

Programming with Agents: New metaphors for thinking about computation

Michael David Travers

Bachelor of Science; Massachusetts Institute of Technology, 1986

Master of Science in Visual Studies; Massachusetts Institute of Technology, 1988

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at the
Massachusetts Institute of Technology

June 1996

© Massachusetts Institute of Technology, 1996
All Rights Reserved

Author:

Program in Media Arts and Sciences
May 3, 1996

Certified by:

Marvin Minsky
Professor of Electrical Engineering & Computer Science
Toshiba Professor of Media Arts & Sciences
Thesis Supervisor

Certified by:

Mitchel Resnick
Assistant Professor of Media Arts & Sciences
Fukutake Career Development Professor of Research in Education
Thesis Supervisor

Accepted by:

Stephen A. Benton
Chair, Departmental Committee on Graduate Studies
Program in Media Arts and Sciences

Programming with Agents: New metaphors for thinking about computation

Michael David Travers

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning on May 3, 1996
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at the
Massachusetts Institute of Technology

Abstract

Computer programming environments for learning should make it easy to create worlds of responsive and autonomous objects, such as video games or simulations of animal behavior. But building such worlds remains difficult, partly because the models and metaphors underlying traditional programming languages are not particularly suited to the task. This dissertation investigates new metaphors, environments, and languages that make possible new ways to create programs -- and, more broadly, new ways to think about programs. In particular, it introduces the idea of programming with "agents" as a means to help people create worlds involving responsive, interacting objects. In this context, an agent is a simple mechanism intended to be understood through anthropomorphic metaphors and endowed with certain lifelike properties such as autonomy, purposefulness, and emotional state. Complex behavior is achieved by combining simple agents into more complex structures. While the agent metaphor enables new ways of thinking about programming, it also raises new problems such as inter-agent conflict and new tasks such as making the activity of a complex society of agents understandable to the user. To explore these ideas, a visual programming environment called LiveWorld has been developed that supports the creation of agent-based models, along with a series of agent languages that operate in this world.

Thesis Supervisors:

Marvin Minsky

Professor of Electrical Engineering &
Computer Science

Toshiba Professor of Media Arts & Sciences

Mitchel Resnick

Assistant Professor of Media Arts & Sciences

Fukutake Career Development Professor of
Research in Education

Doctoral dissertation committee

Thesis Advisor:

Marvin Minsky
Professor of Electrical Engineering & Computer Science
Toshiba Professor of Media Arts & Sciences

Thesis Advisor

Mitchel Resnick
Assistant Professor of Media Arts & Sciences
Fukutake Career Development Professor of Research in Education

Thesis Reader:

Pattie Maes
Associate Professor of Media Technology
Sony Corporation Career Development Professor of Media Arts & Sciences

Thesis Reader:

Alan Kay
Apple Fellow
Apple Computer

Acknowledgments

The ideas in this thesis were shaped by a lifetime of learning from other people and other work. There is room here to only thank some of them:

Marvin Minsky has been my advisor and *agent provocateur* for a long time. His unique way of seeing the world constantly kept me on my toes. Mitch Resnick was my other advisor and provided invaluable assistance in actually getting this work focused and finished. Pattie Maes and Whitman Richards also provided a great deal of good advice and encouragement.

Alan Kay got me started on this path of inquiry when he began the Vivarium project at MIT and Apple in 1986. Since then he has provided consistent support and inspiration for my research over the years. Ann Marion's vision was at the nucleus of this project, and she and the rest of the people involved at Apple's Learning Concepts Group get my heartfelt thanks for thinking in nonstandard directions and encouraging others to do the same.

The Narrative Intelligence reading group provided a suitably underground context in which subversive ideas could flourish. While many wonderful people participated, I have particularly benefited from the conversation and friendship of Amy Bruckman, Marc Davis, Warren Sack, and Carol Strohecker.

The original Vivarium group at the Media Lab provided the initial growth medium for this work. The lifeforms making up the ecology included Margaret Minsky, Steve Strassman, David Levitt, Allison Druin, Bill Coderre, Silas the dog, Noobie the chimera, numerous fishes, and the autonomous blimp. More recently the people of the Epistemology and Learning Group have made me feel at home and helped me through the home stretch.

Special thanks go to those whose work and advice have had a great deal of influence on my own ideas, including Edith Ackermann, Henry Lieberman, Hal Abelson, Seymour Papert, Andy diSessa, and Phil Agre.

LiveWorld, like any large piece of software, was built on the shoulders of other programmers. In this case the giants include Ken Haase, Alan Ruttenberg, and the Macintosh Common Lisp team in its various incarnations at Coral, Apple Computer, and Digitool.

Thanks also go to all who helped me out by reading drafts and giving me feedback and encouragement, including Paul Pangaro, Amy Bruckman, David Mankins, Linda Hershenson, Marcio Marchini, and others.

Deepest thanks go to my friends who helped to keep me together and push me along through this process: David Mankins and John Redford, who have been my friends and sometime housemates for more years than any of us would like to admit; Alan Ruttenberg, my partner in hacking, resistance, and exile; and Amy Bruckman, who raises common sense to an art form.

Thanks to those organizations who provided financial support for my work, including Apple Computer, Toshiba, and the Mitsubishi Electric Research Laboratory.

Last and most, thanks to Linda, Tanya & Forthcoming. A family—what a concept.

Table of Contents

1 Introduction 15

1.1	LiveWorld: an Animate Programming Environment	18
1.2	Examples of Animate Systems	20
1.2.1	Video Games and Animation	20
1.2.2	Animal Behavior Models	20
1.2.3	Graphic Layout and Interactive Constraints	20
1.2.4	Blocks World	21
1.2.5	Putting it All Together	21
1.3	Influences and Inspirations	22
1.3.1	Society of Mind	22
1.3.2	Cybernetics, Ethology, Situated Action	23
1.3.3	Constructivism and Constructionism	24
1.3.4	Interactive Construction Environments	25
1.4	Overview of the Thesis	26
1.4.1	Analysis of Metaphors for Programming	26
1.4.2	Agents and Animacy	26
1.4.3	The LiveWorld Programming Environment	27
1.4.4	Agent-Based Programming	27

2 Metaphors and Models for Computation 29

2.1	Theories of Metaphor	30
2.1.1	The Contemporary Theory of Metaphor	31
2.1.1.1	The Conduit Metaphor	31
2.1.2	Dead Metaphors	34
2.1.3	The Metaphorical Nature of Scientific Understanding	35
2.1.4	Formalization and Metaphor	37
2.2	Metaphors in Programming	39
2.2.1	The Idea of Computation	39
2.2.2	Metaphors Make Computation Tangible	40
2.2.3	Metaphoric Models for Computation	41
2.2.3.1	The Imperative Model	42
2.2.3.2	The Functional Model	43
2.2.3.3	The Procedural Model	47
2.2.3.4	The Object Model	48
2.2.3.5	The Constraint Model	50

2.2.4	Interface Metaphors	53
2.3	Conclusion	54
3	Animacy and Agents	57
3.1	Introduction	57
3.2	The Realm of Animacy	58
3.2.1	The Perception of Causality	58
3.2.2	The Development of Animism as a Category	60
3.2.3	Frameworks of Understanding	62
3.2.4	Animacy and the Representation of Action	63
3.2.5	Conclusion: the Nature of Animacy	65
3.3	Animacy and Computation	66
3.3.1	Animism at the Origins of Computation	68
3.3.2	Animacy in Programming	69
3.3.2.1	The Little-Person Metaphor	71
3.3.3	Body- and Ego-Syntonic Metaphors	72
3.3.4	Anthropomorphism in the Interface	73
3.3.5	Animate Metaphors in Artificial Intelligence	75
3.3.6	Conclusion: Computation Relies on Animate Metaphors	77
3.4	Agent-Based Programming Paradigms	77
3.4.1	Principles of Agent-Based Programming	78
3.4.1.1	Purpose, Goals, and Conflict	79
3.4.1.2	Autonomy	80
3.4.1.3	Reactivity	81
3.4.2	Computational Realizations of Agents	81
3.4.2.1	Agent as Process	81
3.4.2.2	Agent as Rule	82
3.4.2.3	Agent as Enhanced Object	82
3.4.2.4	Agent as Slot and Value-Producer	83
3.4.2.5	Agent as Behavioral Controller	83
3.4.3	Agents and Narrative	84
3.5	Conclusion	86
4	LiveWorld	89
4.1	Overview of LiveWorld	89
4.2	Design	90
4.2.1	General Goals	90
4.2.2	A World of Lively Objects	90

4.2.3	Spatial Metaphor and Direct Manipulation	91
4.2.4	Prototype-based Object-oriented Programming	92
4.2.5	Improvisational Programming	93
4.2.6	Parsimony	95
4.2.7	Metacircularity	95
4.2.8	Graphic Realism and Liveness	95
4.2.9	Learning Path	96
4.2.10	Rich Starting Environment	97
4.3	Box Basics	98
4.3.1	Boxes Form a Hierarchical Namespace	98
4.3.2	Inheritance, Prototypes and Cloning	98
4.3.3	The Basic Box Display	99
4.3.4	Theatrical Metaphor	100
4.4	Interface Details	101
4.4.1	Selection	101
4.4.2	Menus	102
4.4.3	Mouse Operations	103
4.4.4	Cloning	105
4.4.5	Inheritance	105
4.4.6	Interestingness	106
4.4.7	Box Sizes and Positions	106
4.5	Language Extensions	107
4.5.1	Accessors for Boxes	107
4.5.2	Message-Passing with ask	107
4.5.3	Methods	108
4.5.4	Relative Box Reference	110
4.5.4.1	Self	110
4.5.4.2	Boxpaths	110
4.5.5	Demons	111
4.5.6	Global Object	112
4.6	Specialized Objects	112
4.6.1	Animas and Agents	112
4.6.2	Specialized Slots	113
4.6.3	Computed Slots	113
4.6.4	Sensors	115
4.6.5	Multimedia Objects	116
4.6.6	K-lines	117
4.7	Miscellaneous Issues	117
4.7.1	Shallow vs. Deep Cloning	117
4.7.2	Internal Frames	118

4.7.3	Deleting Boxes	119
4.7.4	Install and Deinstall Protocol	120
4.7.5	Error Handling	120
4.8	Some Unresolved Issues	121
4.8.1	Multiple Views of Actors	121
4.8.2	Uniform Interfaces Have a Downside	122
4.8.3	Cloning and Dependent Objects	122
4.9	Relations to Other Work	123
4.9.1	Boxer and Logo	123
4.9.2	Self	124
4.9.3	Alternate Reality Kit	124
4.9.4	Rehearsal World	125
4.9.5	Ágora	125
4.6.6	IntelligentPad	
4.10	Conclusion	126

5 Programming with Agents 127

5.1	Simple Agent Architectures	127
5.1.1	Simple Agents	127
5.1.1.1	Simulating Concurrency	128
5.1.1.2	Handling Conflict	129
5.1.1.3	Presenting Conflict Situations to the User	132
5.1.1.4	Behavior Libraries	133
5.1.2	Goal Agents	134
5.1.3	A Comparison: Teleo-Reactive Programming	136
5.2	Dynamic Agents	139
5.2.1	Overview	141
5.2.2	Structures	143
5.2.2.1	Tasks	143
5.2.2.2	Agents	144
5.2.2.3	Templates	145
5.2.3	Control	148
5.2.3.1	Activation and Expansion	148
5.2.3.2	Cycles	148
5.2.3.3	Success and Failure	149
5.2.4	Special Tasks	150
5.2.4.1	Combining Tasks	150
5.2.4.2	Control Tasks	151
5.2.4.3	Primitive Tasks	151

5.2.4.4	Internal Special Tasks	152
5.2.5	Domain Tasks and Templates: Examples	152
5.2.5.1	Numerical Constraints	153
5.2.5.2	Geometry	154
5.2.5.3	Behavior	156
5.2.6	Conflict	156
5.2.6.1	Slot Conflict	156
5.2.6.2	Goal Conflict	157
5.2.7	Determination	157
5.3	Interface	159
5.3.1	Top-Level Tasks	159
5.3.2	Auto-tasks and Locks	159
5.3.3	The Agent Display	160
5.3.4	Controlling When and How Agents Run	160
5.3.5	Agent Displays and Storyboards	162
5.4	Additional Examples of Agent Systems	166
5.4.1	A Video Game	167
5.4.2	A Physical Simulation	168
5.4.3	A More Complex Graphic Constraint Problem	169
5.4.4	A Creature in Conflict	170
5.4.5	An Ant	171
5.4.6	A Recursive Function	173
5.4.7	A Scripted Animation	174
5.4.8	BUILDER in the Blocks World	175
5.5	Discussion	176
5.5.1	DA as a Procedural Language	177
5.5.2	DA as a Behavioral Control System	178
5.5.3	DA as a Constraint System	179
5.5.3.1	Related Work	181
5.5.4	DA and Anthropomorphism	181
5.5.4.1	Creation of Agents	182
5.5.4.2	Agents, Tasks, and Templates	182
5.5.4.3	Are Agents Too Low-level?	183
5.5.4.4	Variants of Anthropomorphic Mapping	184
5.6	Summary	184
6	Conclusions	187
6.1	Summary and Contributions	187
6.2	Related Work	188

6.2.1.	KidSim	188
6.2.2.	Agentsheets	189
6.2.3.	ToonTalk	189
6.3	Directions for further research	190
6.3.1	What Can Novices Do with Agents?	190
6.3.2	Can Agent Activity Be Made More Understandable to the User?	191
6.3.3	Can Agent Systems Be Made More Powerful?	192
6.3.4	Can Agents be Organized in Different Ways?	192
6.3.5	Can the Environment Be Agents All the Way Down?	194
6.3.6	Agents in Shared Worlds	195
6.4	Last Word	195

Bibliography

197

Chapter 1

Introduction

*We propose to teach AI to children so that they, too,
can think more concretely about mental processes.*

— Seymour Papert (*Papert 1980*)

The computer is a new medium for thought and expression, radically different from traditional media in its dynamism, interactivity, and flexibility. A universal device, the computer can be used to create dynamic interactive models of any conceivable process, mathematical, biological, or wholly imaginary. If we learn the world by constructing it, now we have at hand a medium that enables world-building as an everyday learning activity. Theories about how the world works—say, the laws of physics, or the behavioral patterns of animals—will no longer be mere dry abstractions but instead form the basis for concrete, visible simulations that can be observed, tinkered with, and inhabited. Computational environments that permit this sort of activity ought to be extraordinarily powerful mental tools, with the potential to transform how we think and learn.

But expressing ideas in this new medium is difficult, and its potential as a learning tool still largely unrealized. The fundamental skill needed to express dynamic ideas in the interactive medium is *programming*—the ability to tell the computer what to do and how to do it. What you can express (and, more subtly, what you can conceive of wanting to express) depends upon the tools available. The problem is that current programming tools are rather limited in what they offer in the way of expressive capabilities. While the computer *can* be made to do just about anything, given enough time and expertise, what can be done *readily* depends upon the languages and tools offered by the programming environment. This is especially true for young or novice programmers, who have a limited ability to build up their own abstractions and tools, and so must depend on those that the environment already supplies.

Computers were first developed to solve mathematical problems and still bear the marks of their history. Certain modes of thought and ways of structuring activity are woven into the way we think about computers, modes which are not necessarily the only or best ways of coming to grips with the potential of this new domain. The name “computer” itself reflects this—computation is only a part of what computers do now, and they might better be called “information manipulators” or “dynamic media machines”. Indeed, at the consumer level, this is what computers have become. People are now accustomed to devices that while labeled computers are really video games or virtual reality engines or simulations of cities.

But programming, which is the only level of use in which the full power of computation as an intellectual tool can be realized, is still centered around traditional models of computation. At this level, a computer is seen as a device for performing mathematical calculations, or executing a sequence of rote operations—not as a dynamic world in which lifelike activity can take place. While the computer itself has gone through radical transformations as it penetrates into society at large, the languages, tools, and concepts used to program them have stayed pretty much the same.

As a result, a young programmer who might imagine building a dynamic world of interacting objects (say, a video game or a simulation of an ant colony), will be faced with numerous obstacles that stand in the way of bringing the vision into reality. Some of these will be inherent in the complexity of the task, and may not be possible to eliminate, but others will be due to the lack of proper expressive tools. For instance, few extant programming environments for novices support even so basic a facility as the ability to create multiple graphic objects that can move simultaneously and independently while interacting with each other. More fundamentally, the languages that the environments provide are often not suited to such tasks, because they rely on models of computation that are not particularly suited to the control of dynamic behavior.

I have coined the term “animate system” to describe these sorts of dynamic worlds that involve multiple active and interactive objects. Animate systems are simulated dynamic worlds that contain multiple independent but interacting graphic actors. I chose the term because its connotations include animals, animation, and the mind or soul (or *anima*). This thesis describes a search for a programming paradigm suitable to the control of behavior in animate systems.

The languages used for programming are defined, enabled, and limited by their underlying models and metaphors. While computer science strives towards formal definitions of its subject matter, the practical task of understanding computational entities relies on the informal technique of borrowing terminology and structure from more familiar domains via metaphoric mappings. Metaphorically-structured models are so pervasive that sometimes they are hard to see. Consider the notion of a computational “object”, an understanding of some structure inside the computer that relies, in subtle ways, upon our existing knowledge of physical objects. Two examples of more obvious metaphors are the program-as-a-recipe metaphor that is often used to convey to beginning programmers the idea that the computer is following a sequential list of instructions, and the spreadsheet metaphor that allows business people to fit a form of functional programming into a familiar medium. These metaphors provide powerful frameworks for the understanding and construction of complex systems.

Unfortunately, many if not most real-world tasks do not easily fit into the conceptual frameworks supplied by these common metaphors. This is true both for the tasks faced by professional programmers (who more often than not are faced with the task of designing a program that will function as part of a larger mechanical system, and thus must spend most of its time reacting to and controlling events in a world outside the formal domain of the programming language) and the programs that children would like to create if only they could (for instance, video games with interacting objects). Few languages make tasks like these easy, because the most important aspects of the task—such as multiple autonomous objects, reactivity, and goal-directed behavior—have no direct representation in the underlying metaphor. This is not to say that such systems cannot be built using existing tools, only that building them requires a good deal of mental contortion that may be beyond the abilities of novice programmers.

So we seek new models and metaphors for computation that can enable novice programmers to build dynamic models of behavior. If our interest is in supporting animate systems, the thought that the programs to build such systems can be based on metaphors of life and motion is particularly attractive. Anthropomorphic metaphors are used for teaching

novices how to visualize the operation of a computer, and they are common in the informal discourse of programmers. But such usages are restricted to the margins of the field, and are considered somewhat disreputable. Is it possible or desirable to bring the metaphor of a program as a living thing back into the foreground, and to make greater use of it? What qualities of programming will change if we try to look at it in animate terms?

There are many aspects of the animate domain that can be usefully mapped onto computational activity. The human ability to sequentially follow simple instructions was used as the basis for the Turing's theoretical machine. More recently, the object-oriented programming paradigm makes use of the metaphor of "message-passing" as the basis for structuring the activity of programs that are divided up into a collection of communicating objects. As we shall see, a degree of animism, explicit or implicit, is present in almost all ways of organizing computational activity.

Agent-based programming is my term for programming languages and environments that are explicitly grounded in animate metaphors. An *agent*, as we shall use the term, is any component of a program or system that is designed to be seen as animate. The term "agent" suggests a variety of attributes that have not generally been built into the underlying metaphors of programming; attributes such as purposefulness, autonomy, and the ability to react to outside stimuli. In other words, agent-based programming builds upon and extends the implicit animism of computation. Instead of the standard metaphor of the computer as an animate yet mechanical instruction follower, we substitute the metaphor of the agent that can initiate action autonomously in order to achieve a goal. Collections of agents work together to create complex behavior. The general idea of an agent as a component of a larger system is inspired by Marvin Minsky's Society of Mind theory (Minsky 1987), although the purposes to which agent-based programming puts them is different.

