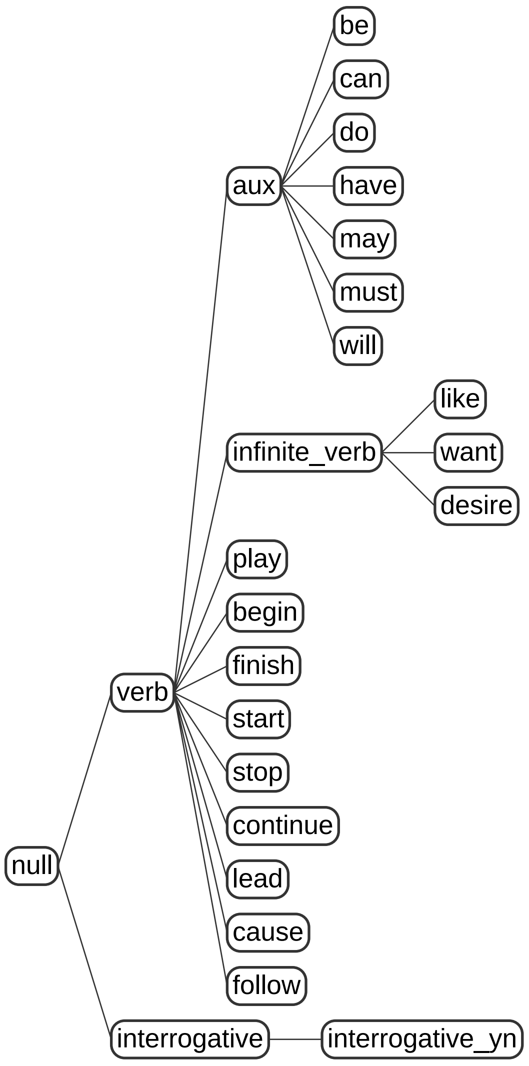
B.2.1 Type to Head Mapping

Each group of a given type has one of its roles defined as the head of the group. As described in Chapter 8, this is used to determine the `projector` of a group. This is the mapping from group types to the role that specifies the head of the group in my grammar. These are not part of the grammar, they are only used for determining the projector of a group.

`(type sentence)`	→	`(role matrix)`
`(type relation)`	→	`(role predicate)`
`(type parameter)`	→	`(role arg)`
`(type obj_parameter)`	→	`(role arg)`
`(type arg)`	→	`(role word)`
`(type word)`	→	`(role root)`
`(type agent)`	→	`(role value)`
`(type action)`	→	`(role value)`

B.3 Abstraction Hierarchy

This graph specifies the abstraction hierarchy used for match expressions in the grammar.



B.4 Combination Knowledge

This section presents the knowledge to smooth the sequence of strings that results from the generation process. For each pair of strings in sequence, the rules expressed in the following chart are applied if applicable. To be applicable the common parent group in the generation process must be of appropriate type, the end of the left string must match the first regular expression, and the beginning of the right string must match the second regular expression. In that case the changes in the last column are performed on the two strings.

Type	End of Left	Beginning of Right	Changes
any	ϵ `#`	any	Capitalize first character of right
~word	~[ϵ`#‘␣`]	~`[,’.!?]`	add an interword space
~word	consonant `y`	~`[ia]`	change `y` to `i`
arg	ϵ `a`	vowel | `h`	change `a` to `an`
word	vowel consonant	vowel	double the consonant (if the vowel on left is stressed)
word	vowel `c`	vowel	change `c` to `ck`
word	`ch` | `sh` | `[sxzo]`	`s`	change `s` to `es`
word (projector is verb)	`[aeo]` `[rla]` f	`s`	change `s` to `es`
word (projector is noun)	`[aeo]` `[rla]` f	`s`	change `f` to `v` and change `s` to `es`
word	consonant `y`	`s`	change `y` to `i` and change `s` to `es`
word	`[cg]` `e`	`[ei]`	remove `e` from left
word	`ie`	`i`	change `ie` to `y`
word	~(`c` | `g` | vowel) `e`	vowel	delete `e`

Bibliography

[Agents 1997] *Proceedings of the First International Conference on Autonomous Agents*, Marina del Rey, CA, 1997. Published by ACM Press.

[Agre and Chapman 1987] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, July 1987.

[Agre and Chapman 1990] Philip E. Agre and David Chapman. What are plans for? In *Robotics and Autonomous Systems*. Elsevier Science Publishers, 1990.

[Appelt 1985] Douglas E. Appelt. *Planning English Sentences*. Cambridge University Press, 1985.