Thinking of program components in animate terms suggests a variety of new techniques for describing program activity to a user. If an agent has an explicit goal, the state of the agent can now be made available to the user in a meaningful way, through anthropomorphic interfaces. Goals and satisfaction provide the conceptual tools for thinking about inter-agent conflict, an important issue in a system with distributed control. A world of autonomous agents pursuing goals and coming into conflict with one another suggests that program activity can be represented in the form of narrative, which traditionally deals with motivated actors, their efforts to realize their goals, and their successes, failures, and conflicts.

Agent-based programming is an attempt to design tools and find new conceptual foundations for programming that are more suitable to the task of constructing animate systems. Programming has been said to provide us with an entirely new way of thinking, sometimes called *procedural epistemology*—"the structure of knowledge from an imperative point of view" (Abelson and Sussman 1985). One goal of this thesis is to provide a friendly critique of the existing forms of procedural epistemology, and an attempt to improve upon them. Call it an investigation into *animate epistemology*—an examination of the way we think about the animate world, how computation makes use of this way of thinking, how it fails to, and how it might do better.

1.1 LiveWorld: an Animate Programming Environment

Any programming environment designed to support the construction of animate systems has to offer some basic world-modeling capabilities, and offer them in a readily accessible form. It must support the simultaneous animation of multiple objects. It must support object autonomy; in that objects can be seen as operating under their own control rather than under the control of a central program. The objects must be able to sense their environment, which will consist mostly of other animate objects.

LiveWorld (see figure 1.1) is a programming environment designed to support the construction of animate systems. While LiveWorld itself is not based on agents, it is a tool that facilitated the development of the agent-based programming techniques, and as its name implies, was designed to support a feeling of “liveness”, a quality difficult to define but a crucial part of establishing the context for agent-based programming. A live environment is one in which there are a lot of simultaneous activities going on, both at the direct command of the user and as a result of autonomous agent activity. Users can interact with autonomous objects, and change their agents while they are running. The feel of LiveWorld sets the stage for agents, by providing a world in which they can act.

One goal of LiveWorld is to provide a world where computational objects of all kinds can be readily manipulated. Animate systems are built out of a diverse set of components, and it is important to have relatively easy-to-use tools that support creation and combination of a variety of object types. LiveWorld is built around a prototype-inheritance object system which allows objects to be easily cloned (copied) and modified. Objects are represented as hierarchically nested boxes, allowing structures of varying degrees of complexity to be manipulated as single units. Complex objects can be assembled by cloning simpler objects and dropping them into a container object: for instance, a sensor or behavior can be added to an animal body in this way. The system is intended to be a construction kit for animate systems, allowing simulations to be built out of components from libraries.

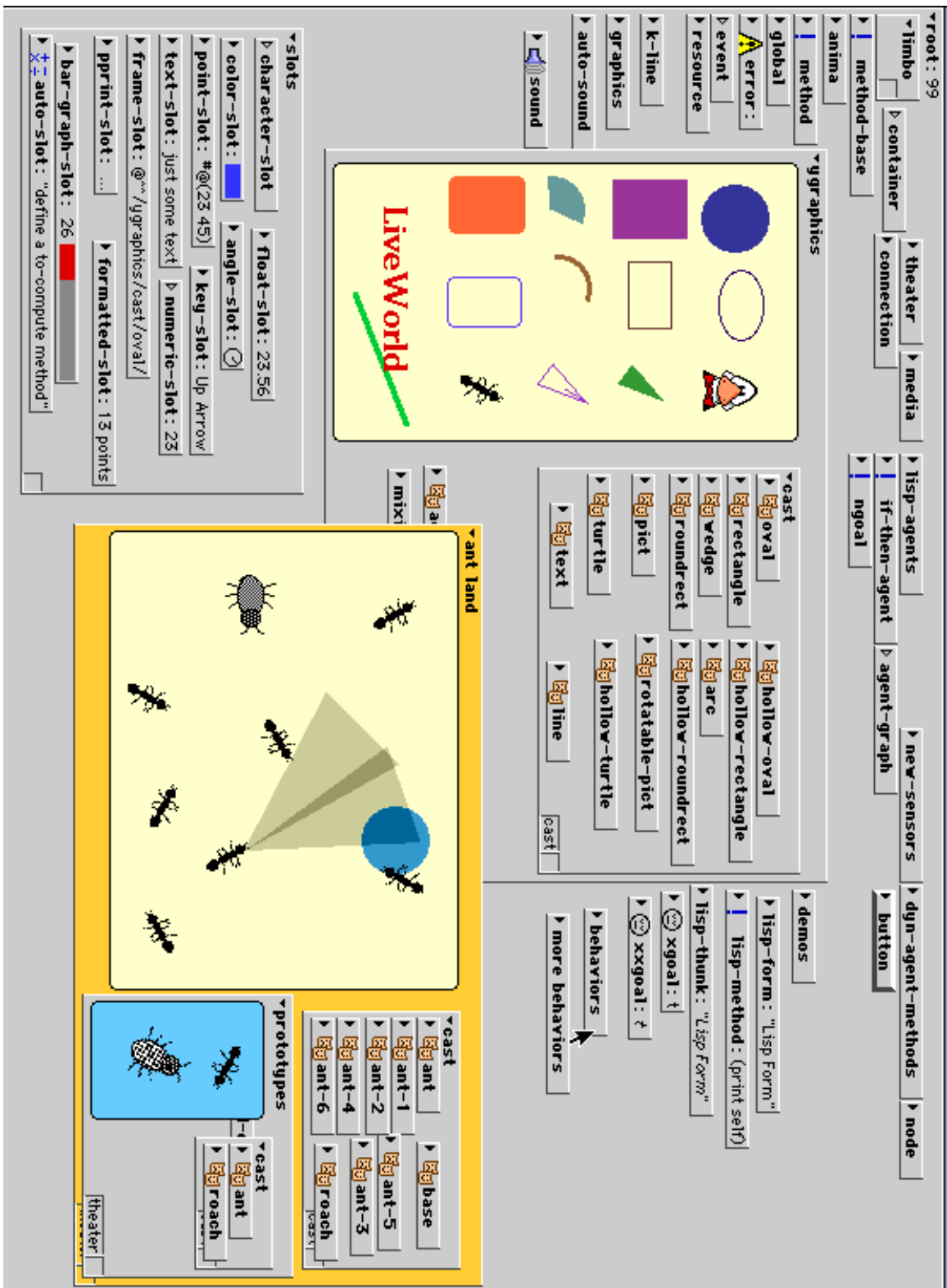


Figure 1.1: The LiveWorld environment.

1.2 Examples of Animate Systems

The idea of animate systems has its origins in the Vivarium project, initiated by Alan Kay with the mission of developing environments for simulating animal behavior. My work on this project led to a number of systems for modeling animal behavior in software. One goal of the present work is to extend some of the ideas developed for simulating animals into new application domains. Each of these domains involves actors animated by agents, but each generates a somewhat different intuitive idea about what an agent is and how they should work together. My own implicit purpose in designing LiveWorld's agent systems was to see if I could find an agent-based programming framework that could subsume and unify these divergent notions of agent.

1.2.1 Video Games and Animation

These tasks provide basic tests of the interactive graphic environment. None of them require very complicated agent systems. Video games require that objects be able to sense each other and trigger actions appropriately; this can be done by relatively simple forms of agents. Objects must be able to be created and deleted, and the user must be able to interact with objects while they are active. Simple video games don't require goal-driven agents, although more complex ones might include characters that pursue goals, which brings them into the realm of animal behavior.

1.2.2 Animal Behavior Models

Tinbergen's drive-centered model of animal behavior (Tinbergen 1951), in which a hierarchical network of drives constituted the control system of an animal, was one of the ancestors of the Society of Mind theory and provided a basis for LiveWorld's predecessor, the Agar animal behavior simulation environment (Travers 1988). LiveWorld's agent systems provide a similar level of functionality in a more flexible, general and coherent framework.

In animal behavior models, agents correspond to basic drives, both general and specific. Some examples of animal tasks are: "survive", "find-food", "orient-away-from-predator" or "run". Drives are at best temporarily satisfied and are always in some degree of conflict. That is, an animal might have multiple top-level drives such as eating, cleaning, and mating, all of which are to some degree unsatisfied at any one time, and they cannot in general be pursued simultaneously. Even the top-level goals of survival and reproduction, dictated by evolution, will be in tension with each other. The satisfaction condition or goal of a drive cannot be easily modeled as a Boolean predicate, since it may be satisfied to various degrees (i.e. there is a continuum of levels of hunger-satisfaction between starving and satiated). Animal goals will usually not be satisfied by a single operation, but will require an iterated effort to achieve a final state. For instance, the goal "be at the watering hole" will require a series of movement operations to be achieved.

1.2.3 Graphic Layout and Interactive Constraints

Another domain usefully viewed in animate terms is that of graphic layout problems, or more broadly the task of maintaining relationships among graphic objects under change. Interactive constraint systems have been applied to such problems in the past. As discussed in section 2.2.3.5, constraints may be usefully viewed as an amalgam of declarative and procedural information: a constraint both states a condition that should be held true, and contains procedures for changing the world to make it true. This provides yet another model for agents. One goal of the LiveWorld agent system, therefore, is to show that agent-based programming is a useful technique for building interactive graphic constraint systems. LiveWorld is particularly suited to exploration in this domain because of its integration of graphic objects and agents.

Agents in constraint problems take a different form than those in animal behavior domains. In a constraint problem, goals are expected to be satisfied simultaneously rather than sequentially. Goals are expressions of allowable final states rather than ongoing needs. Even if constraints are implemented as agents, the role of action is less dominant—whereas the actions of animals are necessarily limited by their physical constraints, the actions of a constraint agent can be essentially arbitrary.

The ability to solve graphic layout problems opens up the potential for LiveWorld to define its own interface behavior using agents. For instance, the constraint that a box be big enough to display its annotations could be realized by means of appropriate agents and goals, rather than special purpose internal code. This sort of “interface metacircularity” could serve to make a more integrated and flexible environment (see 4.2.7).

1.2.4 Blocks World

The blocks world is a classic artificial intelligence microworld domain, and one used in Society of Mind to illustrate agent operations. While little AI is required to implement these examples, they do require one capability that has been absent from some of the earlier efforts at agent-based programming. In particular, tower-building requires the sequential achievement of goals, a requirement which is not found in constraint problems nor in most simple animal behavior problems. The precursors to LiveWorld’s agent systems, such as Agar (Travers 1988) and Playground (Fenton and Beck 1989), did not have any real support for sequencing, which led to some frustration. The blocks world provides a test of the ability of the agent system to handle sequential tasks that require control state.

1.2.5 Putting it All Together

The problem domains mentioned pose a diverse set of tasks, but the real challenge is to develop an agent-based programming system that allows simple solutions to the problems using a single paradigm. To do so, we have to extract the common elements from these diverse problems and use them as the basis for the design of the system. The common element we seek is, of course, the *agent*. All the problems above suggest solutions expressed as collections of simple, cooperating, task-based, goal-directed modules. The challenge is to design a single agent system that can express these solutions.

1.3 Influences and Inspirations

A number of powerful ideas have informed this research and the designs that grew out of it. This section sketches out this background of ideas and situates the present work in relation to them.

1.3.1 Society of Mind

The concept of agent used here derives from Marvin Minsky's Society of Mind (SOM) theory (Minsky 1987). SOM pictures a mind as a collection of interacting agents, which are defined operationally as "any part or process of the mind that by itself is simple enough to understand". The mind is thus seen as a collection of simple entities, each with their own goals but somehow connected so that their actions are coordinated. The word "agent" has obvious animate connotations. While agents are seen as mechanisms, they are mechanisms that are seen in loosely anthropomorphic terms, having a purpose and (perhaps) a degree of autonomy.

Agents serve many different purposes: some have representational or perceptual functions (such as **apple** or **red**), others are procedural (**build**, **find-place**). Still others are based on a wide variety of AI methodologies including neural nets, frame-based representations, and GPS-style goal followers. Because agent is defined so loosely, and there are many different kinds, it is probably unwise to say much about agents in general. For our purposes, procedural agents are of the greatest interest. These, more so than others, are described in highly animate terms: "Your **grasping** agents want to keep hold"; "**add** must call for other agents' help" (SOM, p20-21). The idea of procedural agents clearly derives from the computational notion of a procedure, and like procedure invocations can be seen as hierarchical bureaucracies that perform complex tasks by parceling them out from top to bottom. But unlike procedures, they also can have desires and are otherwise anthropomorphized.

In Minsky's terminology, "agent" has a dual term: "agency". If "agent" is any understandable part of the mind, then "agency" is "any assembly of parts considered in terms of what it can accomplish as a unit, without regard to what each of its parts does by itself." With these definitions, "agent" and "agency" are to be seen as dual terms for the same thing, "agent" meaning a mechanism simple enough to be understood in mechanical terms, while "agency" indicates the same mechanism but viewed in functional or intentional terms.

I recycled the old words "agent" and "agency" because English lacks any standardized way to distinguish between viewing the activity of an "agent" or piece of machinery as a single process as seen from outside, and analyzing how that behavior functions inside the structure or "agency" that produces it (Minsky 1991).

The distinction between these two views or levels of analysis is crucial to Minsky's attempt to build a new metaphor system for describing minds. However, the metaphoric mapping is somewhat unclear. In other domains of research, "agent" has come to mean the very opposite of what Minsky intends: that is, "agent" (rather than "agency") now usually means a semi-intelligent program as seen from the outside. This terminological confusion is unfortunate. Here, we will use "agent" to mean any process or component that is intended to be understood from

an animistic perspective. What it means to understand something in this way is discussed in Chapter 3.

The animate systems that LiveWorld can build are much simpler than human minds, and incorporate only the most basic parts of SOM. For instance, none of the representational ideas are implemented, other than a weak version of K-lines (see section 4.6.6). Nonetheless SOM has functioned as a key stylistic inspiration for agent-based programming. I view LiveWorld as a step towards building implementations of SOM, if only in that the availability of agent-based programming environments for children might encourage the development of appropriate styles of thinking and programming.

1.3.2 Cybernetics, Ethology, Situated Action

The task of simulating animal behavior was the goal of my first efforts in agent-based programming. Modeling animals led to a new set of perspectives about the nature of intelligence. The intelligence of animals necessarily manifests itself in the form of behavior and interaction with their environments, rather than the ability to solve symbolic problems. At the time this project began, the main stream of artificial intelligence research had rather little to say about these issues. Instead I was inspired by works from cyberneticists, especially Valentino Braitenberg's book *Vehicles* (Braitenberg 1984), which illustrated a series of ideas about control by encapsulating them in simple creatures made up of sensors, motors, and neuron-like components. My first effort at an environment for animate systems, BrainWorks (Travers 1988), was based on this book.

Another source of inspiration was the work of ethologists, scientists who studied animal behavior in the world. Niko Tinbergen, one of the founders of this field, developed a general theory of behavioral control (Tinbergen 1951) that formed the basis of my second system, Agar (Travers 1988), which was based upon an idea of agent derived from Tinbergen's drive centers and was more oriented towards producing simulations of actual behavior.

At this time, several groups within the AI community were becoming dissatisfied with the standard approaches to controlling action and developing alternatives very much along these lines. In particular, Rod Brooks' work with robots (Brooks 1986) (Brooks 1991) and the work of Phil Agre and David Chapman (Agre and Chapman 1987) (Chapman 1991) were influences on my own work. This school of thought, which has been labeled the "situated action" movement, is still controversial within AI. This movement has generated a good deal of critical reflection within AI, which can only be to the good. From a constructive standpoint, however, the techniques of situated action have had demonstrated successes only in limited domains. It is not clear how to extend these approaches, which rely on highly specialized control mechanisms, to the kinds of general capabilities that AI has striven for. However, for the purposes of this thesis, that is enabling the construction of autonomous and reactive creatures in interactive environments, they have a clear utility.

Uniting these three strands of research is the idea that understanding intelligence requires taking into account the relationship between an intelligent creature and its environment. Intelligence, then, is located not so much in the head, but in the relationship between the mind

and the world. I've found this simple idea intriguing if not absolutely convincing. It has value as an alternative perspective on thinking about mental processes (some of the epistemological implications are discussed further in section 2.1.4). However, this approach to thinking about intelligence is quite appropriate for the task at hand, that is, designing animate environments and programming systems for them. This task requires that the environment and the minds and agents be co-designed and work together. The lesson for programming environments is that what you want your programming language to do depends on the environment in which it will operate.

1.3.3 Constructivism and Constructionism

The constructionist approach to learning (Papert 1991) underlies the design of LiveWorld and the general framework of my research. Constructionism derives its name as a variant of the kindred psychological school of *constructivism*. The constructivist view of learning and development (Piaget 1970) presents an image of these processes as both active and creative. It is active in that it is primarily a process of spontaneous development, in which the learner must discover knowledge rather than having it delivered from the outside by a teacher. It is creative in that the process of learning involves building structures of knowledge within the mind, structures which are original creations, at least relative to the learner. Constructionism shares the spirit of constructivism, but extends it by emphasizing that learning can be aided by building actual material objects in the world. The process of construction then takes place in both the mind and the external shared world. Both approaches to learning encourage the learner to assume an active and creative role rather than be a passive recipient of information, and both are opposed to the more prevalent educational philosophy which they label *instructionism*, in which learning is seen as a process in which students passively receive information from teachers.¹

Constructionism applied to the use of computers in education results in the idea that students should be able to use the computer as a construction medium. The implications are that the computational medium should be flexible: like a piece of paper, it ought to support a number of different modes of use, levels of skill, and target domains. It also implies that the locus of control should be the student rather than the computer, as opposed to the more instructionist versions of educational computer use, in which the model is for the computer to present the student with a structured series of lessons and quizzes.

The constructionist philosophy underlies the development of Logo, Smalltalk, Boxer and related programming environments for learning. The LiveWorld environment too is constructionist in orientation—it provides a computational medium that makes a variety of computational objects available to novice programmers, and encourages experimentation and improvisation. One important aspect of the design is to make all aspects of LiveWorld's operation and all relevant computational objects accessible to the user through a direct-

¹ See section 2.1.1.1. The distinction between instructionism and constructionism directly parallels the distinction between the conduit and toolmaker images of language use.

manipulation interface. This also allows some elements of the construction process to take place through spatial manipulation of parts. Beginning programmers in LiveWorld can build systems simply by copying, dragging, and combining parts via direct manipulation.

Constructionism has sometimes been overinterpreted to mean that the computational environment should initially be a completely blank slate, empty so that it eventually be populated solely by the creations of the user. LiveWorld takes a looser approach to constructionism, providing kits of parts and examples that can be copied and modified. Indeed its prototype-based object system emphasizes this mode of use (see section 4.2.10). This form of constructionism more realistically reflects how real-world learning takes place, and makes it possible for students to tinker with complex systems that they could not easily build from scratch.

A constructivist philosophy also underlies the study of metaphors for computation presented in Chapters 2 and 3. This method of thinking about how people think about computation is based upon the view that the knowledge of both experts and novices is individually constructed, with metaphor being a fundamental construction technique. The fact that our models of computation are constructed leads to the possibility that there could be different ways of understanding computing that would provide powerful alternative points of view. This premise underlies the attempt to construct an approach to programming centered around the idea of agents. By explicitly designing a language around a powerful metaphor, the activity of programs can be viewed from a new and useful perspective.

Constructivism and constructionism are radical ideas. Constructivism forces us to acknowledge that learning is a creative process and that knowledge is constructed rather than received. Constructionism suggests that this process takes place not only inside the heads of individuals, but involves interaction with the material and social worlds. Both imply a great deal of freedom and the possibility of alternative epistemologies. This thesis is an attempt to explore some of these possibilities for the domain of programming.