[Badler *et al.* 1990] Norman Badler, Bonnie Webber, Jeff Esakov, and Jugal Kalita. *Making Them Move: Mechanics, Control, and Animation of Articulated Figures*. Morgan-Kaufmann, 1990.

[Badler *et al.* 1997] Norman I. Badler, Barry D. Reich, and Bonnie L. Webber. Toward personalities for animated agents with reactive and planning behaviors. In Robert Trappl and Paolo Petta, editors, *Creating Personalities for Synthetic Actors, Towards Autonomous Personality Agents*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 1997.

[Ball *et al.* in press] Gene Ball, Dan Ling, David Kurlander, John Miller, David Pugh, Tim Skelly, Andy Stankosky, David Thiel, Maarten Van Dantzich, and Trace Wax. Lifelike computer characters: the Persona project at Microsoft Research. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, in press.

[Bates *et al.* 1992] Joseph Bates, A. Bryan Loyall, and W. Scott Reilly. Integrating reactivity, goals, and emotion in a broad agent. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, IN, July 1992.

[Bates *et al.* 1994] Joseph Bates, A. Bryan Loyall, and W. Scott Reilly. An architecture for action, emotion, and social behavior. In Cristiano Castelfranchi and Eric Werner, editors, *Artificial social systems : 4th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '92, S. Martino al Cimino, Italy, July 29-31,* *1992 : selected papers*, Lecture notes in artificial intelligence. Springer-Verlag, Berlin, 1994.

[Bates 1992] Joseph Bates. Virtual reality, art, and entertainment. *PRESENCE: Teleoperators and Virtual Environments*, 1(1):133–138, 1992.

[Benedetti 1985] Jean Benedetti. *Stanislavski, an introduction*. Methuen, London, 1985.

[Blumberg 1994] Bruce Blumberg. Action-selection in Hamsterdam: Lessons from Ethology. In *From Animals To Animats, Proceedings of the Third International Conference on the Simulation of Adaptive Behavior*, 1994.

[Blumberg 1996] Bruce Mitchell Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, Media Arts and Sciences, Massachusetts Institute of Technology, 1996.

[Braitenberg 1984] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA, 1984. Bradford books.

[Brooks 1986] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.

[Cahn 1989] Janet Cahn. Generating expression in synthesized speech. Master's thesis, Media Arts and Sciences, Massachusetts Institute of Technology, 1989.

[Cassell *et al.* 1994] Justine Cassell, Catherine Pelachaud, Norman Badler, Mark Steedman, Brett Achorn, Tripp Becket, Brett Douville, Scott Prevost, and Matthew Stone. Animated conversation. *Computer Graphics*, 28, 1994. Proceedings of SIGGRAPH '94.

[Chaplin 1964] Charles Chaplin. *My Autobiography*. Simon and Schuster, New York, 1964.

[Chapman 1990] David Chapman. *Vision, Instruction, and Action*. PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1990.

[Clements and Musker 1992] Ron Clements and John Musker, directors. *Aladdin*. Walt Disney Productions, 1992.

[Curtiz 1942] Michael Curtiz, director. *Casablanca*. Warner Brothers, 1942.

[Egri 1960] Lajos Egri. *The Art of Dramatic Writing; Its Basis in the Creative Interpretation of Human Motives*. Simon and Schuster, revised edition of 1946 book by same title. edition, 1960. Originally published by Simon and Schuster in 1942 as *How to write a play*.

[Elliott 1992] Clark Elliott. *The Affective Reasoner: A Process Model of Emotions in a Multi-agent System*. PhD thesis, Institute for the Learning Sciences, Northwestern University, 1992.

[Elliott 1997] Clark Elliott. I picked up catapia and other stories: A multimodal approach to expressivity for 'emotionally intelligent' agents. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[Enters 1965] Angna Enters. *On Mime*. Wesleyan University Press, Middletown, CT, 1965.

[Fikes and Nilsson 1971] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Fikes *et al.* 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[Firby 1989] James R. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Department of Computer Science, Yale University, 1989.

[Foner 1997] Leonard N. Foner. Entertaining agents: A sociological case study. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[Forgy 1982] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Forgy 1991] Charles L. Forgy. *Rule-extended Algorithmic Language Language Guide*. Production Systems Technology, 5001 Baum Boulevard, Pittsburgh, PA 15213, May 1991.

[Gardner 1984] John Gardner. *The Art of Fiction: Notes on Craft for Young Writers*. Alfred A. Knopf, New York, 1984.

[Georgeff and Lansky 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, July 1987.

[Giannetti 1987] Louis Giannetti. *Fourth Edition Understanding Movies*. Prentice-Hall, 1987.

[Hammett 1957] Dashiell Hammett. *The Maltese Falcon*. Knopf, New York, 1957.