1.3.4 Interactive Construction Environments

If constructionism is a philosophy of learning, interactive computer environments provide both the worlds and raw material where construction can take place. A few programming environments that were designed to support construction have already been mentioned. But there are other examples of interactive software that, while not providing as powerful general-purpose programming tools as Logo (Papert 1980) or Smalltalk (Goldberg and Robson 1983), succeed in creating highly interactive virtual worlds which are constructionist in their own fashion. These include interactive constraint systems such as Sketchpad (Sutherland 1963) and ThingLab (Borning 1979), spreadsheet programs (in terms of widespread adoption, the most successful variety of constructivist computational media), and some visual construction kits such as Hookup (see section 2.2.3.2).

What distinguishes these systems is their ability to create a virtual world of *reactive objects* that operate autonomously according to rules that the user can construct and control. An important goal of LiveWorld was to enable this sort of construction. The “liveness” of

LiveWorld extends not only to being able to build simulations of living animals, but to having the computational medium itself be lively and reactive.

There is now a large body of software, mostly aimed at children, that presents worlds in which limited forms of construction take place. Examples are mostly in the form of games such as SimCity, Pinball Construction Kit, and The Incredible Machine. These systems generally present a fixed set of objects that operate according to fixed rules, with the user given the ability to arrange the objects into new configurations. As such, they are not as general or flexible as the other environments mentioned, but provide inspiration due to their high degree of liveliness and their ability to provide rich and habitable worlds.

1.4 Overview of the Thesis

The body of this thesis is in four chapters. The first two are primarily conceptual and analytical, while the last two are more technical and describe implemented software.

1.4.1 Analysis of Metaphors for Programming

Chapter 2 is an analysis of how metaphor systems are used in the understanding and construction of computation and programming languages. The object of the chapter is to find a way of thinking about the informal conceptual underpinnings of a field whose knowledge is more commonly described in mathematical formalisms. It first gives a brief overview of contemporary metaphor theory, then applies the theory to the analysis of a number of different models or paradigms for programming. By means of this analysis we can reveal some of the hidden assumptions that underlie the concept of computation and the languages with which we approach it.

1.4.2 Agents and Animacy

Chapter 3 continues the task of the previous chapter by exploring a particular type of metaphor that employs some form of anthropomorphism or animism. Animacy is seen to be, in itself, a basic category for dealing with the world, with roots deep in innate perceptual processes. Three relevant characteristics of the animate realm are selected for attention, because of their centrality to the concept and because of their relevance to computation: purposefulness, reactivity, and autonomy in the sense of being able to initiate action. Animism is seen to be central to understanding action, and thus also central to understanding computational activity.

The chapter examines the use of animism in programming languages, as well as the more explicitly anthropomorphic constructs found in interface agents and artificial intelligence. We find that most programming languages involve implicit animate metaphors, but an animacy of a particularly limited sort that derives from the original conception of the computer as a device for following instructions.

At this point, it is possible to more precisely define agent-based programming as an attempt to construct a new paradigm for programming that makes computational animism explicit and attempts to extend it beyond the instruction-following metaphor to include the central properties of animism revealed by our analysis, such as autonomy and goal-directedness. Various methods for realizing an animism with these properties in computational forms are explored. Some previous attempts to construct programming systems that operate in this style are examined and critiqued.

1.4.3 The LiveWorld Programming Environment

Chapter 4 describes the LiveWorld system in more detail. LiveWorld is a visual object-oriented programming environment that supports the research into agent-based programming techniques. LiveWorld provides the necessary substrates for this work, including a flexible prototype-based object system, a simple two-dimensional world for computational actors to perform in, and a graphical interface that makes the computational objects tangible and accessible to manipulation. While LiveWorld is primarily a tool to support the main object of research, it is of some interest in its own right and a necessary step on the road to developing an environment for agent-based programming. Some of the more innovative features of LiveWorld are sensors that allow objects to be aware of each other, and the integration of a spreadsheet-like interface with animated graphic objects.

1.4.4 Agent-Based Programming

Chapter 5 presents a series of agent-based programming systems implemented using LiveWorld. The first two, Simple Agents and Goal Agents, are computationally fairly trivial but serve to illustrate some of the key concepts underlying agent-based programming. These include the idea of agents as separate objects that can be independently manipulated and integrated into a computational actor, the need to handle concurrency and conflict, the idea of using goals as a method to organize agent activity, and the use of anthropomorphic icons to convey agent state and activity to the user. The final agent system, Dynamic Agents, illustrates how these ideas can function as part of a more powerful and general programming system. A series of examples shows how the agent systems can implement systems for the selected problem domains.

Chapter 2 Metaphors and Models for Computation

It may be that universal history is the history of a handful of metaphors.

— Jorge Luis Borges, “The Fearful Sphere of Pascal” (Borges 1962)

We have defined agent-based programming as a method of thinking about computational activity that is explicitly organized around animate metaphors. This chapter and the next are attempts to expand this definition by looking closely at both metaphor and animacy and the roles they play in structuring the computational domain. We will see that computation, like almost all specialized domains of knowledge, is founded on a variety of metaphors that relate its structures to those of other domains. The domain of animate behavior is a particularly important source of concepts and vocabulary for computation, for reasons to be explored. Examining the role of metaphor and animacy in detail will both lay the groundwork for designing agent-based programming systems and provide some insight into the epistemology of computation and programming in general.

Our understanding of a subject as abstruse as programming is necessarily grounded in other things we know. A novice learning to program must construct understandings by bootstrapping them from existing bodies of knowledge and skills. Suitable bases on which to build include experientially grounded systems of knowledge about the physical and social worlds, or more abstract bodies of knowledge such as mathematics or economics. Ultimately abstract knowledge must be grounded in either innate structures or basic experience. While this process of grounding out knowledge takes many forms, the role of *metaphor*—the overt or implicit description of one thing in terms of another—is one of the most important.

The discourse of programming and programmers is heavily metaphorical, yet the role of metaphor in computer science is often ignored or even vilified. The formal presentation of computer languages tends to hide the necessarily informal methods that we use to understand them. Thus the purpose of this chapter is to lay an analytical framework for thinking about the ways in which people come to understand computers and programs, and to argue for metaphor as a legitimate and necessary tool for this process. This will provide background for the following chapter, which takes a look at one particular metaphor or way of thinking that is of particular importance to computation, that of animate or anthropomorphic metaphor.

In the present chapter, I will first outline some general theories of metaphor and particular ideas from these theories that will be useful, then move on to look at the role of metaphor in scientific discourse and in computation in particular. A number of frameworks for programming are analyzed in terms of the metaphoric structures they impose on their domain and on their users.

Learning a new field involves learning a new language, or more precisely, new ways of using language. Technical fields in particular deploy everyday terms and usages in new ways, and the fact that words with everyday meanings suddenly take on new meanings that are related but not identical to their old ones is a source of confusion for both the novice and the professional. David Pimm (Pimm 1987) refers to the peculiar mannerisms of mathematical discourse as the “mathematics register”, that is, a particular way of speaking which must be learned and relies strongly on metaphorical constructs. Computation, being a relatively new and diversified field, does not yet have a single “computational register”. By looking at some of the metaphors employed in some parts of computational discourse, I hope to understand how the discourse of computation is structured. This approach is somewhat contrarian, in the sense that it deliberately emphasizes the non-mathematical, metaphorical, and informal aspects of a field that has traditionally structured itself in terms of formal mathematics. By investigating the informal conceptual foundations of computing, which I believe in some senses are deeper than the mathematical foundations, I hope to be able to gain insights that will permit the design of new languages and environments for programming.

2.1 Theories of Metaphor

...“metaphor” refers to all those processes in which the juxtaposition either of terms or of concrete examples calls forth a network of similarities which help to determine the way in which language attaches to the world.(Kuhn 1993, p539).

The term “metaphor” is used here not in a narrow linguistic sense, but in the sense of rich and complex *metaphoric models* (Lakoff and Johnson 1980). A metaphoric model is a way to structure the knowledge of one domain (the target) by mapping onto it concepts and relations from an existing domain (the source) that is already familiar. Metaphor in this sense is not a mere linguistic device used only for the figurative embellishment of otherwise straightforward language, but a fundamental way of learning and structuring conceptual systems, a part of everyday discourse.

Metaphors are so pervasive that they are sometimes hard to see. For instance, the common model and terminology of the behavior of electrical circuits is based on a metaphoric mapping to fluid flow in pipes, with voltage mapping to water pressure, batteries to pumps, current to flow, and so forth. In a less technical and less obviously metaphorical example, the American conception of anger is metaphorically derived from the image of a heated fluid in a container, giving rise to expressions like “blowing off steam” and “containing his rage” (Lakoff and Kövecses 1987). The metaphoric structure underlying such common concepts indicates that metaphoric models are not merely optional stylistic overlays to a fundamentally objective and literal mode of representation. Rather, they are a fundamental mechanism for encoding knowledge. Much of our common cultural knowledge is in terms of metaphoric models (Lakoff and Johnson 1980).

Since metaphoric processes are a fundamental part of how knowledge is structured, there can be no hard line drawn between metaphoric thought and other kinds. From a certain point of view, all thought is metaphoric:

No two things or mental states ever are identical, so *every* psychological process must employ one means or another to induce the illusion of sameness. Every thought is to some degree a metaphor (Minsky 1987, p299).

If all thoughts are equally metaphorical, then metaphor is not a very useful analytic category. However, some thoughts seem to be more metaphorical than others. There is a large space of possibilities lying between the purely literal and the obviously metaphorical. Perhaps the most interesting cases are those in which a phrase or thought appears initially to be literal, but upon examination turns out to be based in metaphor after all. We will see that many metaphors used in the discourse of mathematics, science, and technology are like this.

It will be useful to distinguish the concept of a structuring metaphor from the closely related idea of an analogy (Winston 1982) (Gentner 1989). Both aim to establish understandings by the creation of mappings between domains, but the two terms connote different aspects of this cognitive process. Analogy usually refers to the construction of *explicit* mappings between two well-established domains, whereas metaphor is more often implicit. The linguistic form of analogy, the simile, keeps the two domains safely separated by using terms such as *like* (“life is *like* a bowl of cherries”), while metaphor draws a more immediate connection (“life *is* a journey”). More importantly, metaphor often plays a foundational role in the establishment of new, previously unstructured domains. The two terms really connote different views of the same broad cognitive process, with the view through metaphor being more oriented towards the foundations of cognition. Analogies can be powerful learning tools, but if the thing learned is to become an important part of thinking, it must become integrated into the structures of the mind—that is, it must become a structuring metaphor. One draws an analogy, but one lives in metaphor.

2.1.1 The Contemporary Theory of Metaphor

My primary tool for thinking about metaphor will be the theory put forth by George Lakoff and Mark Johnson (1980)(Lakoff and Johnson 1980) and developed further by them, Eve Sweetser (1990)(Sweetser 1990), Mark Turner (1991)(Turner 1991) and others, and more recently labeled “the contemporary theory of metaphor” (Lakoff 1993). In this theory, metaphor is to be understood as any mapping between normally separate conceptual domains. The purpose of this mapping is to structure an abstract, unfamiliar, or unstructured domain (the *target*) in terms of one that is more concrete, familiar, or structured (the *source*).

Metaphor is viewed more as a basic tool of cognition rather than a special turn of language, and most concepts are generated by metaphors. The exceptions are those concepts that are thought to be perceptual or cognitive primitives, such as *up* or *cat*. Aside from these references to concrete physical objects and experiences, metaphorical understanding is the rule. In Lakoff’s words, “metaphor is the main mechanism through which we comprehend abstract concepts and perform abstract reasoning” (Lakoff 1993). The more narrowly linguistic meaning of metaphor is called a “metaphorical expression” to distinguish it from the broader view of metaphor as a conceptual mapping.

2.1.1.1 The Conduit Metaphor

The contemporary theory has its roots in Michael Reddy’s work on what he called the Conduit Metaphor (Reddy 1993), a detailed exposition of the system of ideas underlying the concept of communication. Reddy found that there was a consistent metaphorical substrate

underlying talk about communications and ideas. This metaphor was based on the idea that ideas were like physical objects, and that the purpose of language was to package up ideas for transfer between minds. This insight was illustrated by example sentences like:

- 1) He couldn't put his thoughts *across* well.
- 2) Try to *pack* more thoughts *into* fewer words.
- 3) I *gave* him that idea.
- 4) We *tossed* the ideas *back and forth*.

The implications of this metaphor are that words function like packing crates for meanings, and that writing or speaking is a process of packing, shipping, and unpacking. Language as a whole is seen as a conduit for transferring meanings from mind to mind. The unstated implication is that meaning is unproblematically conveyed by language. It further implies that the listener is essentially a passive recipient of meanings generated by speakers.

These usages and their underlying metaphor are so commonplace that their contingent nature might be hard to see. To better illustrate this, Reddy proposed an alternative metaphor system which he called the Toolmakers Paradigm. This metaphor system posits listeners isolated in cells, who transmit and receive crude instructions in the form of blueprints rather than words between neighboring cells. To make any use of these transmissions, they must

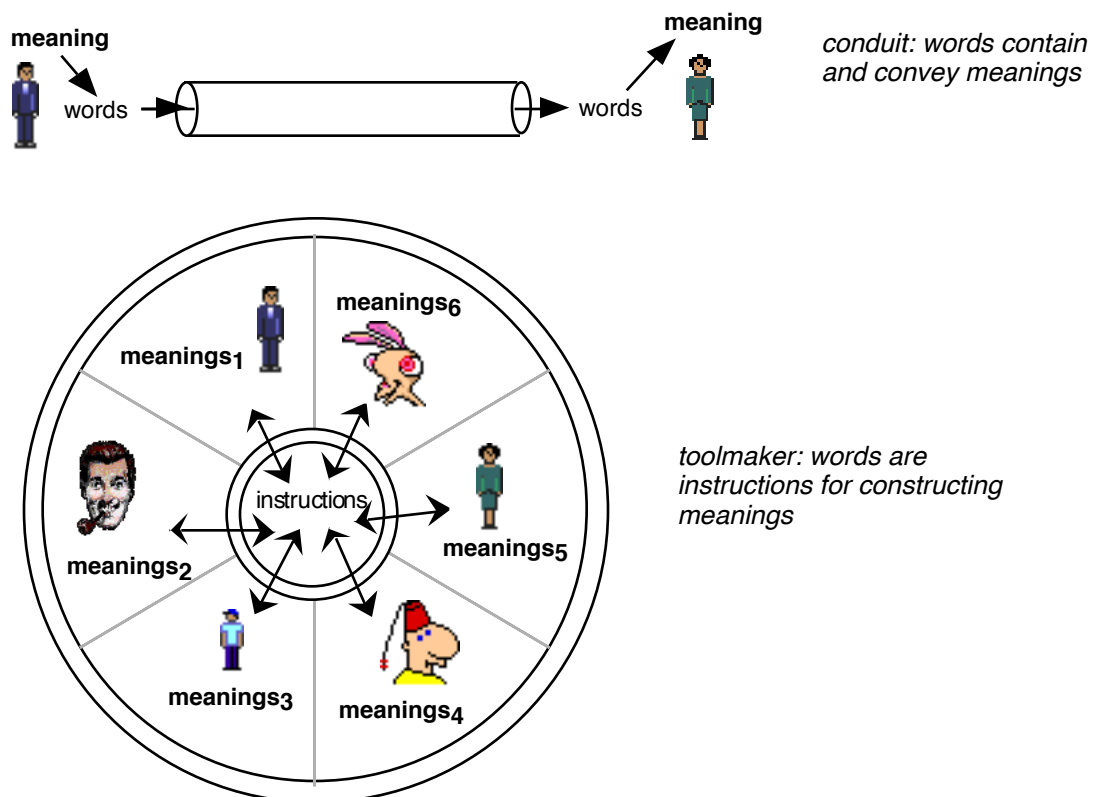


Figure 2.1: Comparing the conduit metaphor and the toolmakers paradigm (after Fig. 1 in (Reddy 1993)).

actively follow the instructions and construct the described object. Naturally transmission is imperfect in this system, and each recreation of meaning is subject to the interpretation of the hearer.

The Toolmakers Paradigm is intended to be a metaphorical representation of a constructivist theory of language, in which listeners are active constructors of meanings rather than passive recipients. The existence of an alternate metaphor is a powerful tool for thinking about alternate theories of language. In the conduit metaphor, the sharing of meaning is assumed to be the default case, and any divergence of interpretation is a problem to be explained by faults in the transmission system. By contrast, the Toolmakers Paradigm emphasizes that shared meanings are achievements, requiring careful coordination and interaction through limited channels of communication.

Although the conduit/toolmakers example is a powerful dual metaphor for thinking about constructivist theories of mind, Reddy's main concern is to illustrate the pervasiveness of the conduit metaphor and the way that it can place a strong bias on theories of language. The conduit metaphor plays a dual role in the history of the contemporary theory of metaphor. It stands as an example of a common metaphor worked out in detail, and as an illustration of how linguistic theory itself can be built on and influenced by unquestioned metaphors.

Lakoff and Johnson, inspired by Reddy's effort, embarked on a more comprehensive effort to analyze the metaphor systems underlying everyday thought. They summarize their findings by expressing the metaphors as short declarations of the mapping: LOVE IS A JOURNEY or EMOTIONS ARE SUBSTANCES. According to Lakoff:

Most people are not too surprised to discover that emotional concepts like love and anger are understood metaphorically. What is more interesting, and I think more exciting, is the realization that many of the most basic concepts in our conceptual system are also normally comprehended via metaphor—concepts like time, quantity, state, change, action, cause, purpose, means, modality, and even the concept of a category. These are concepts that enter normally into the grammars of languages, and if they are indeed metaphorical in nature, then metaphor becomes central to grammar (Lakoff 1993, p.212).

Since computer programming concerns itself with many of these same basic concepts, we should not be surprised to find that metaphors underlie the discourse of computation, and that these metaphors are variants of those found in ordinary discourse. To take one example, Lakoff's studies have revealed that the metaphorical representation of events in English involves a basically spatial metaphor, with states being represented as locations, state-change as movement, causes as forces, purposes as destinations, and so forth. This metaphor surfaces in computation through such phrases as "the process is *blocked*" or "the machine *went into* a run state". These usages are so much a part of everyday use that, like the conduit metaphor, they hardly seem like metaphors at all.

The viewpoint of the contemporary theory of metaphor leaves us with two points that will inform the rest of this analysis: first, that some of our most fundamental concepts are structured metaphorically, and second, that it is possible (as the Toolmakers Paradigm shows) to gain a new viewpoint on these concepts by proposing alternate metaphors.

2.1.2 Dead Metaphors

We've seen that some metaphorical structures are so ingrained into our habits of thought and language that they are very difficult to see as metaphors. They raise the question of whether such usages are really metaphorical in any meaningful sense. A *dead metaphor* is one that has become a conventional usage or phrase and so has (according to some theories of metaphor) lost the live mapping between its domains. "Falling in love" is one example—the phrase is so routinized that it does not recall any feelings of physical falling, although the metaphoric mapping is easy enough to recreate. "Kick the bucket" is a better example, since its live roots are even more distant².

There is wide diversity of opinion on the question of dead metaphors. According to Black (Black 1962), dead metaphors are not really metaphors at all, but should instead be considered as separate vocabulary items. Lakoff is dubious about the utility of the concept—he believes that most conventionalized phrases still retain traces of their origins in living metaphors (Lakoff and Johnson 1980, p55). Gibbs (Gibbs 1993) points out that if a metaphor was truly dead it would lose its compositional qualities, but in fact they still remain. You can understand expressions like "falling head-over-heels in love" even if you had never heard that particular variant of "falling in love". Since the mapping between domains can be reactivated to comprehend this sort of novel phrase, the metaphor lives on after all.

Metaphors used in technical discourse often appear to be dead. Since the technical meanings of phrases like "a blocked process" or "pushing a value onto the stack", are perfectly clear to experts, their metaphorical origins are often dismissed and ignored. Nonetheless the viewpoint of Lakoff and Gibbs is still useful and perhaps necessary to understanding the conceptual bases of technical understanding. In this view, even when technical terms have taken on what seems like an unproblematic formal meaning, they continue to maintain a link back to their metaphor of origin, because the mechanisms for understanding are metaphorical at their roots. Metaphors can differ in the degree to which they are taken for granted and kept out of consciousness, but are rarely so dead as to completely detach themselves from their origins.

Another extremely conventionalized metaphor used in computation is the treatment of memory as space and data as objects that are located and move within that space. The degree of conventionalization of these usages is so high that some people get quite annoyed if attention is drawn to their metaphorical underpinnings. The MEMORY IS SPACE metaphor might be considered dead since it is extremely conventionalized, but it is still alive in Lakoff's sense — the mapping between domains is still present and can be generative of new constructs, such as the slangy term "bit bucket" (the mythical space where lost bits go) or the endless stream of respectable technical terms that reflect the metaphor ("garbage collection", "partition", "allocation", "compacting", and so forth).

Perhaps *transparency* is a better metaphor than death to describe the condition of the metaphors underlying technical terminology. They do such a good job of structuring their target

² Apparently the phrase derives from the actions of dying farm animals.

domain that they seem to disappear, and the lens of metaphor becomes an invisible pane of glass. The question remains as to what makes particular metaphors achieve transparency. I speculate that metaphors become transparent when they impose a strong structure on a domain that was previously unstructured. These metaphors essentially force one to think about the target domain in their own terms, to the point where any alternative way of structuring the domain becomes forgotten and almost unthinkable. At this point, the winning metaphor is ready to be taken literally. Such metaphors have been labeled *theory-constitutive metaphors* by some philosophers of science. That is, rather than simply mapping between the concepts and vocabulary of two existing domains, as conventional metaphors do, a theory-constitutive metaphor can be said to *create* the structure of a new domain, based on the structure of an existing one.

Where one theory-constitutive metaphor is dominant (for instance, the metaphor of electricity as the flow of fluid, or of computer memory as a space), the terms that are brought to the target domain tend to become conventionalized and transparent, such as the term *current* in electrical theory. But not all theories and metaphors are as well established as these. Even memory is sometimes viewed through alternate metaphors, i.e. as a functional mapping between addresses and values. Computational discourse seems to have a particular need to mix metaphors, as we shall see.

2.1.3 The Metaphorical Nature of Scientific Understanding

The place of metaphor in the discourse of science has always been problematic. The distinction between literal and metaphorical meanings was first promulgated by Aristotle, who grudgingly acknowledged the utility of metaphor in poetry but demanded that it be eliminated from the discourse of natural science. Lloyd argues that this dichotomy was in fact necessary for the creation of a new rhetorical territory in which metaphor would be banned and literalism could flourish:

...the distinction between the literal and the metaphorical...was not just an innocent, neutral piece of logical analysis, but a weapon forged to defend a territory, repel boarders, put down rivals (Lloyd 1989, p23).

So, the domain of science was in a sense brought into being by the very act of banishing metaphor and other poetic forms of language. Scientific thought was to be of a form that dealt only with literal truth.

Despite Aristotle, metaphors are commonly employed in scientific discourse, particularly in informal and educational settings. While scientific rhetoric may aspire to the literal, it cannot avoid the need to bootstrap new theories from old concepts using metaphor. Some theories of metaphor in science relegate the use of metaphors for training to a separate category of “exegetical metaphor”, but as Kuhn points out (Kuhn 1993), every scientist must be trained and thus such metaphors are not at all marginal, but instead are a crucial part of the way in which a scientific field reproduces itself. The question then becomes whether the exegetical metaphors are like scaffolding, used to erect a formal structure in the mind but discardable when the task of construction is completed, or whether the structure maintains a live, dynamic relationship to the metaphors that allowed it to be built.

Given that metaphor is a part of everyday scientific practice, why do most scientists act as literalists, paying little or no attention to metaphor and occasionally expressing hostility to the very idea of investigating them (Gross and Levitt 1994)? The roots of scientific rhetoric's adherence to literalism may be sought in the social practices of scientists. The practice of science demands the use of a rhetoric that promotes literal rather than metaphoric construals of language. Latour (Latour 1987) paints a picture of science as a contest to establish facts, a contest that depends as much on rhetorical moves as it does on laboratories. Scientists jockey to make their statements strong, so they will be taken as facts, while simultaneously working to weaken the statements of rivals by painting them as constructions, hence questionable. The rhetorical tools for making a statement factual Latour calls *positive modalities* (i.e., a bald statement of fact) while the tools for doing the opposite are *negative modalities* (i.e., "Dr. X claims that [statement of fact]). "It is around modalities that we will find the fiercest disputes" [Latour, *op. cit.*, p25]. Here, since we are interested specifically in rhetoric rather than ongoing controversies among philosophers of science, we need not trouble ourselves over whether Latour's model of science is complete. It does lead us to speculate that the competitive pressure among scientists to establish facts will also contribute to their tendency to hide or strip the metaphors from the language. A statement that contains obvious metaphors is weaker than one that contains either no metaphors or only those so conventionalized as to be dead. Metaphor use is not exactly a modality in Latour's sense, but it can be seen that similar dynamics might apply and tend to either strip metaphor out of scientific discourse, or disguise it as something else.

However, not all science is sufficiently developed that it can maintain the pretense of literalness. Metaphors are commonly used to introduce vocabulary and basic models into scientific fields: "their function is a sort of *catachresis*—that is, they are used to introduce theoretical terminology where none previously existed." (Boyd 1993). The term catachresis was introduced by Max Black in his influential early work on the interaction theory metaphor (Black 1962). The interaction view posited a dynamic interaction between elements of the two linked domains. But Black did not believe that metaphors used in science were still interactive, since the meanings of the scientific terms were fixed, and that metaphoric vocabulary creation was *mere* catachresis, rather than a proper metaphor. Boyd disagrees, holding instead that scientific use of metaphor does double duty—it creates vocabulary to describe a new domain, and at the same time makes this new domain interact with the other domain involved in the metaphor.

Boyd terms metaphors that are essential in science *theory-constitutive* metaphors. He distinguishes these from metaphors used solely for pedagogic purposes, although these might have been more important earlier in the science's history. A good theory-constitutive metaphor is a tool that lets a scientist do his job of "accommodating language to the causal structure of the world" or "carving the world at its joints."³ His primary example of a theory-constitutive metaphor is the use of computation as a foundational metaphor for cognitive psychology.

³A metaphor introduced by Plato in *Phaedrus*.

The use of metaphor in theory formation and change depends upon this open-endedness, especially in young fields. However, the metaphor persists even as the scientific theory matures and the particular points of analogy become explicit. Sometimes complete explication is impossible, but this is not an indication that metaphor is too imprecise to serve as the basis of scientific theorizing. Rather, it means that metaphors are tools among other tools that scientists use to achieve their goals. Metaphoric interpretation remains open-ended as long as scientific theories remain incomplete.

2.1.4 Formalization and Metaphor

Computer science contains a strong ideological bias against the recognition of metaphor. Its emphasis on formalism might almost be seen as a technology for making metaphors seem as dead as possible. Formalists naturally oppose metaphor as a particularly insidious form of non-rigorous thinking. In computer science, Edsger Dijkstra has made his opinions of metaphor widely known:

By means of metaphors and analogies, we try to link the new to the old, the novel to the familiar. Under sufficiently slow and gradual change, it works reasonably well; in the case of a sharp discontinuity, however, the method breaks down....Coping with radical novelty requires... [that] one must consider one's own past, the experiences collected, and the habits formed in it as an unfortunate accident of history, and one has to approach the radical novelty with a *blank mind*, consciously refusing to try to link history with what is already familiar, because the familiar is hopelessly inadequate [emphasis added].

...both the number of symbols involved and the amount of manipulation performed [in complex computations] are many orders of magnitude larger than we can envisage. They totally baffle our imagination, and we must, therefore, not try to imagine them (Dijkstra 1989).

This view, while it seems wildly wrong to me, is at least grounded in a definite epistemological theory. To Dijkstra, certain domains like computation (quantum mechanics is another example) are so radically new that they must be approached with a totally blank mind. This theory is basically the opposite of the one we are developing here, namely that computation, like anything else, is understood in terms of structures defined by mappings from other domains of knowledge. In place of metaphor and imagination, Dijkstra advocates the use of formal mathematics and logic.

However, formalism does not really offer an escape from metaphor, for two separate reasons. First, even formal mathematics is riddled with metaphorical terms and concepts, such as the notion of a function having a slope⁴ (a physical metaphor) or being well-behaved (an animate metaphor). Secondly, very few mathematicians would claim that the use of formal methods exempts them from the need to use their imagination!

A more realistic viewpoint on the relationship between metaphor and formalism may be found in Agre's claim that the defining characteristic of technical language is that it links together two separate domains of reference: the real-world domain being formalized and the "Platonic

⁴ These examples are taken from David Pimm's analysis of mathematical language (Pimm 1987).

realm of mathematics" (Agre 1992) (Agre 1996). This cross-domain mapping is essentially a metaphorical process in which aspects of the real world target domain are understood in terms of the formalized and well-understood source domain of mathematics.

Such a mapping will always emphasize certain parts of the world at the expense of others. Aspects that are readily translated into mathematical terms will be preserved by the metaphor, while other aspects will become *marginalized*. The concept of margins derives from Derrida's philosophical deconstructions, in particular the idea that metaphors or world-views can contain "hierarchical oppositions" which classify phenomenon into central and marginal categories. The phenomenon of marginalization will cause research programs to organize themselves around particular central problems (those that the mathematics can describe) while pushing other equally important problems out into subfields, into "areas for future work", or out of consideration entirely. Agre's solution to this form of intellectual tunnel-vision is to deploy new metaphors and more importantly to develop a "reflexive critical awareness" of the role of metaphor in technical work.

Deconstruction is a set of techniques for achieving this sort of awareness by systematically questioning the dominant oppositions and metaphors that structure a field of knowledge. One of these techniques, inversion, is to construct an alternate system that inverts the center and margins created by the dominant metaphor, thereby exposing the assumptions of the original metaphor and bringing the previously marginalized aspects of the phenomenon to attention. Reddy's Conduit metaphor is a good example of such an inversion.

Agre's project is to deconstruct what he sees as the dominant mentalistic metaphors within AI and Cartesian-influenced views in general. Mentalism is a broad structuring metaphor that posits an internal representational space in both humans and computers, in which representational objects dwell and mental processes take place. Mentalism, Agre argues, emphasizes the internal mental processes that take place inside this space at the cost of marginalizing the interactions between inside and outside. To counteract this, he offers the alternative metaphor system of *interactionism*, which emphasizes precisely the opposite phenomena. From the interactionist perspective, the dynamic relationship of an agent and its environment is of central interest, while the machinery inside the agent that generates its behavior is secondary.

My own project can be viewed, loosely, in deconstructive terms. In the next chapter, I explore use of animism in describing the world in general and computation in particular. The language of computation appears to involve both formal mathematical language and a more informal use of various metaphors, particularly metaphors that map computation on to the domain of animacy. While neither mode is completely dominant, the use of animism is generally confined to pedagogic or informal contexts, and is in some sense marginalized by the strong bias in favor of mathematical formalism that pervades academic discourse. By making animism the central concept of my analysis of programming languages, and designing new agent-based languages that incorporate the use of animate metaphors, I hope in some sense to deconstruct the languages of computation.

The purpose of such deconstruction is to be critical, not destructive. Computation has unquestionably provided powerful new conceptual tools for laying hold of the world. But

computation has its limits, as its practitioners are constantly forced to acknowledge. Some of these limitations manifest themselves in reliability problems, brittleness, and the difficulties encountered by both novices and experts in expressing simple ideas in the form of programs. Probing the conceptual underpinnings of the metaphors used to construct computation is one way to try and understand, and possibly circumvent, these limitations.

2.2 Metaphors in Programming

Computers are useless; they can only give you answers.
—Pablo Picasso

Metaphor plays a key role in the discourse of science as a tool for constructing new concepts and terminology. The utility of theory-constitutive metaphors depends upon how accurately the concepts they generate actually do “carve the world at its natural joints”, in Boyd’s terms. More radically constructivist writers have argued over the assumption that such natural joints exist to be found (Kuhn 1993), but most are realists who believe at least that the natural world exists and has structure independently of our metaphors. However, in the discourse of computer science metaphor plays an even more central role. Here metaphors are used not so much to carve up a pre-existing natural world, but to found artificial worlds whose characteristics can then be explored. The metaphors *create* both the world and the joints along which it can be carved.

Computer scientists thus live in metaphor the way fish live in water, and like fish rarely take note of their medium. Their metaphors tend to become transparent, and so terms that start their careers with clear metaphorical roots, such as “structures”, “objects”, or “stacks”, very quickly gather formalized technical meanings that appear to be detached from their origins in the source domain. This phenomenon exists in other fields but is particularly acute in computer systems because the objects involved are so hidden from view that the only terms we have for referring to them are metaphorical. This makes the metaphor dead on arrival—since there is no truly literal way to refer to computational objects, the metaphorical terms soon take on a literal quality.

But under our view of metaphor, technical terminology does not really ever become completely detached from its metaphorical roots. In this section we’ll take a look at some of the metaphors underlying computation and the diverse set of metaphorical models that underlie programming languages. A theme of the discussion will be the idea that anthropomorphic metaphors are often present in computation, in varying degrees of explicitness. This derives from the fact that programs are often concerned with action and actors, and that our tools for understanding this domain are grounded in our understanding of human action. This idea is taken up in more detail in the next chapter.

2.2.1 The Idea of Computation

Computation *itself* is a structuring metaphor for certain kinds of activity in both people and machines. Human behavior may be seen as a matter of “computing the answer to the problem of getting along in the world”, although there are certainly other ways to look at human activity.

Similarly, a computer (which from a more literal view might be seen as nothing more than a complex electrical circuit) may be seen variously as solving problems, monitoring and controlling external devices, servicing its users, simulating a collection of virtual machines, and so forth. The use of the term “computation” to describe the activity of certain complex electronic devices is itself metaphorical, a historical artifact of the computer’s origins. As an organizing metaphor, it privileges certain aspects of the domain over others. In particular, like the mentalistic view of the mind, it privileges the formal operations that take place inside the a computer while marginalizing the interactions of the computer with the outside world.

Historically, computation grew up around the formal notion of a mechanical realization of a mathematical function. A computer was seen as a device for accepting an input string, generating an output, and then halting. This model was perfectly adequate for the tasks that early computers were asked to perform (such as cryptography) but was stretched further by later applications that could not be so readily cast as “problem solving”. In particular, the problem-solving model lacked any notion of an ongoing relationship with the external world. Cyberneticists such as Gregory Bateson were thus impelled to attack the computational model for ignoring feedback relationships (Bateson 1972), and more recently a rebellious faction of the artificial intelligence field has grown dissatisfied with the problem-solving model of controlling autonomous agents and has proposed alternative models that emphasize interaction with the world (Agre 1995).

Of course, despite the limitations of the formal model of computation, computing devices have since been employed in a vast array of applications that involve this kind of ongoing, time-embedded control. A great many more computers are employed as embedded control devices in mechanisms such as cars, airplanes, or microwave ovens than as purely symbolic problem-solvers. So the limitations of the metaphor have not, in this sense, proved to be a limitation on practice. However, they have certainly had their affect on computational theory, which on the whole has had little relevance to the design of embedded control systems.

2.2.2 Metaphors Make Computation Tangible

Explicit metaphors are often used to teach beginners how to program. One common example, by now a cliché, is to describe the interpretation of a program as similar to following the steps of a recipe. These instructional metaphors allow a student to understand the abstruse operations of the computer in terms borrowed from more familiar domains. This is almost a necessity for learning about the operations of a system that cannot be directly perceived. Since the purpose of these metaphoric models is to describe something that is hidden in terms of something visible, the source domains are often taken from the concrete⁵ physical world, such as boxes containing pieces of paper as a metaphor for variables containing values.

⁵ I dislike the term “concrete” and its paired opposite “abstract” in this context, but since they are generally accepted I will continue to use them. The reason for my dislike is that they are inaccurate. The process of executing a **MOVE** instruction inside the computer is just as *concrete* as the act of moving a block from one box to another: it is not an abstraction, but an actual physical change. The difference is not one of concreteness but of familiarity and accessibility to the senses. I prefer the term *tangible* (from the Latin *tangere*, to touch) as a replacement for concrete, because it better denotes the relevant feature of

(Mayer 1989) showed that giving novices metaphorical models of computer language interpreters resulted in improved learning compared to a more literal technical presentation. Mayer used a variety of metaphorical models in his experiments. One such model included mappings such as ticket windows for input and output ports, scoreboards for memory, and a to-do list with an arrow marker for the interpreter's program and program counter. This model was presented both as a diagram and a textual description. Students who were presented with the model did better on tests, particularly on problems requiring "transfer"—that is, problems that involved concepts not presented directly in the original instructional materials. Further studies showed that presenting the model before the main body of material resulted in students who scored higher on tests than those who had the model presented to them afterwards. This supports the idea that familiarity with the model aids in the assimilation of the technical content by giving it a meaningful context.

Sometimes tangible metaphors can result in invalid inferences that bring over irrelevant characteristics of the source domain. In one case, students who were told a variable was like a box inferred that, like a physical box, it could hold more than one object (Boulay 1989). In a similar vein, students shown the sequence of assignment statements **LET A=2; LET B=A** interpreted them to mean (again using the container metaphor) that a single concrete object, **2**, is first placed into **A**, then *taken out* of **A** and put into **B**, leaving **A** empty. In this case the students were overapplying the object and containment metaphors, concluding that **2** had the property of only being capable of being in one place at one time and thus having to leave **A** before it could be in **B**. These sorts of overattribution errors indicate that learning to apply a metaphor is not always a simple matter. In addition to the metaphoric mapping itself, one must also know the limits of its application.

Of course an alternative to using concrete metaphors for computation is to change the computer system itself so that its operations actually *are* concrete (that is, tangible). This possibility is discussed in Chapter 4. Systems with tangible interfaces still might generate problems of invalid inference from the physical domain, but provide an opportunity for users to debug their metaphoric mapping through interaction with the objects of the target domain.

2.2.3 Metaphoric Models for Computation

In this section we examine some of the common models of programming and the metaphor systems that underlie them. These include:

- the *imperative* model, in which the computer is cast in the role of a sequential instruction follower;
- the *functional* model, which emphasizes the computation of values rather than the sequential following of instructions;

being accessible to the senses. There is no such substitute for "abstract", however. *Abstruse* or *obscure*, both of which essentially mean "hidden from view", are better, but they have an unfortunate connotation of inherent difficulty or complexity.

- the *procedural* model, which integrates the imperative and functional metaphors and provides a powerful method of decomposing programs into parts;
- the *object-oriented* model, which reorganizes a program around the data that it manipulates, and is deliberately founded on a metaphor of physical and social objects;
- the *constraint* model, which allows a programmer to combine procedures with declarative information about what they do.

The analysis presented here is in terms of the broad metaphors used to explain and understand programs, and will gloss over many of the more formal properties of programming languages. For instance, (Steele and Sussman 1976) presents the fascinating result that many imperative constructs, such as **goto**, can be easily simulated using only functional constructs given a language that supports recursive high-order procedures. Despite this theoretical result, and many other similar ones that demonstrate that different languages have equivalent formal powers, the basic concepts used to make sense of imperative and functional programs remain quite distinct.

The discussion here sets the stage for the next chapter, which explores one particular aspect of metaphorical grounding in more detail (in particular, see section 3.3.2). This is the role of anthropomorphic or animate metaphors in the description of computational activity. This metaphor system is pervasive in computation for historical and practical reasons. In particular, we will look at *agent-based* models of programming, which are explicitly organized around anthropomorphic metaphors.