[Hayes-Roth and van Gent 1997] Barbara Hayes-Roth and Robert van Gent. Story-making with improvisational puppets. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[Hayes-Roth 1995] Barbara Hayes-Roth. An architecture for adaptive intelligent systems. *Artificial Intelligence: Special Issue on Agents and Interactivity*, 72:329–365, 1995.

[Hovy 1988] Eduard Hovy. *Generating Natural Language under Pragmatic Constraints*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[Huston 1941] John Huston, director. *The Maltese Falcon*. Warner Brothers, 1941.

[Jones 1989] Chuck Jones. *Chuck Amuck: the Life and Times of an Animated Cartoonist*. Farrar, Straus & Giroux, New York, 1989.

[Kaelbling and Rosenschein 1990] L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6(1-2):35–48, June 1990.

[Kantrowitz and Bates 1992] Mark Kantrowitz and Joseph Bates. Integrated natural language generation systems. In R. Dale, E. Hovy, D. Rosner, and O. Stock, editors, *Aspects of Automated Natural Language Generation*, volume 587 of *Lecture Notes in Artificial Intelligence*, pages 13–28. Springer-Verlag, 1992. (This is the Proceedings of the Sixth International Workshop on Natural Language Generation, Trento, Italy, April 1992.).

[Kantrowitz 1990] Mark Kantrowitz. Glinda: Natural language text generation in the Oz interactive fiction project. Technical Report CMU-CS-90-158, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Kantrowitz in press] Mark Kantrowitz. *Generating Natural Referring Expressions*. PhD thesis, Computer Science Department, Carnegie Mellon University, in press.

[Kelso *et al.* 1993] Margaret Thomas Kelso, Peter Weyhrauch, and Joseph Bates. Dramatic presence. *PRESENCE: Teleoperators and Virtual Environments*, 2(1), Winter 1993.

[Kurosawa 1954] Akira Kurosawa, director. *The Seven Samurai*. Toho, 1954.

[Lasseter 1986] John Lasseter, director. *Luxo Jr.* Pixar, 1986.

[Lasseter 1987a] John Lasseter. Principles of traditional animation applied to 3D computer animation. *Computer Graphics*, 21(4):35–44, 1987. Proceedings of SIGGRAPH ’87.

[Lasseter 1987b] John Lasseter, director. *Red’s Dream*. Pixar, 1987.

[Lasseter 1988] John Lasseter, director. *Tin Toy*. Pixar, 1988.

[Lasseter 1989] John Lasseter, director. *Knickknack*. Pixar, 1989.

[Lehman *et al.* 1995] Jill Fain Lehman, Julie Van Dyke, and Robert Rubinoff. Natural language processing for IFORs: Comprehension and generation in the air combat domain. In *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*, 1995.

[Lester and Stone 1997] James C. Lester and Brian A. Stone. Increasing believability in animated pedagogical agents. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[Lonsdale in press] Deryle Lonsdale. Modelling SI: A cognitive approach. *Interpreting: International Journal of Research and Practice in Interpreting*, in press.

[Loyall and Bates 1993] A. Bryan Loyall and Joseph Bates. Real-time control of animated broad agents. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, Boulder, CO, June 1993.

[Loyall 1997] A. Bryan Loyall. Some requirements and approaches for natural language in a believable agent. In Robert Trappl and Paolo Petta, editors, *Creating Personalities for Synthetic Actors, Towards Autonomous Personality Agents*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 1997.

[Maes *et al.* 1995] Pattie Maes, Trevor Darrell, Bruce Blumberg, and Alex Pentland. The ALIVE system: full-body interaction with autonomous agents. In *Proceedings Computer Animation '95*, 1995.

[Mauldin 1994] Michael L. Mauldin. Chatterbots, tinymuds, and the Turing test: Entering the Loebner prize competition. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[Meehan 1976] James Meehan. *The Metanovel: Writing Stories by Computer*. PhD thesis, Computer Science Department, Yale University, 1976. Research Report #74.

[Nagao and Takeuchi 1994] Katashi Nagao and Akikazu Takeuchi. Social interaction: Multimodal conversation with social agents. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.

[Neal Reilly 1996] W. Scott Neal Reilly. *Believable Social and Emotional Agents*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1996.

[Newell 1990] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.

[Nilsson 1984] Nils Nilsson. Shakey the robot. Technical Report 323, Artificial Intelligence Center, SRI International, 1984.

[Nilsson 1994] Nils J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[Ortony *et al.* 1988] A. Ortony, G. Clore, and A. Collins. *The Cognitive Structure of Emotions*. Cambridge University Press, 1988.