2.2.3.1 The Imperative Model

One of the fundamental metaphor systems used to describe computer processes is the *imperative model*. This model underlies most discourse about the hardware levels of computer systems, and is the source of such terms as *instruction* and *command*. The imperative metaphor underlies most naive models of computing such as the transaction level model (Mayer 1979) and the descriptions of computation found in beginner's texts. It also forms the conceptual basis underlying popular early computer languages such as BASIC and FORTRAN. But it may also be found in its earliest forms in Turing's and von Neumann's description of the first theoretical computing machines, and so is really at the very root of the modern idea of computation itself.

The imperative model captures the notion of the computer as a device capable of sequentially executing simple instructions according to a stored program. The basic elements of this metaphoric model are:

- a fixed set of primitive actions that the computer is capable of executing in a single step, that perform operations on memory or on I/O devices;
- a single active instruction follower that reads a program a step at a time and takes an action based on the current step;
- a program that controls the instruction follower, consisting of a sequence of steps that either specify primitive actions or control-flow changes;
- a passive memory that the follower can read from and write to;

- a set of input and output devices.

In the next chapter we will look at the anthropomorphic roots of the imperative metaphor. Here we should just notice the emphasis on a single implicit agent, step-by-step activity, and the mechanical nature of each step. In the imperative metaphor, the interpreter is visualized as a sort of person, albeit a rather stupid or robotic person, reading instructions and following them in a manner that could readily be duplicated by machinery. Each primitive action is simple enough to be executed without any need of further interpretation; no intelligence or knowledge is required of the instruction follower.

What sort of language is suitable for specifying the program for such a computer? This model is called the imperative model because the elements of such languages are *commands*. If such a statement were to be translated into English it would be in the imperative mode. They are instructions to the implicit agent inside the computer. An imperative sentence (i.e., “Give me the pipe!”) has an implied subject, namely the target of the command, which does not appear explicitly as a word but is implied by the structure of the sentence. Similarly, in an imperative language the subject of the instruction does not appear explicitly but is implied—the computer itself, or the instruction follower within it, is the implied subject who will execute the command.

2.2.3.2 The Functional Model

If the imperative model emphasizes control of sequential operations, then the functional model emphasizes values, expressions, and computation in the mathematical sense. In functional languages (i.e. Haskell (Hudak, Jones et al. 1991)), the fundamental unit is not an imperative command, but an expression that specifies a value. While most languages support functional expressions to some extent, pure functional languages enforce the functional style by having no support for state and no imperative constructs like assignment and sequencing. Most functional languages support high-order functions, or functions that can accept other functions as arguments and return them as values.

The functional model uses the concept of a mathematical function as its founding metaphor. Like the imperative model, the functional model was present at the very start of the history of computing. Whereas the imperative model emphasizes action, the functional model emphasizes the results of action, expressed as a functional relation between input and output items. The Turing machine computes a function using imperative operations. In some sense, the joining of these two different ways of thinking in the Turing machine was the founding act of computer science, and the two models continue to be interwoven in various ways as the field grows.

Functional languages are favored as a theoretical tool by computer scientists, because functional programs are much easier to analyze than those that incorporate state and state change. They also permit the use of a variety of powerful expressive techniques, such as lazy evaluation, which are problematic in the presence of state change. Conversely, functional languages do poorly at integrating imperative constructions and state, which in turn introduces issues of time, control, and serialization. There have been quite a few efforts to graft imperative capabilities onto purely functional languages, but as one paper on the subject put it, “fitting action into the functional paradigm feels like fitting a square block into a round hole” (Jones and Wadler 1993).

One of the most successful end-user programming techniques ever invented, the spreadsheet, uses what is essentially a functional model of programming. Each cell in a spreadsheet contains a functional expression that specifies the value for the cell, based on values in other cells. There are no imperative instructions, at least in the basic, original spreadsheet model. In a sense each spreadsheet cell *pulls* in the outside values it needs to compute its own value, as opposed to imperative systems where a central agent *pushes* values into cells.⁶ In some sense each cell may be thought of as an agent that monitors its depended-upon cells and updates itself when it needs to. As Nardi (1993)(Nardi 1993) points out, the control constructs of imperative languages are one of the most difficult things for users to grasp. Spreadsheets eliminate this barrier to end-user programming by dispensing with the need for control constructs, replacing them with functional constructs.

Functional programming lends itself to metaphors of connection and flow. Functions can be pictured as physical devices, akin to logic gates, with a number of input and output ports, continuously computing the appropriate values for their outputs given their inputs. Functional composition then is simply connecting up the input ports of one device to the output ports of other devices. The network acts to continually maintain a relationship between inputs and outputs.

Flow metaphors are straightforward to represent graphically, and there have been quite a number of visual programming environments that make use of them, including Hookup (Sloane, Levitt et al. 1986), VennLISP (Lakin 1986), Fabrik (Ingalls, Wallace et al. 1988) and Tinkertoy (Edel 1986). Graphic dataflow languages like these are especially well-suited to building programs that operate real-time devices or process streams of data. In this context, a program essentially operates as a filter-like device, accepting a stream of data, processing it a single element at a time, and producing a corresponding output stream. Hookup, for instance, was designed to work in real time with streams of MIDI data to and from electronic musical instruments, while LabView was designed to handle laboratory instrumentation tasks.

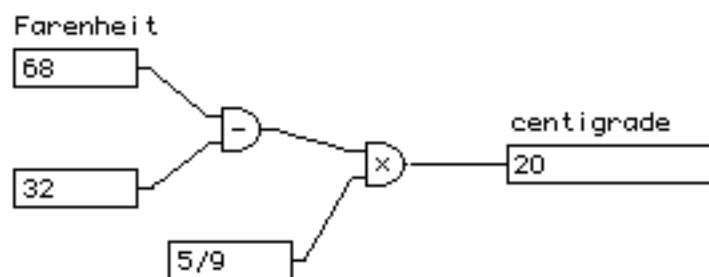


Figure 2.2 (after Hookup). A network for computing centigrade temperature from Fahrenheit. Data flows left to right.

⁶ For a longer discussion of pushing/pulling metaphors in multiagent systems see (Travers 1988).

Figure 2.2 shows an example of a graphic representation, modeled after the appearance of Hookup, of a functional program to convert temperature from Fahrenheit to centigrade units using the formula:

$$C = (F - 32) * 5/9$$

The flow of data is left to right. Functions are represented by graphical symbols akin to those used to represent gates in digital logic. Input and output values are indicated in boxes (input boxes are on the left and supply values, output boxes receive values). In Hookup, the values of outputs are effectively being computed continuously, and so the value in centigrade will update instantaneously whenever any of the inputs change. Note that this presents a somewhat different environment than a standard functional language, in which the application of a function to arguments must be done explicitly. In a Hookup-like language, function application is performed automatically whenever an argument changes. Graphic data flow languages thus take a step towards viewing functions as continuously maintained relations rather than procedures.

Whereas regular functional languages do not deal well with input and output, the dataflow variant is able to model these in the form of continuous streams of changing values. In this model, input devices appear as sources for the flow of values through the networks, but sources that change their values over time. Output devices correspondingly are sinks which accept a stream of changing values. The mouse icon, highlighted at the center of figure 2.3, is an example of an input device, with three separate output flows for **X**, **Y**, and **BUTTON**.

Hookup also extended the flow metaphor to deal with state. Its dataflow metaphor was based loosely on digital logic, with functions represented as gates. In addition to stateless devices such as gates, Hookup included devices with state that functioned in a manner analogous to registers in digital logic. A register had a special clocking input that would cause the current input value to become current and present on the output wire. This way of handling state at least made sense within the dominant metaphor of the system. However, the presence of state also introduces a requirement for sequential control, which was *not* readily supported by the metaphor. Hookup included clocks and clockable sequence objects that provided some ability to provide sequences of values, but using these to control sequences of events was awkward.

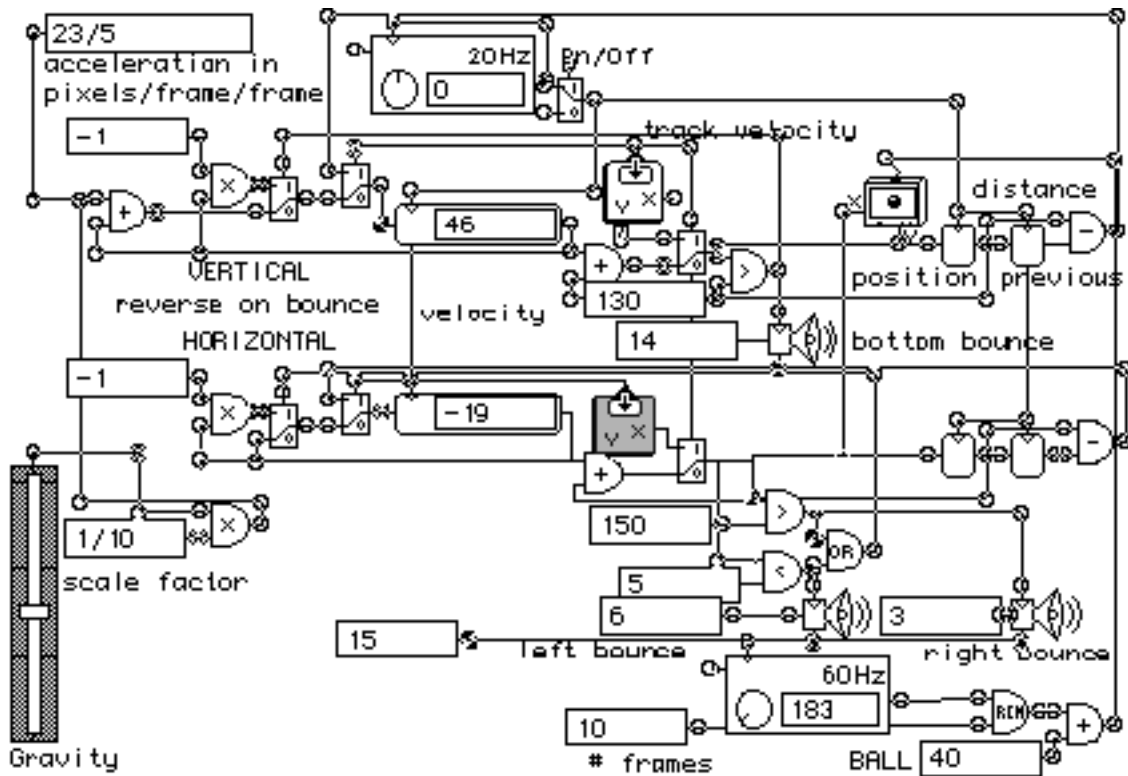


Figure 2.3 A Hookup network to control a bouncing animated figure. It incorporates input from the mouse, output with sound and animation, and internal state registers.

The Playground environment (Fenton and Beck 1989) was another interactive system and language that was organized around a functional model but ran into trouble when trying to deal with tasks that were more naturally expressed using imperative constructs. Playground, like LiveWorld, was designed to be a platform for modeling worlds in which graphic objects were expected to perform actions. The basic unit of computation was an “agent” that functioned more-or-less as a functional spreadsheet cell. Slots inside objects were agents, and each agent was in charge of computing its own value in a functional style (specified by means of an expression language with English-like syntax). This included slots specifying basic graphic information such as x and y position and size. In essence the processing of the system involved a parallel recomputation of all cells, with the new value specified as a functional expression.

As might be expected, it was difficult to express actions using this model. For instance, you could not easily say “go forward 10 steps”—instead, you had to specify separate functional expressions for computing the next values of the x and y coordinates. It was possible to have a rule for, say, the x cell that set its value to $x + 10$ continually, and this which would produce a constant motion in the x direction, and this could even be made conditional, but essentially this model forced the program to be organized around low-level concepts like position rather than behavior. Eventually this spreadsheet-based programming model had to be augmented with additional imperative constructs.

By dispensing with state and control issues, the functional metaphor presents a very simple model of computation that can be made readily accessible to novices through a variety of

interface metaphors. The lack of state and control drastically simplifies the language and makes the system as a whole more transparent. But Playground shows that there are fundamental problems with the use of the functional model as the sole or primary basis for programming animate systems. The downside of eliminating control is that programs that need to take action or exert sequential control are difficult⁷ or impossible to construct. For animate programming, where action is foremost, functional programming seems unnatural in the extreme.

2.2.3.3 The Procedural Model

The procedural model of programming, which underlies modern languages like Lisp and Pascal, combines elements of the imperative and functional metaphors within the more powerful overarching framework of procedural abstraction. The procedural model thus is *not* founded directly on a single metaphor, although it lends itself to new and powerful forms of description based on anthropomorphic and social metaphors.

Under the procedural model, a program is constructed out of smaller programs or procedures. A procedure can both carry out instructions (like an imperative program) and return values (like a function). The procedural model introduces the notion that one procedure can *call* another to perform some task or compute some value. The metaphoric and animate suggestiveness of the term *call* indicates the beginnings of an anthropomorphic, multiagent view of computation. A procedure, encapsulates as it does both the imperative and functional aspects of the computer, is in essence a miniature image of the computer as a whole.

Of course, “procedures” in the loose sense of the word can be created in any language. What procedural languages do is to reify the notion of procedure, so they become objects for the user, that can be manipulated and combined into more complex arrangements. Some beginner’s languages (i.e. Logo) have a distinct notion of procedure, while others (BASIC, at least in its original form) do not. The availability of named procedures can have an effect on the developing epistemology of the student:

In programming cultures like those of LISP, Pascal, and LOGO, in which procedures and hierarchical structures have been given concrete identity, programmers find powerful metaphors in tree searches and in recursive processes. There is a tendency to anthropomorphize, to look at control mechanisms among procedures and within the flow of procedures in terms of actors or demons, or other creatures resident in the computer capable of giving advice, passing data, receiving data, activating procedures, changing procedures, etc. (Solomon 1986, p. 98).

Papert also emphasizes the importance of procedures as a thinking tool. They are a computational realization of what he calls the principle of epistemological modularity, that is, the idea that knowledge and procedures must be broken up into chunks that can be called up from the outside without the caller having to know about the inside:

Everyone works with procedures in everyday life ... but in everyday life, procedures are lived and used, they are not necessarily reflected on. In the LOGO environment, a procedure becomes a

⁷There are some tricky techniques that allow functional languages to express imperative constructs (Henderson 1980) (Jones and Wadler 1993), for instance turning actions into representations of actions that can then be manipulated through functions. These techniques are theoretically interesting but do not really affect the arguments here.

thing that is named, manipulated, and recognized as the children come to acquire the idea of procedure (Papert 1980, p154).

In procedural languages, control is still essentially imperative, in that there is a single locus of control serially executing commands, but instead of a program being an undifferentiated mass of instructions, it is organized into a multiplicity of procedures. The locus of control passes like a baton from procedure to procedure, with the result that one can see the operation of a program in either single-actor or multiple-actor terms.

The procedural model lends itself to an animistic metaphor of “little people” who live in the computer and can execute procedures and communicate among themselves (see section 3.3.2.1). The properties of the procedural model that lend itself to anthropomorphization include the modularization of programs into small, task-based parts and the existence of a simple yet powerful model for inter-procedure communication through calling and return conventions. The procedural model, then, is the first step towards an agent-based model of computation.

Procedures, like the computer itself, can be approached through any or all of the metaphor systems mentioned thus far: imperative, functional, and anthropomorphic. Since a procedure is as flexible and conceptually rich as the computer itself, it essentially permits recursive application of the metaphorical tools of understanding. But the procedural world introduces new powers and complications. Because there are multiple procedures, they need to have ways to communicate and new metaphors to support communication. In languages that support procedures as first-class objects, the metaphor is complicated by the fact that procedures can create and act on other procedures, as well as communicate with them. Issues of boundaries and modularity also arise in a world with multiple actors. Some of these issues are treated by object-oriented models of programming.

Although procedures are anthropomorphized, they are in some sense more passive than the metaphor suggests. They will only act when called from the outside. This, too, derives from the formal Turing model of the computer as a whole, which treats it as a device for computing the answer to a single input problem and then halting, with no interaction with the world other than the original parameters of the problem and the final result. Real computers are much more likely to be running a continual, steady-state, non-terminating process, constantly interacting with external devices. The formal model does not adequately capture this aspect of computation, and the procedural model too tends to marginalize it. Real procedural programming systems often, but not always, make up for this by extending the model to include ways to interface with the outside world, for instance by being able to specify a procedure that will execute whenever a particular kind of external event occurs. This is a useful feature, but still leaves control, interaction, and autonomy as marginal concepts relative to the basic procedural model. One purpose of agent-based models of programming is to bring these issues to the center.

2.2.3.4 The Object Model

Object-oriented programming (OOP) is an interesting example of a programming methodology explicitly organized around a powerful metaphor. In OOP, computational objects

are depicted metaphorically in terms of physical and social objects. Like physical objects, they can have properties and state, and like social objects, they can communicate and respond to communications.

Historically, OOP arose out of languages designed for simulation, particularly Simula (Dahl, Myrhaug et al. 1970) and for novice programming in graphic environments such as SmallTalk (Goldberg and Kay 1976). In object-oriented simulations, the computational objects are not only treated as real-world objects, but they also *represent* real-world objects. A standard example is the spaceship, which is modeled by a computational object that has properties like **position**, **orientation**, and **mass**; and can perform actions like **rotate** and **accelerate**. The object-oriented metaphor explicitly acknowledges the representational relationship between computational structures and real-world objects, and encourages the development of such representational relationships. But because computational objects have properties that are quite different from spaceships and other real-world objects, the elements of the mapping must be carefully selected so that, on one hand, the computational elements are both powerful and parsimonious, and on the other, a sufficiently rich subset of real-world properties and behaviors are encompassed. In most OOP languages, objects are organized into classes or types that make up an inheritance hierarchy.

OOP may be viewed as a paradigm for modularizing or reorganizing programs. Rather than existing in an undifferentiated sea of code, parts of programs in OOP are associated with particular objects. In some sense they are contained in the objects, part of them. Whereas the procedural model offered communication from procedure to procedure, through the metaphor of calling, in OOP, communication between procedures (methods) is mediated by the objects.

A variety of metaphors thus are used to represent the communication and containment aspects of OOP. The earliest OOP languages used the metaphor of sending *messages* to objects to represent program invocation. Objects contain *methods* (or *behaviors* or *scripts* in some variants) for handling particular kinds of messages; these methods are procedures themselves and carry out their tasks by sending further messages to other objects. OOP languages use constructs like **send**, **ask**, or **<==** to denote a message send operation. Objects also contain *slots* that hold state. In general the slots of an object are only accessible to the methods of that object—or in other words, the only way to access the internal state of the object is by means of sending the object messages. Objects can have a tightly controlled interface that hides its internal state from the outside world. The interface of an object thus acts somewhat like the membrane of a cell.

These simple elements have given range to a wide variety of extensions to the basic metaphor, and a correspondingly vast literature on object-oriented methodologies and more recently, object-oriented “design patterns” (Gamma, Helm et al. 1994). The diversity of such schemes indicates that while the basic mapping between computational and real-world objects may be intuitive and straightforward, the ramifications of that mapping are not. In any real OOP system, there are always hard design choices to make reflecting the fact that there will always be more than one way to carve the world up into objects.

The history of object-oriented programming shows how technology that evolved around a particular metaphor can be subject to forces that tend to stretch or violate that metaphor. The

original simple idea behind OOP—objects receive messages and decide for themselves how to respond to them—is complicated by many issues that come up when trying to realize the idea. One complication is the related issues of object types, hierarchies, inheritance, and delegation. Multiple inheritance, while a useful technique, involves quite complicated issues and does not have a single natural formulation. The prototype-based, slot-level inheritance of Framer and LiveWorld (see chapter 4) are attempts to deal with some of these problems in a more intuitive way.