[Park 1992] Nick Park, director. *A Grand Day Out*. National Film & Television School, 1992.

[Park 1993] Nick Park, director. *The Wrong Trousers*. Aardman Animations, 1993.

[Park 1995] Nick Park, director. *A Close Shave*. Aardman Animations, 1995.

[Pearson *et al.* 1993] Douglas J. Pearson, Scott B. Huffman, Mark B. Willis, John E. Laird, and Randolph M. Jones. Intelligent multi-level control in a highly reactive domain. In *Proceedings of the International Conference on Intelligent Autonomous Systems*, Pittsburgh, PA, 1993.

[Perlin and Goldberg 1996] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. *Computer Graphics*, 29(3), 1996.

[Perlin 1995] Ken Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1), 1995.

[Reynolds 1987] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987. Proceedings of SIGGRAPH '87.

[Rhodes 1996] Bradley James Rhodes. Phish-nets: Planning heuristically in situated hybrid networks. Master's thesis, Media Arts and Sciences, Massachusetts Institute of Technology, 1996.

[Rich *et al.* 1994] Charles Rich, Richard C. Waters, Yves Schabes, William T. Freeman, Mark C. Torrance, Andrew R. Golding, and Michal Roth. An animated on-line community with artificial agents. *IEEE MultiMedia*, 1(4):32–42, Winter 1994.

[Rosenbloom *et al.* 1993] Paul Rosenbloom, John Laird, and Allen Newell. *The Soar Papers: Readings on Integrated Intelligence*. MIT Press, Cambridge, MA, 1993.

[Rubinoff and Lehman 1994] Robert Rubinoff and Jill Fain Lehman. Real-time natural language generation in NL-Soar. In *Proceedings of 7th International Workshop on Natural Language Generation*, June 1994.

[Sengers 1996] Phoebe Sengers. Symptom management for schizophrenic agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eigth Innovative Applications of Artificial Intelligence Conference*. AAAI Press, 1996.

[Seuss 1975] Dr. Seuss. *Oh, the THINKS You Can Think!* Random House, New York, 1975.

[Silver 1988] Joan Micklin Silver, director. *Crossing Delancey*. Warner Brothers, 1988.

[Simmons 1991] Reid Simmons. Concurrent planning and execution for a walking robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Sacramento, CA, 1991.

[Sloman 1987] Aaron Sloman. Motives mechanisms and emotions. *Emotion and Cognition*, 1(3):217–234, 1987.

[Stanislavski 1961] Constantin Stanislavski. *Creating a role*. Theatre Arts Books, New York, 1961. edited by Hermine I. Popper. translated by Elizabeth Reynolds Hapgood.

[Stanislavski 1968] Constantin Stanislavski. *Stanislavski's Legacy*. Theatre Arts Books, New York, revised and expanded edition edition, 1968. edited and translated by Elizabeth Reynolds Hapgood.

[Stone and Lester 1996] Brian A. Stone and James C. Lester. Dynamically sequencing an animated pedagogical agent. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[Suchman 1987] Lucy A. Suchman. *Plans and situated actions: The problem of human machine communication*. Cambridge University Press, 1987.

[Thalmann and Volino 1997] Nadia Magnenat Thalmann and Pascaal Volino. Dressing virtual humans. In Robert Trappl and Paolo Petta, editors, *Creating Personalities for Synthetic Actors, Towards Autonomous Personality Agents*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 1997.

[Thalmann *et al.* 1997] Daniel Thalmann, Hansrudi Noser, and Zhiyong Huang. Autonomous virtual actors based on virtual sensors. In Robert Trappl and Paolo Petta, editors, *Creating Personalities for Synthetic Actors, Towards Autonomous Personality Agents*, Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 1997.

[Thomas and Johnston 1981] Frank Thomas and Ollie Johnston. *Disney Animation: The Illusion of Life*. Abbeville Press, New York, 1981.

[Thórisson 1996] Kristinn Rúnar Thórisson. *Communicative Humanoids: A Computational Model of Psychosocial Dialogue Skills*. PhD thesis, Media Arts and Sciences, Massachusetts Institute of Technology, 1996.

[Walker *et al.* 1997] Marilyn A. Walker, Janet E. Cahn, and Stephen J. Whittaker. Improvising linguistic style: Social and affective bases for agent personality. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[Walter 1963] W. Grey Walter. *The Living Brain*. W. W. Norton, 1963.

[Weyhrauch 1997] Peter Weyhrauch. *Guiding Interactive Drama*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1997.

[Witkin and Kass 1988] Andrew Witkin and Michael Kass. Spacetime constraints. *Computer Graphics*, 22:159–168, 1988. Proceedings of SIGGRAPH '88.