It is known that message-passing and procedure calling are formally equivalent (Steele 1976). Some OOP languages (like CLOS (Steele 1990) and Dylan (Apple Computer 1992)) try to exploit this by getting rid of the message-passing metaphor and using regular procedure calling to invoke object methods. This has the advantage that method selection can be specialized on more than one object. This technique, while powerful, is somewhat at variance with the object-oriented metaphor as previously understood. Because a method can be specialized on any argument, the method can no longer be seen as associated with or contained inside a single object or class. Here we have a case of two metaphors for communication clashing and combining with each other. Proponents of the generic procedure approach point out that it is more powerful, more elegant, and (in the case of CLOS) more integrated with the underlying procedural language. Opponents decry the violation of the object metaphor and the increased complexity of dispatching on multiple arguments.

The Actor model of computation (Hewitt 1976) was another important early influence in the development of OOP, and deserves mention here. The name is obviously anthropomorphic, and a variety of anthropomorphic metaphors influenced its development, including the little-person metaphor (Smith and Hewitt 1975) and the scientific community metaphor (Kornfeld and Hewitt 1981). The Actor model was explicitly designed to support concurrency and distributed systems.

The object-oriented programming model is a natural outgrowth of the procedural model, and shares a good many of its features. From a broad historical perspective, it can be seen as a further step in the reification and anthropomorphization of parts of programs, necessitated by the need to manage more complex programs and distributed systems. Rather than programs and data existing in an undifferentiated mass, the various components are organized, managed, and encapsulated. The emphasis as a result is on communication between the now separated parts of the system.

2.2.3.5 The Constraint Model

The constraint model is something of a departure from the other programming models considered so far. Despite their differing metaphoric bases, to program in all the previous models is to provide the computer with a fully deterministic procedure for carrying out a computation. Even functional languages generally have an imperative interpretation so that the programmer will be aware of the sequence of events which will occur when the program is executed. Constraint languages, in contrast, implement a form of declarative programming in which only the *relations* between objects are specified by the programmer while leaving the procedural details of how to enforce these relations up to the constraint-solving system. As a

result, constraint languages require significantly more intelligence in their interpreter, whose operation is thus harder to understand.

However, from the metaphorical standpoint constraint systems may be seen as a natural extension of the flow metaphor found in functional languages. In a functional language, flow is unidirectional, but in a constraint system, data can flow in either direction along a link. To illustrate this, let's return to the temperature conversion example of section 2.2.3.2. In a constraint language, the statement:

$$C = (F - 32) * 5/9$$

is not only an instruction about how to compute C given F, but a general declaration of a relationship between the two quantities, so that either may be computed from the other. The constraint system has the responsibility for figuring out how to perform this calculation, and thus must have some algebraic knowledge or the equivalent.

This knowledge takes the form of a variety of constraint-solving techniques. The simplest technique, local propagation of known values, is readily expressed through the flow metaphor. In the functional version of the flow metaphor, values flow from inputs along wires, through devices, and eventually produce output values. In local propagation, values flow in a similar way, but the wires are bi-directional and input and outputs are not distinguished. There are still computational devices, but instead of having distinguished input and output ports any port can serve as an input or an output (or in other words, instead of implementing functions they implement relations). Such a device will produce an output on any port whenever it receives sufficient inputs on its other ports.

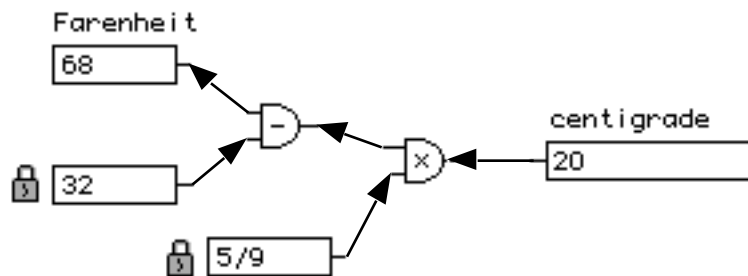


Figure 2.4: A constraint network.

Figure 2.4 shows a constraint-based variant of the dataflow network from Figure 2.2. In contrast with the earlier figure, here data can flow in either direction along a wire, so that the value in the centigrade box might have been specified by the user or computed by the network. The arrows show one possible flow pattern, which would result from the user specifying the value of centigrade. The distinction between input and output boxes no longer holds, but a new distinction must be made between constants and variables—otherwise the network might choose to change the value of the constant 32 rather than the value in the Fahrenheit box! Constants are indicated by putting a padlock beside them, to indicate to the solver that it is not to alter those values.

This circuit-like metaphor for constraints was introduced by Sketchpad (Sutherland 1963) which was the first system to represent constraints graphically. The technique was used as an expository device in (Steele 1980) and implemented as an extension to ThingLab (Borning 1986).

Constraint programming may be viewed as one particularly simple and powerful way of combining declarative and procedural knowledge.⁸ A constraint combines a relationship to enforce, expressed in declarative terms (i.e., an adder constraint that enforces the relationship $a = b + c$) with a set of procedural methods for enforcing it. In the case of the adder, there would be three methods corresponding to the three variables that can serve as outputs; each of which computes a value from the remaining two variables

Other languages, most notably logic programming languages like Prolog, have been fashioned along the idea that programs should take the form of declarative statements about relationships. One slogan put forth by advocates of logic programming is “algorithm = logic + control” (Kowalski 1979), where “logic” refers to an essentially declarative language and “control” refers to some additional mechanisms for controlling the deductive processes of the language interpreter. The problem with Prolog is that it shortchanges both logic and control by attempting to use the same language to specify both. Thus the language does not have the full expressive power of logic, because doing so would make the imperative interpretation of a logic program intractable. And while Prolog adds some control constructs such as the **cut** to its declarative language, in general the ability to control the deductions of the interpreter are limited.

Constraint languages have the potential to overcome this limitation, since they separate out the declarative and imperative parts of a program. Declarations of intent and procedures can each be made in the idiom appropriate to the task, then linked together in a single constraint. ThingLab (Borning 1979) was organized in this fashion. However, rather than develop this idea as a programming paradigm, ThingLab, its descendants (i.e. (Freeman-Benson, Maloney et al. 1990)), and constraint-based systems in general evolved in a different direction.

A typical contemporary constraint system (i.e. Juno-2 (Heydon and Nelson 1994)) is designed as a declarative language together with a black-boxed constraint solver which can solve constraint systems in that language. The user of such a system can specify constraints using a given declarative language, but cannot specify procedures for satisfying them. In other words, the imperative side of the equation is given short shrift. The reason this path was taken is probably a desire for constraint solvers that are both fast and theoretically tractable. A system that permits constraints to be built with arbitrary user defined procedures would be quite difficult to control.

Constraint Imperative Programming (CIP) (Lopez, Freeman-Benson et al. 1993) is an attempt to integrate declarative constraints with an imperative programming language. The idea behind CIP is to combine an object-oriented procedural language with automatic constraint solving capability that can enforce relations between slots of objects. While this is a promising line of

⁸ As far as I know this idea was first formulated by Alan Borning (Borning 1979).

research, CIP languages are still limited in their expressive power. The constraint solver is monolithic—you can't create new imperative methods to solve declarative constraints—and in some sense subordinate, relegated to the role of automatically maintaining relations among variables in an otherwise procedural language. The same is true of other efforts to combine constraints with more general models programming (Hetenyck 1989) (Siskind and McAllester 1993).

These efforts, while certainly valuable, do not seem to me to explore the full range of possibilities of constraints as an organizing metaphor for programming. All of them take an essentially fixed constraint solver and make it an adjunct to an otherwise ordinary procedural language. A programming system that was fully organized around a constraint metaphor would have to have a much more powerful concept of constraints, one that could encompass computation in general, as does the object-oriented model. Constraints would have to be as powerful as procedures (which they could be, if they had a fully expressive procedural component) but also capable of being invoked without an explicit call. The agent-based systems described in chapter 5 are attempts to realize this alternative version of the constraint model.

2.2.4 Interface Metaphors

The only area of computer science that makes a regular practice of engaging in explicit discourse about its own metaphors is the field of human interface design. No doubt this is due to the nature of the interface task: to make hidden, inner, abstruse worlds of computational objects and actions accessible to users who may not have any direct knowledge of their properties.

From our point of view, just about everything about computation is metaphorical anyway, and the distinction between a highly metaphorical interface (such as the Macintosh desktop) and a command-line interface (such as that of UNIX) is a matter of degree only. It is not hard to see that the UNIX concepts of files and directories are just as grounded in metaphors as the documents and folders of the Macintosh—the former have simply been around longer and thus have achieved a greater degree of transparency.

Interface metaphors are generally easier to design than metaphors for programming languages, for a couple of reasons. First, interface metaphors are usually designed for more specific tasks than languages. They generally have only a small number of object types, relations, and actions to represent, and so each element can be given a representation carefully tailored to its purpose. A programming language, by contrast, has an extensible set of objects, operations, and relations, and an interface that presents its elements to the user must necessarily operate on a more general level. For example, a non-programmable interactive graphics program might have special icons for each of the objects it allows the user to draw (i.e. rectangles and lines) and operations it can perform (i.e. erase, move, resize). On the other hand, a general-purpose graphic programming environment will have an ever-expanding set of objects and operations and thus its interface, unless extended by the user, can only represent objects and operations in general, and thus will be limited in the metaphors it can employ.

One way to deal with this problem is to find an intermediate level between application-specific but non-programmable tools, and general-purpose environments that are programmable but difficult to present metaphorically. This approach is used by Agentsheets (Repenning 1993), an environment building tool which is designed using a layered approach. The substrate level is a general-purpose programming environment featuring a grid-based spatial metaphor which can contain active objects, programmed in a general-purpose language (an extension of Lisp called AgenTalk). Adapting Agentsheets for a new domain task involves having a metaphor designer build a set of building blocks that work together using a common metaphor, such as flow. The end-user then can construct simulations using the block-set and the metaphor, but is insulated from the power and complexity of the underlying programming language.

Another reason that interface metaphors are easier to design than metaphors for programming is that direct-manipulation interfaces in general don't need to represent action. Under the direct-manipulation paradigm, all actions are initiated by the user. While the results of action or the action itself might have a graphic representation (for instance, opening a folder on a desktop can be represented by a zooming rectangle) the action itself is soon over and does not require a static representation. Nor is there a need to represent actors, since the user is the only initiator of action in the system. The interface world is beginning to realize limits to the direct-manipulation paradigm and embrace representations of action in the form of anthropomorphic interface agents (see section 3.3.4).

Programming, however, is all about action and thus programming environments have a need to represent actions and actors. It's interesting that functional programming models, in which action is de-emphasized, have been the most amenable to presentation through graphic metaphors. To represent action, programming metaphors may also need to turn to anthropomorphism.

2.3 Conclusion

My main complaint is that metaphor is a poor metaphor for what needs to be done.
— Alan Kay (Kay 1990)

Our basic ideas about computation are founded on metaphoric models. The daily activity of programmers involves inventing and dealing with new forms of computational structure, and thus they are engaged in a continual process of inventing and deploying metaphors that make these structures understandable. Such metaphors are not created arbitrarily, however. This point is often lost in battles between constructivists and objectivists. Our ideas about the world are constructed, but they are not constructed just as we please. Scientific metaphors must map everyday concepts onto the actual properties of the natural world and thus the choices of source domains and mappings are highly constrained. There may be more freedom in choosing metaphors to organize computation, but not all metaphorical structuring will prove equally powerful.

A good computational metaphor should use only a few real-world concepts for its source domain, and map them onto computational concepts that are powerful. It is hard to capture exactly what determines the power of a metaphorical system. Perhaps an example of a *bad*

metaphor would help: I once used a language that had started as a tool for building computer-assisted instruction “lessons”. It had since grown into a general purpose language, but was still structured in terms of lesson units and control flow was organized around the underlying metaphor of question-and-answer drills. All sorts of operations had to be shoehorned into this framework. The question-and-answer metaphor did not translate into powerful and general tools for computation, and as a result the language was quite awkward to use.

The moral is that not all real-world domains are suitable for use as the basis of a general computational model. The ones we have examined in this chapter are dominant because they *are* powerful and general, and because they have evolved over time as the nature of the computational realm has become better understood. But we should hope that the evolutionary process has not finished and that there is room for the invention of new models for programming that will overcome the limitations and biases of the existing ones.

In a sense, programming is a form of writing. The most basic model of computation, the imperative model, is closely linked to the idea of narrative and drama. Programming is a process of writing scripts for computational actors. Yet the imperative model of programming is theoretically suspect, spawning a search for languages that are more tractable, such as functional programming. Functional programming has not made wide inroads in environments for novices or even for practicing programmers. The reasons for this are many, but surely one is that functional programming discourages thinking of computation in terms of actions and actors. Instead of fighting against the tendency to think in terms of actions and actors, we should work with our natural epistemological tools, and embrace the use of metaphors based on the actors we know, that is to say, humans and other animate entities. The nature of animism and its use in computation is the subject of the next chapter.

Chapter 3 Animacy and Agents

Sometimes I think it is a great mistake to have matter that can think and feel. It complains so. By the same token, though, I suppose that boulders and mountains and moons could be accused of being a little too phlegmatic.

— Winston Niles Rumfoord

3.1 Introduction

There appears to be a fundamental dichotomy in human thought that separates the living from the inanimate, the purposive from the mechanical, the social from the natural, and the mental from the physical. The tendency to divide the world along such lines is strong, possibly innate, crucial to social functioning, and a key part of folk psychology (Dennett 1987). This grand divide is no doubt an example of what Minsky calls a “dumbbell theory”, or false dualism (Minsky 1987, p117). I should make it clear that the dualism explored in this chapter is an epistemic dualism that exists in our theories of the world, not in the world itself, and thus should be distinguished from ontologically dual theories such as Cartesianism or vitalism.

Science has, at least on the surface, outgrown both sorts of dualism. But because the distinction between animate and inanimate is such a universal and basic part of human thinking, it haunts even the most sophisticated efforts to grapple with the nature of the mind. We all begin as dumbbells, and we do not easily rid ourselves of our dumbbell theories. Computational theories of the mind and computation itself are in some ways attempts to bridge this gap—to make lifelike behavior and intelligence from inanimate parts, and to explain the behavior of living things in terms drawn from the physical world. But the partial successes of AI, cybernetics, and cognitive science in devising theories that partially unite these realms has not yet remade our everyday minds accordingly. Nor are the scientific theories themselves immune to the effects of this persistent divide.

It is a common practice to describe the activity of machines or other non-living objects in animate terms. I will refer to these sorts of mappings as *animate metaphors*. The term *anthropomorphism* is more common, but is not quite as accurate, since other animals besides humans can serve as the source domain. Another suitable term is *ethopœia*, the classical Greek word for attributions of human qualities to non-human objects (Nass, Steuer et al. 1993). This chapter explores the use of animism and animistic metaphors in the construction of the computational realm. In particular, it examines the ways in which animacy is used to make the operations of complex systems such as computers comprehensible.

Animate metaphors are powerful because they are capable of exploiting our knowledge about humans and human action. When dealing with systems that are too complicated to be understood in physical and mechanical terms, the leverage that animate metaphors can give is crucial. It allows us to think in terms of purpose and function, and thus to understand without having to know the “implementation details” of the system. At the same time, the use of animate

metaphors to describe machines can be dangerous, because of the risk of overattribution errors. For instance, we can talk of a simple robot vehicle using negative feedback control as “wanting” to propel itself towards some stimulus (as in (Braitenberg 1984)) but the nature of this wanting is not the same as that of a person—the vehicle does not know that it wants, does not necessarily feel anything like disappointment if it does not get its goal, and so forth. The metaphor is compelling, but can also be misleading.

The first section of this chapter explores how the concepts of animate and inanimate arise, how they develop, and how they are used. It traces a few key properties of the animate realm through various attempts to analyze the two domains. Three key concepts emerge as the defining properties of the animate realm: autonomy, purposefulness, and reactivity.

The second section looks at how animate metaphors have been used in computation and what roles they play in the description of various programming languages. We will find that while animism plays a central role in the foundation and continuing development of computation, it tends to be a rather peculiar and limited form of animism, one that omits some of the most important qualities of the domain.

The last section explores a new kind of programming paradigm based on *agents*, that is, on collections of explicitly anthropomorphic units. Agent-based programming seeks to combine the power of computation with the aspects of animism that previous forms of computational animism have left out. It will also explore the use of narrative as a natural way to express the action and interaction of animate beings.

3.2 The Realm of Animacy

Animacy is a basic category, yet one that is difficult to define precisely. The category of animate things is not the same as living things (since plants and sessile animals for the most part should not be included, while robots and other artificial beings perhaps should be). Animacy is more than simply a category, but constitutes what Minsky calls a “realm of thought” with its own distinctive style of mental representation and reasoning. The dichotomy between animate and inanimate seems to manifest itself in different ways at different cognitive levels. In this section I attempt to sketch out some different approaches to understanding how we think about the animate realm. The roots of animacy are found in basic perceptual processes, but its ramifications go beyond perception and affect the way we think about action and social behavior.

3.2.1 The Perception of Causality

The division between animate and inanimate appears to be both a universal and innate part of how we see the world. An obscure but fascinating thread of research in perceptual psychology indicates that the ability to distinguish between animate and inanimate objects may be in some sense perceptual primitives, hard-wired into our brains at the lowest level (see (Gelman and Spelke 1981) for a survey). At this level, animacy is manifest in the perception of motion, specifically motion that appears to arise autonomously, without visible external

causation. The ability to distinguish between caused and autonomous motion appears to be innate and can be detected experimentally.

Research on the perception of causality began with the work of Michotte (Michotte 1950). His experiments involved showing subjects a set of moving abstract shapes on a screen, and asking them to make judgments about what they had seen. In a typical experiment, a moving shape A would approach a second stationary shape B. When A contacted B, depending upon the experimental situation B might start to move either immediately, or after a span of time, and A might stop or continue to move. The subjects were then asked to interpret what had happened.

All subjects tended to interpret these scenes in terms of causal relationships between the two shapes: that is, the experimental situation would strongly suggest either a direct causal relationship between the two movements, or an autonomous (non-caused) origin for the movement of the second object. For example, in the case where A stopped and B immediately began moving, the scene was interpreted as a collision or “launching”. In the case where A continued to move along with B, the scene was interpreted as a case of A dragging B along with it. In another case, similar to the first but with the difference of including a delay before B began to move, the subjects would attribute autonomy or self-causation to B’s movement. The striking thing about these experiments was the immediacy and power of their ability to induce particular kinds of causal perceptions in the subjects, given the highly abstract character of the objects and motions involved.

Michotte’s experiments have been replicated with six-month old infants (Leslie 1979)(Leslie and Keeble 1987). It was found that these young subjects could distinguish between causal and non-causal scenarios. These experiments involved showing the subjects a sequence of presentations similar to those of Michotte, which would consist of all causal presentations with one interposed non-causal presentation, or the inverse. In either case the infants would show a startle response when the interposed presentation was shown.

Stewart (1982)(Stewart 1982) performed similar moving image experiments on adults. In this case, both the motions and the judgments observers were asked to make were somewhat more complex than in the earlier experiments. Stewart’s shapes would follow one another or avoid one another, or start, stop, or change directions arbitrarily. Observers were asked to judge whether the shapes in motion were “objects or creatures”. All of these motions induced judgments of animacy or aliveness in observers, whereas shapes that followed less complex paths were judged as inanimate or mechanical.

To generalize from the results of these experiments, it appears that humans strongly distinguish between externally caused and autonomous motions, and that the ability to do so is to some extent innate, or at least appears early in development. These perceptual abilities seem to form the perceptual basis for the later development of concepts about living vs. non-living things and purposefulness. To a first degree of approximation, any object moving in ways that appear to violate basic Newtonian laws of motion—that is, an object that changes its motion without apparent external cause, other than friction—tends to induce a judgment that the object is alive, purposeful, and autonomous. On the other hand, objects in unchanging motion, or

objects that change their motion only in response to visible external causes, are judged to be inanimate.

The implications of these experiments are that the perceptual system is attuned to changes in motion and relations in changes in motion, and that there is a need to interpret or explain such changes by assigning their cause to either internal or external factors (Premack 1990). The apparent innateness and universality of these effects argues that causality is in some sense a perceptual primitive. Related concepts of aliveness, purpose, intent, self-awareness build upon and refine these primitives.

That there should be such an innate mechanism makes a good deal of sense from an evolutionary standpoint, since it is important for animals to be able to judge which objects in their environment are alive and which are not. From the physiological point of view, it is well-known that the visual system includes many sorts of motion detection mechanisms. The ability to sense changes in motion and relations between different motions is a step up in complexity, but not a very large one. There is even some evidence from studies on brain-damaged and autistic subjects that indicates the presence of a physiological localized mechanism to detect agency and causality (Baron-Cohen 1995).

Do even six-month old infants construct causal models of their world and distinguish between external and internal causes? It seems more likely that there are at least two levels of processing at work here: one, the level of basic perceptual processing, and another process of interpretation and explanation that involves more conceptual thought. The perceptual mechanisms may be able to provide the basic ability to distinguish between caused and uncaused motions, while the conceptual mechanisms, which develop later, have the task of fitting these perceptions into categories and constructing explanations for them.

3.2.2 The Development of Animism as a Category

Piaget's studies of children's thinking revealed a phenomenon he called *childhood animism* (Piaget 1929). This is the tendency of children to attribute properties normally associated with living things to the non-living. The qualities that are projected onto the inanimate world by childhood animism include both aliveness and consciousness. These are obviously two separate concepts, but they apparently follow similar stages through development, suggesting that they are based on a common underlying concept. Other qualities that are brought into play by childhood animism include freedom and the ability to have intentions or will. While some of Piaget's results have been challenged as being oversimplified (see (Carey 1985)), his central results are still of interest.

Piaget found that children will attribute the quality of being alive to different object types at different stages of development. In general, each stage exhibits a refinement, or narrowing, of the class of things that are considered alive. The stages are outlined in Table 3.1.

As a result of this developmental process, children can exhibit notions of aliveness that depart quite dramatically from the adult version of the concept. For instance, a child of 8 (in stage 1), when asked if a bicycle was alive, replied “No, when it doesn’t go it is not alive. When it goes it is alive”. Objects might be endowed with consciousness of some things but not others:

If you pricked this stone, would it feel it?—*No*.—Why not?—*Because it is hard*.—If you put it in the fire, would it feel it?—*Yes*.—Why?—*Because it would get burned* (p176).

Children in the earlier stages (0, 1, and 2) tend to produce mixed and unsystematic judgments. Piaget characterizes stage 2 as primarily a transition period in which the child develops the distinction between things that move on their own (such as a person) and things that only move under outside impetus (such as a bicycle). When this distinction is made the child reaches the third stage, which Piaget considered to be the most “systematic and interesting” of the four. Here is an example of a child struggling to express the Stage 3 idea that autonomous motion is the salient quality that determines the ability to feel:

Tell me what you think, what makes you think that perhaps the wind doesn’t feel when it is blowing?—*Because it is not a person*.— And why do you think perhaps it does feel?—*Because it is it that blows* (p183).

Piaget gives special attention to this last answer as a striking illustration of the nature of third-stage childhood animism. “It is it that blows” beautifully illustrates the child’s mind arriving at the idea that it is *initiators of action* that are thought to be feeling and thus animate.

Piaget’s view is that only at the third stage (usually reached at ages 7 or 8) do children make the distinction between motion in general and autonomous motion. This would appear to be in conflict with the results of Michotte, Leslie, *et al*, who believed that this distinction was present much earlier. This conflict might be resolved by realizing that the Piagetian studies operate purely in the conceptual realm, whereas the other researchers are studying the perceptual ability to detect distinctions. Children may have an innate ability to distinguish types of observed motion, but lack the ability to construct coherent explanations for such motion. The latter ability must be learned, and in some sense the Piagetian developmental sequence is a case of the conceptual mind lagging behind and eventually catching up to the distinctions generated

Stage	Name	Description
Stage 0	No concept	random judgments
Stage 1	Activity	anything active is alive
Stage 2	Movement	only things that move are alive
Stage 3	Autonomous movement	only things that move by themselves are alive
Stage 4	Adult concept (animals)	only animals (and plants) are alive.

Table 3.1: (after (Carey 1985)) The stages of development of the concept of “alive”.

innately by the perceptual system.

Piaget's theory of animism has been critiqued on a number of grounds. For instance, the questions asked by the experimenter can induce particular forms of thought and styles of judgment (Piaget himself was aware of this). Also, children at all ages seem to use a large variety of techniques for judging whether or not something is alive, and never use exclusively the fact of autonomous movement (Carey 1985). Carey's critique seems valid—the development of children's representation of the concept of living things is no doubt a more complex story than the simple stage theory has it. However, as Carey admits, her study is not aimed at the heart of the Piagetian theory of animism, which is concerned more with the developing notions of causality rather than the definition of the word "alive" in any narrow sense. Despite her disagreements with Piaget, she agrees that "the separation of intentional causality from other types is central to what is changing." This is really the only aspect of the theory that concerns us here.

3.2.3 Frameworks of Understanding

A more sophisticated way to handle the distinction between animate and inanimate is to treat them, not as categories into which objects must be slotted, but as two alternative approaches with which to analyze everyday phenomena. Even children are able to flexibly apply these approaches to some extent (recall the child who thought the bike was alive when it moved and not otherwise). Philosophers and psychologists, who perhaps have a greater need to think in fixed categories than children, often try to fix these approaches into categories, but even they eventually have abandoned vitalism and dualism, recognizing the need to treat animacy and aliveness as epistemological stances rather than fixed facts of the world. Dennett has articulated this about as well as anybody with his distinction between intentional and physical stances (Dennett 1987), but see also (Boden 1978) on the checkered history of purposive explanation in psychology, and (Schaefer 1980) on how Freud blended both mechanistic and animate stances in his metapsychology.

The sociologist Erving Goffman explored the question of how such stances (frameworks, in his terms) are used in everyday life and how they function in organizing cognition and society. His approach (Goffman 1974) identifies two primary frameworks: social and natural, which correspond roughly to Dennett's intentional and physical stances:

When the individual in our Western society recognizes a particular event, he [employs] one or more frameworks or schemata of interpretation of a kind that can be called primary. I say primary because application of such a framework or perspective is seen by those who apply it as not depending on or harking back to some prior or "original" interpretation; indeed a primary framework is one that is seen as rendering what would otherwise be a meaningless aspect of the scene into something that is meaningful...

In daily life in our society a tolerably clear distinction is sensed, if not made, between two broad classes of primary frameworks: natural and social. Natural frameworks identify occurrences seen as undirected, unoriented, unanimated, unguided, "purely physical"...It is seen that no willful agency causally and intentionally interferes, that no actor continuously guides the outcome. Success or failure in regard to these events is not imaginable; no negative or positive sanctions are involved. Full determinism and determinateness prevail.

Social frameworks, on the other hand, provide background understanding for events that incorporate the will, aim, and controlling effort of an intelligence, a live agency, the chief one being the human being. Such an agency is anything but implacable; it can be coaxed, flattered, affronted, and threatened. What it does can be described as “guided doings.” These doings subject the doer to “standards,” to social appraisal of his action based on its honesty, efficiency, economy, safety, elegance... A serial management of consequentiality is sustained, that is, continuous corrective control, becoming most apparent when action is unexpectedly blocked or deflected and special compensatory effort is required. Motive and intent are involved, and their imputation helps select which of the various social frameworks of understanding is to be applied (p.21-22)/

These “guided doings” are subject to constant evaluation both from the doer and from other participants on the scene as to their efficacy and quality, distinguishing them from purely natural happenings, which are not so judged. Social frameworks, but not natural frameworks, involve actors, that is, humans (or occasionally other entities such as animals or gods) that can initiate actions and are responsible for their success.

Framework-based interpretation is fluid; a scene that appears to be a guided doing may be transformed (through accidents or “muffings”, or by the realization that an act that seemed intentional was actually an accident) into a scene in which physics takes over from intention (for example, an ice-skater who slips and falls).

The concepts of muffings and fortuitousness have a considerable cosmological significance. Given our belief that the world can be totally perceived in terms of either natural events or guided doings and that every event can be comfortably lodged in one or the other category, it becomes apparent that a means must be at hand to deal with slippage and looseness (p34-35).

The relevance of Goffman to this discussion is his specifically cultural perspective on the dichotomy between animate and inanimate. The ability to distinguish between caused and autonomous motions may be innate, but the types of situations in which this distinction is applied can be enormously varied and conditioned by culture. For instance, differences in religious views can often lead to differing use of frameworks: where some might see a natural disaster others might see the will of a god.

The animate realm now takes on aspects of morality and the necessity of judgment: our social lives require us to make these distinctions and to be subject to them. Trials in which insanity defenses arise are an excellent example of a situation in which the distinction between natural and social arises. A defendant who can be judged subject to the presumptively external causation of insanity can escape having his actions judged according to the standards of social frameworks. Social frameworks impose standards. Actions viewed within social frameworks can be successful or otherwise, because they are initiated and guided by social actors.

3.2.4 Animacy and the Representation of Action

Animacy appears to be a basic category of the human mind, appearing in a variety of forms at different levels of cognition. But to think of animacy as simply a category, in isolation from its role in understanding, is to miss its significance, which stems from its key role in the understanding of action and causality.

To try and understand the place of animacy in cognition, we need a theory of representation that takes into account the importance of action. We will use Minsky’s frame-

based theory of understanding (Minsky 1987, p245), in particular the theory of *trans*-frames which has its roots in Schank's conceptual dependency scheme for representing semantic information (Schank 1975). Using these tools, I will attempt to sketch out such a cognitive theory of animacy.

In Minsky's theory, a frame is essentially a conceptual template or schema, representing a single concept such as "chair" or "throw", with labeled slots or *terminals* that serve to connect it to other frames that represent related parts. So for instance a "chair" frame would have terminals to represent legs and back, while a "throw" frame would have terminals representing the object thrown, the thrower, and so forth. Understanding a scene, sentence, or situation involves selecting a frame for it and finding appropriate entities to fill its terminals.

One particularly important kind of frame, used to represent action and change, is called a *trans*-frame. A *trans*-frame is a generalized representation of action, which can be specialized for particular circumstances by filling in terminals for various roles such as ACTION, ORIGIN, DESTINATION, ACTOR, MOTIVE, and METHOD. *Trans*-frames roughly correspond to sentences, in terms of the scale of the concept they encode, although the frame with its many terminals is capable of including more information than a typical sentence. Schank further categorizes *trans* events into PTRANS, MTRANS, and ATRANS types, corresponding to whether the action involves physical movement, the transmission of information, or the transfer of abstract relationships like ownership.

Trans-frames give us a way to think about the representation of action, and thus a way to think about animacy, leading to a simple theory of what it means for something to be animate: A thing will be seen as animate if its representation can plausibly fit into the ACTOR terminal of a *trans*-frame. Or in other words, something will seem animate if it can be seen as the initiator of an action. From the standpoint of language, this is equivalent to saying that it can be the subject of a sentence describing the action (assuming the sentence is in the active voice). This definition of animacy suggests the classical Aristotelian definition of an actor or agent as one who initiates action. Of course, this definition is slightly tautological, and it doesn't tell us what things will be able to take on this role or why. But it does give us a convenient way to think about how the animate category functions in relation to other mechanisms for understanding action.

Consider again the example from Piaget, where a child explains why he thinks that the wind can feel—"because it is it that blows". It appears that the child has a frame for BLOWING, which has an actor slot that needs to be filled. Unable to find anything else to put in that slot, the wind itself must be there: "it is it that blows". And, when the wind takes on this role, it is somehow endowed with the properties of the living, despite the fact that the child had concluded earlier that the wind could *not* feel, because "it is not a person". Another child denies animacy to the wind, "because it is the cloud that makes it blow" (that is, the wind here is an object rather than an actor), and yet another bestows feeling upon the clouds for the same reason. ACTORhood seems to play a powerful role in determining whether or not something is thought to be animate.

Why are actions and actors so important to cognition? Although the world may be in reality a unified flux of physical causation, we can't think about it that way on a practical level. Instead, we need to divide the changing world up into representable parts such as objects and actions.

An action is a segment of activity with a beginning, generally in the form of an actor's intention, and an end, in which the world is in a new state as a result of the action. The idea of action, and the related ideas of autonomous, intentional actors (and the notion that they can have "free will") is an artifact of this need to represent the world in manageable chunks.

More complex reasoning requires understanding sequences of events, which in turn involves chaining *trans*-frames together. In such a chain, individual objects will change their roles: what was an OBJECT of action in one frame may become the ACTOR in the next. The ability to manipulate chains of changes is important to reasoning and suggests that the ability to switch between animate and inanimate may actually be a necessity born of the need to create such chains (see (Ackermann 1991) for a similar argument). The act of explanation chains trans-frames in the opposite direction: when we ask a question like "what made him do that?" we are seeking to build a frame that puts what was an ACTOR into an OBJECT-like role. When an ACTOR's initiation of action is explained by its internal properties, it appears purposeful; when explained by external factors, it appears reactive. If it can't be explained at all, which is often the case, the action might have to attributed to something like free will.

3.2.5 Conclusion: the Nature of Animacy

We have attempted to define the animate realm through multiple approaches. Based on the above analyses, I would like to posit three properties which together seem to characterize the animate domain:

- **Autonomy:** the ability to initiate action. Animate creatures are capable of taking action "on their own", without an apparent external physical cause.
- **Purposefulness:** the actions undertaken by animate creatures are often directed towards the achievement of a goal. This implies further that such actions can be evaluated as to their effectiveness.
- **Reactivity:** animate creatures are responsive to their environment and adapt their actions as necessary to the changes of the environment.

These properties relate to each other in complex ways, and in some respects are in tension with each other. The concept of purposefulness is bound up with autonomy—an object's actions will be *seen as* more autonomous if they are directed toward a goal rather than being driven solely by outside forces. Reactivity is a crucial part of purposefulness—a creature acting to achieve a goal cannot do so blindly, but must be able to respond to changing conditions in the world.

However, reactivity can be in tension with autonomy, since if the creature is reacting to outside conditions, then it *is* in some sense being driven by outside forces, rather than initiating action. For instance, a frog catching a fly with its tongue can be considered to have both reacted to the fly and to have initiated the action. The frog is the central actor in this version of the story. But it would be just as valid, if odd, to tell a story with the fly as the main actor, acting causally on the frog to cause its own demise. The frog in this version is still reactive, but its autonomy has disappeared. The presence of autonomy is particularly dependent upon point of view and the choice of starting points when describing a sequence of causal events.

Animacy, then, is more properly understood as a framework or way of thinking, rather than as a category. Animate thinking stems from a basic need to explain happenings and tell simple stories about them, and a need to fit things into roles in the stories as actors and objects of action. Scientific and mechanistic ways of thinking are in some sense attempts to get beyond these basic animistic tendencies, in that they tend to eliminate autonomy by searching for a cause for every action. But the tendency to describe the world in terms of autonomous actors is strong. Even scientists who should know better tend to try to find simple explanations and single actors rather than grapple with the distributed causality of a complex system (Keller 1985) (Resnick 1992).

3.3 Animacy and Computation

Animacy is a primary domain, in that it is not itself grounded in metaphor but in more basic processes. However, it can and does serve as a source domain for the metaphorical understanding of other realms. Obvious and fanciful versions of these metaphors are quite common in everyday life, as in utterances like “this bottle just doesn’t want to be opened”. Some mechanical systems are so complex that they need to be treated as having moods, particularly vehicles, and are thus anthropomorphized. I will refer to such usages as *animate metaphors*. Computer systems in particular are prone to be anthropomorphized, due to their complexity and apparent autonomy:

Anthropomorphization⁹ — Semantically, one rich source of jargon constructions is the hackish tendency to anthropomorphize hardware and software. This isn't done in a naive way; hackers don't personalize their stuff in the sense of feeling empathy with it, nor do they mystically believe that the things they work on every day are 'alive'. What is common is to hear hardware or software talked about as though it has homunculi talking to each other inside it, with intentions and desires. Thus, one hears “The protocol handler got confused”, or that programs “are trying” to do things, or one may say of a routine that “its goal in life is to X”. One even hears explanations like “... and its poor little brain couldn't understand X, and it died.” Sometimes modeling things this way actually seems to make them easier to understand, perhaps because it's instinctively natural to think of anything with a really complex behavioral repertoire as 'like a person' rather than 'like a thing'.

The computer bridges the animate and inanimate worlds as few other objects can. Although even humans can become subject to physical rather than intentional explanations, the computer, as a manufactured object designed around animate metaphors, inherently straddles the divide. People who interact with computers must do the same. Sherry Turkle has written extensively on children’s reactions to computers as “marginal objects”, not readily categorizable as either living or inanimate:

Computers, as marginal objects on the boundary between the physical and the psychological, force thinking about matter, life, and mind. Children use them to build theories about the animate and the inanimate and to develop their ideas about thought itself (Turkle 1984, p31).

But animate metaphors for computation can be problematic as well. AI, which might be considered an attempt to build an extended metaphorical mapping between humans and

⁹From the communally-written Hacker Jargon File, Version 3.0.0, 27 July 1993, http://fount.journalism.wisc.edu/jargon/jarg_intro.html

machines, impinges upon highly controversial issues in philosophy, and gives rise to contradictory intuitions, intense passions, and stubborn disagreements about whether computational processes can truly achieve intelligence (Dreyfus 1979), (Penrose 1989), (Searle 1980). The stereotypical AIs and robots seen in fiction are lifelike in some ways, but mechanical in others—inflexible, implacable, even hostile. Computers are said to lack certain key components of humanity or aliveness: consciousness, free will, intentionality, emotions, the ability to deal with contradiction. People exposed to computers will often end up defining their own humanness in terms of what the computer cannot apparently do (Turkle 1991). Something about the nature of computers, no matter how intelligent they are, seems to keep them from being seen as full members of the animate realm.

Our focus on animate metaphors allows us to sidestep this often sterile philosophical debate. Instead, we will examine the ways in which computational practice makes use of animism to structure itself.

Within the field of computation, anthropomorphic metaphors are sometimes denied as wrongheaded or harmful:

Never refer to parts of programs or pieces of equipment in an anthropomorphic terminology, nor allow your students to do so...The reason for this is that the anthropomorphic metaphor...is an enormous handicap for every computing community that has adopted it...It is paralyzing in the sense that because persons exist and act *in time*, its adoption effectively prevents a departure from operational semantics and, thus, forces people to think about programs in terms of computational behaviors, based on an underlying computational model. This is bad because operational reasoning is a tremendous waste of mental effort (Dijkstra 1989).

But most computer practitioners are not of the opinion that anthropomorphism (or operational thinking, for that matter) is such a bad thing. Indeed, the deliberate use of anthropomorphism has been a promising tool in computer education:

One reason turtles were introduced [into Logo] was to concretize an underlying heuristic principle in problem-solving—anthropomorphize! Make the idea come alive, be someone...Talking to inanimate objects and thus giving them life is an implicit pattern in our lives; we have tried to turn it to our advantage and make it an explicit process (Solomon 1976).

A common worry about anthropomorphic descriptions of computational system is the danger of causing overattribution errors. That is, people might draw incorrect inferences about the abilities of computers, assuming that they share more properties of the animate than they actually do, such as the ability to reason or to learn. Sometimes this is dealt with by instructing students to regard the little people as particularly dumb, mechanical instruction followers with no goals, judgment, or intelligence of their own, much like a player of the “Simon Says” game. In Papert’s phrase, “anthropomorphism can concretize dumbness as well as intelligence”.

This section explores the animate roots of computation and the ways in which the programming paradigms introduced in the last chapter make use of animate metaphors. We will also look at the relation of animism to human interface design, artificial intelligence, and the teaching of programming. The questions to be asked are which attributes of animacy are mapped into the computational domain, and onto what parts.

3.3.1 Animism at the Origins of Computation

The entire enterprise of computation might be seen as being built around a series of anthropomorphic metaphors, beginning with Turing's description of what are now called Turing machines. At this point in time, the word "computer" referred to a human calculator, who was destined to be replaced by the described machine:

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book... the behavior of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment... Let us imagine the operations performed by the computer to be split up into 'simple operations'... Every such operation consists of some change of the physical system consisting of the computer and his tape [a one-dimensional version of the squared paper]... The operation actually performed is determined, as has been suggested by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation.

We may now construct a machine to do the work of this computer (Turing 1936).

Here the anthropomorphism underlying computation is made quite explicit. Computation is first conceived of as a human activity, albeit one carried out in a rather formal and limited manner. The metaphorical mapping from the human domain has some questionable components—in particular, the characterization of the human computer as having a 'state of mind' that is discrete and drawn from a finite set of possible states. Nonetheless this metaphor has proven extraordinarily powerful, and forms the basis of the imperative model of programming described in section 2.2.3.1.

This prototypical computer, while described in animate terms, has few of the properties that are central to animacy. The computer is not particularly autonomous—it is "doing what it is told, no more, no less". Its purpose, if any, comes from outside itself—it has no representation or access to its goal. It has no relationship with a world outside of its own tape memory, so certainly cannot be said to be reactive. It is, however, repeatedly performing actions, and so despite its limitations is seen as animate. I call this particular reduced form of animism "rote instruction follower animism", to contrast it with the concept of animacy used in everyday life, which is linked to autonomy, purposefulness, and consciousness.

It is interesting to contrast this form of computational animism with the related imagery found in the discourse of computation's close cousin, cybernetics. Cybernetics and the technology of computation developed in parallel during and after World War II. Cybernetics concerned itself precisely with those properties of animacy marginalized by Turing's hypothetical machine and the later actual machines: purposefulness, autonomy, reactivity, and the importance of the relationship between an organism and its environment. These sciences had origins in different technological tasks (feedback control in the case of cybernetics, code-breaking in the case of computation), different mathematical bases (continuous time series vs. discrete symbolic manipulation), and ultimately in different ideas about the mind (regulative vs. cognitive). While the two approaches were discussed together during the period of the Macy conferences (Heims 1991), they ultimately went their separate ways. Recent developments in AI, such as the situated action approach to behavioral control (Agre and Chapman 1987) are in some sense an attempt to bring back into AI the cybernetic elements that were split off.

3.3.2 Animacy in Programming

The various programming models described in the last chapter can be analyzed in terms of animacy. This will mean looking at each in terms of what each model offers the programmer in terms of tools for thinking about action and actors. Like *trans*-frames, statements of programming languages can be considered as representations of change. If we assume that the operation of programs is understood, at least in part, by means of the application of such frames, we can ask ourselves questions about the contents of the terminals of such frames. What are the actions, what are the ACTORS and ORIGINS and DESTINATIONS? The identities of the ACTORS are of particular interest: who or what is making these actions happen, who is in charge? What images of control can inform our understanding?

To take an obvious example, consider an imperative assignment statement like **LET A=B**. In the above terms, it is a command to move a value OBJECT (contained in B) from a SOURCE (B) into a DESTINATION (A). This is essentially represented metaphorically as a physical transfer (PTRANS) of an object from one place to another, although neither the objects nor places are truly physical in any tangible sense. But what about the ACTOR terminal of the *trans*-frames that represents this action? Who is performing the move operation?

Imperative statements or commands like the one above are like imperative sentences—they do not explicitly name their ACTOR, instead they have an implied actor who is the recipient of the command. In the computational case, the computer itself will take on the role of ACTOR when it executes the instruction. This mode of address meshes with the image of the computer as an animate but dumb being, who must be instructed in detail and cannot really take any action on its own.

While object-oriented programming languages also follow a basically imperative model, the image of their operation is somewhat different, because the objects are available to fill ACTOR terminals. In OOP, programs are written as methods for specific object classes, and are seen as executing on behalf of particular objects. This tends to make the implied actor of an imperative instruction that occurs inside a method be the object, rather than the computer as a whole. The fact that methods can refer to their owning objects through reserved words like **self** heightens this effect.

Message-passing OOP languages introduce MTRANS or information-transmission events to the metaphorical underpinnings of the language. A message send is an action that transmits a request or order from one object to another (or to itself). OOP languages often use animistic names like **ask** or **send** for this operation. In a frame representation of a message-passing action, objects will occupy both the ACTOR and DESTINATION terminals. Once the message is transmitted, the DESTINATION object executes its own method for the message and, if it should take any action, will fill the ACTOR role of the next operation.

Message-passing languages thus provide a richer conceptual image of their activity than can be found in the basic imperative model, even though the computations they support may be identical. The activity is no longer the actions of a single actor, but a complex of interactions between objects. Even though, in an ordinary object-oriented language, the objects are not in any real sense active or autonomous (that is, they only take action when explicitly activated

from the outside), they provide better handles for programmers to apply animistic thinking. The partially animistic nature of objects creates natural “joints” that allow the programmer or program reader to carve up the activity of the program into small segments, each of which is more readily understood in animate terms. Whereas previously the computation was seen as a single actor manipulating passive objects, under OOP objects are seen as taking action on their own behalf.

Functional languages, by contrast, are those in which action, time, and change have all been banished for the sin of theoretical intractability. Such languages do not support animate thinking at all, and there are no implied actors to be found. Some programming environments that are quasi-functional, like spreadsheets, are amenable to a rather limited animate interpretation, in the sense that spreadsheet cells that have a functional definition can be thought of as actively “pulling in” the outside values needed to compute their own value. Spreadsheet cells are also responsive to change, and purposeful in that they act on their own to maintain their value in the face of change. But because they cannot act to change the world outside of themselves, they are not going to be understood as animate in any broad sense.

Procedural languages blend the imperative and functional models, and so admit to being understood as animate to varying degrees depending on the style of the program. The modularization of a program into a collection of procedures allows each procedure to be seen in animate terms; this is exemplified by the little-person metaphor, described below.

Constraint systems and languages vary in their treatment of actors and agency. Like the functional model, they emphasize declarative statements of relationships rather than specification of action, and so tend to be actorless. But some, like ThingLab, explicitly treat the constraints as objects with procedural methods. This suggests that the constraints themselves

paradigm	organizing metaphors and principles	treatment of agents
imperative	action, instruction-following	single implicit agent
functional	functional mappings	no agents
dataflow	flow of values through network	cells can be seen as “pulling” agents
procedural	society of computing objects; call and return metaphors; combines imperative and functional modes	procedures can be seen as agents; little-person metaphor
object-oriented	communication metaphors, message-sending, encapsulation	encapsulation helps to present objects as agents
constraint	declarations of relationships to be maintained	constraints can be seen as agents with goals.

Table 3.2: Summary of metaphors and agents in programming paradigms.

could take the role of actors in the system. But in practice, constraints are seen as passive data used by a unitary constraint-solver. The procedural side of constraints are single statements that are manipulated as data objects by a planner, rather than active entities in their own right.

Animacy is a powerful conceptual tool with which to analyze programming paradigms (see Table 3.2 for a summary). In particular, it can help explain the specific appeal and utility of object-oriented programming, which has in recent years become an extremely popular model for commercial programming. OOP is fundamentally just a way of organizing a program, and it has not always been clear why or even if it is a superior way to do so. Animism provides a theory about why OOP is powerful: its particular style of modularization divides up a program so that animate thinking can be readily applied to any of its parts. Functional programming, on the other hand, provides a model that systematically excludes animism, which might explain why, despite its undeniable theoretical advantages, it has little popularity outside of a small research community.

Animism lurks in the background of these classic programming models. Can it be used as an explicit basis for the design of new ones? Agent-based programming, to be described, attempts to do just that. In particular, it seeks to provide a language for programs that are understandable in animate terms, but with the ACTOR slot filled by objects that partake of a higher degree of animacy than the rote instruction followers found in the imperative and object models. They should be capable of being seen as possessing the main qualities of real animacy: purposefulness, responsiveness, and autonomy.

3.3.2.1 The Little-Person Metaphor

The little-person metaphor is a teaching device used to explain the operation of Logo to beginners. The little-person metaphor was invented by Seymour Papert in the early days of Logo development; the description here derives from the descriptions in (Harvey 1985) and (diSessa 1986). Under this metaphor, the computer is populated with little people (LPs) who are specialists at particular procedures, and “hire” other LPs to perform subprocedures. LPs are normally asleep, but can be woken up to perform their task. Whenever an LP needs a subprocedure to be run, it wakes up and hires an LP who specializes in that procedure, and goes to sleep. When the hired LP finishes executing its procedure, it reawakens the caller. A “chief” LP serves as the interface to the user, accepting tasks from outside the LP domain and passing them on to the appropriate specialists.

The LP metaphor is an explicitly animate metaphor for a procedural language. Procedures are still rote instruction followers, accepting commands from the outside and executing fixed scripts, and not capable of autonomous activity in any sense. Still, the LP metaphor succeeds quite well in “animating” procedures and making their activity understandable. In some respects it is easier to see a procedure as animate than the computer as a whole. I believe this is because a procedure, being specialized for a particular task, brings along with it a feeling of purposefulness. LPs thus can be seen as fulfilling a task rather than as simply carrying out a sequence of instructions. The social and communicative aspects of the metaphor are also important, since they give a metaphoric basis to the *relationships* between procedures.

The little-person metaphor has been quite successful as a device for teaching the detailed workings of the Logo language.¹⁰ Sometimes the model is taught through dramatization, with students acting out the parts of the little people. Not only does the model provide a good tangible model for the otherwise abstruse idea of a procedure invocation, but it turns them into animate objects, allowing students to identify with them and to project themselves into the environment.

3.3.3 Body- and Ego-Syntonic Metaphors

The Logo turtle was developed to encourage what Papert calls *syntonic learning* (Papert 1980, p.63). Turtles are said to be *body syntonic*, in that understanding a turtle is related to and compatible with learners' understandings of their own bodies. The turtle may also be *ego syntonic* in that "it is coherent with children's sense of themselves as people with intentions, goals, desires, likes, and dislikes". Syntonic learning, then, is any form of learning which somehow engages with the student's existing knowledge and concerns, in contrast to the more common style of *dissociated learning* (such as the rote learning of historical events or multiplication tables).

Papert's theory of syntonic learning resonates with our theory of metaphorically-structured understanding, which holds that all real conceptual learning involves building connections with existing knowledge. Looking at syntonic learning as a metaphor-based process might give us a way to think about how it works. Body syntonic learning involves creating a correspondence between the body of the learner and some anthropomorphic element in the problem world (the turtle). The learner is thus able to bring to bear a large store of existing knowledge and experience to what was an unfamiliar domain (geometry, in the case of the turtle). Metaphors based on the body have the advantage of universality: everyone has a body and a large stock of knowledge about it, even young children. The power of the turtle lies in its ability to connect bodily knowledge with the seemingly more abstract realm of geometry.

The utility of body-syntonicity for problem solving has also been explored by (Sayeki 1989). Sayeki explored how people project themselves into an imaginary scene such as the world of a physics or geometry problem, and found that problem solving times could be drastically decreased by including cues that allowed solvers to map the problem onto their own bodies (i.e. making what was a geometric figure resemble a body by the addition of a cartoon head). By introducing anthropomorphic forms (which he labels *kobitos*, or little people) he can apparently make it much easier for subjects to find the appropriate mappings from their own bodies to the objects of the problem, a process he refers to as "throwing-in". This is an interesting case of the deliberate deployment of metaphor in order to aid in problem-solving. It also highlights the active nature of the process of metaphorical understanding—learners actively project themselves into the problem domain, as is the case with the turtle.

¹⁰ Seymour Papert, personal communication.

The realization of body-syntonicity through turtles and similar physical or graphic objects that permit identification would seem to be a clear success. Ego-syntonicity, however, is more problematic. The Logo turtle is only weakly ego-syntonic. A turtle always has bodily properties (position and heading), but does not in itself have goals and intentions of its own, or behaviors that take into account the environment (Resnick and Martin 1991). If it has any kind of ego, it is the simple-minded one of the rote instruction follower. And this of course does allow the learner to project some of their own mental activity onto the turtle, but only a highly limited subset. The turtle can, in theory, be programmed to have ego-syntonic properties like goals, but they are not inherent in the basic Logo turtle or the Logo environment. In particular, in many implementations of Logo the turtle lacks any kind of sensory capability and thus cannot really have goals because there is no way to verify when the goals are satisfied. Later work by Papert and his students (Papert 1993) (Martin 1988) addresses the issue of augmenting the turtle with sensors so that it could have responsive behaviors and gave more emphasis to the role of feedback. However, the Logo language is still essentially oriented around the procedural model, rather than a reactive or goal-oriented one.

There are strong emotional factors at work in this process of projection and identification. Child programmers may have different styles of relating to the computer that affect how or if they can identify with and project themselves onto the machine. Such styles are strongly conditioned by gender and other social factors (Turkle 1984) (Papert 1993). It has been suggested, for instance, that the child-rearing role of women predisposes them to “take pleasure in another’s autonomy” while men are more likely to be preoccupied with their own autonomy and thus more prone to domination and mastery rather than nurturing the autonomy of others (Keller 1985). This difference in style has been observed in the different programming styles of boys and girls learning to program (Motherwell 1988) (Turkle and Papert 1991). In Motherwell’s study, girls were considerably more likely to treat the computer as a person than were boys.

3.3.4 Anthropomorphism in the Interface

Human interface design has considered anthropomorphism in the form of “agents”. In this context, an agent is an intelligent intermediary between a user and a computer system, visualized as an anthropomorphic figure on the screen with whom the user interacts, often by means of natural language (Apple Computer 1991). Interface agents are still mostly a fantasy although there are some explorations of anthropomorphic interfaces with minimal backing intelligence (Oren, Salomon et al. 1990), as well as efforts to use more caricatured, less naturalistic anthropomorphic metaphors that use emotion to indicate the state of a computational process (Kozierok 1993).

There is a long-standing debate in the interface community about the utility and ethics of agents (Laurel 1990). Anthropomorphic agents promise easier to use interfaces for novices; but they also threaten to isolate the more experienced user from the ability to directly control their virtual world (Shneiderman 1992) (Lanier 1995). Other ethical issues revolve around the question of whether having a computational system present itself as a mock person requires or elicits the same sort of moral attitudes that apply to a real person, along with the even more

provocative notion that dealing with simulated people who can be abused (deleted or deactivated, for instance) will lead to similar callous attitudes towards real people.

Nass demonstrated that people interacting with computers are prone to treat computers as social actors, when interaction is framed appropriately (Nass, Steuer et al. 1993). That is, if the interaction is framed as a conversation, users will apply the usual social rules of conversation to the interaction (i.e., praising others is considered more polite than praising oneself). The subjects of the experiments applied these rules even though they believed that such rules were not properly applied to computers. This research suggests that people can be induced to take an animate view (in Goffman's terms, to apply a social framework) by certain specific cues, such as voice output. This cueing process seems to operate almost beneath conscious control, much as the motional cues used by Michotte to induce perceptions of animacy.

Many interfaces are not as explicitly anthropomorphic as those above, but incorporate a few cues that induce a mild degree of animism. The Logo language, for example, is designed to use natural-language terms in a way that encourages a form of interface anthropomorphism. Commands, syntax, and error messages are carefully crafted so that they resemble natural language communication, casting the Logo interpreter, procedures, or turtle as communicating agents. For instance, if a user gives Logo an unrecognized command (say SQUARE), it responds with "I DON'T KNOW HOW TO SQUARE", rather than something on the order of "Undefined procedure: SQUARE". The student can then "instruct" Logo by saying:

```
TO SQUARE  
  REPEAT 4 [FD 70 RT 90]  
END SQUARE
```

The interesting detail here is the syntax for procedure definition makes the definition resemble instruction in natural language: "To [make a] square, do such-and-such...". By putting the user in the position of addressing the computer (or the turtle) in conversation, the system promotes the use of animate thinking. This syntactical design allows Logo teachers to employ the metaphor of "teaching the computer" or "teaching the turtle" to perform procedures (Solomon 1986).

Consider the different shadings of meaning present in the following different ways of expressing a textual error message:

- 1 "Missing argument to procedure **square**."
- 2 "Procedure **square** requires more arguments."
- 3 "Procedure **square** needs more arguments."
- 4 "Procedure **square** wants more arguments."

Of these, message 1 is the most formal, and the least animate. It is not even a sentence, just a declaration of a condition. Message 2 casts the procedure as the subject of a sentence, which is a large step in the direction of animacy. Messages 3 and 4 are of the same form as 2, but alter the verb so as to imply increasingly human attributes to the procedure. All these variants and more occur in various places in the discourse of computer science and in the discourse-like interactions between programmers and their environments. Most professional programming environments use messages that are like message 1 or 2, while those like message 3 are found in

some Logo environments in keeping with the language's use of animate and natural-language constructs (see below). Statements like message 4 are rarely used as error messages, but are common in the informal discourse of programmers.

Consider also the messages of the form "I don't know how to **square**", issued by some Logo environments to indicate an undefined procedure. The use of the first person makes these more explicitly anthropomorphic than the earlier group. But since there is no procedure, the "I" must refer to the computer as a whole, or the interpreter, rather than a specific part. These presentational subtleties are indicative of subtle modulations of animism in the Logo environment: the interpreter/turtle is fairly heavily animate, to the point where it can refer to itself in the first person, whereas procedures are less so: they have "needs", but do not present themselves as agents, instead letting the interpreter speak for them.

In recent years the term "agent" has become overused in the commercial software world almost to the point of meaninglessness. Still, some core commonalities appear among most products and systems touted as agents. One of these meanings is "a process that runs in the background, autonomously, without being directly invoked by a user action". Even the simplest programs of this type, such as calendar and alarm programs, have been dignified as agents. Less trivial applications include programs that try to learn and automate routine user actions (Kozierok 1993) (Charles River Analytics 1994).

This usage seems to derive from the overwhelming success of the direct manipulation interaction paradigm. Because of the current dominance of this model of computer use, any user application that operates outside of the direct manipulation mode, even something as simple as an alarm clock, becomes an agent if only by contrast.

Another popular type of "agent" is the migrating program, such as "Knowbots" that roam the network looking for information that meets preset criteria (Etzioni and Weld 1994). Agents of this sort can transport themselves over a network to remote locations where they will perform tasks for their user. Special languages, such as Telescript (General Magic 1995) are under development to support the development and interchange of these agents. It is not clear to me why a program running on a remote computer is any more agent-like than one running on a local computer. The animism apparently arises from the ways in which these programs are intended to be used. Remote execution might contribute to a greater feeling of autonomy, and agent programs can migrate from machine to machine, like a wandering animal, and reproduce by copying themselves. The development of computer networks has given rise to spatial metaphors (like the overused "cyberspace") which in turn seems to encourage the use of animate metaphors for the entities that are to populate the new space. Telescript in particular features *places*, which are similar to processes, as one of its main object types.

3.3.5 Animate Metaphors in Artificial Intelligence

The task of Artificial Intelligence is to derive computational models of human thought, a process that is metaphorical in nature but maps in the opposite direction from the one we have been considering here, that is, the use of animate metaphors for computation. The two ways of knowing are intimately related, of course: metaphorical mappings of this broad sort tend to be

bidirectional, so that models of the mind and of the computer are necessarily co-constructed in each other's image. I will not directly address the issues raised by computational metaphors for mind, a topic addressed widely elsewhere (see (Agre 1996) for a view that emphasizes the specific role of metaphor).

However, mental or animistic metaphors for computation are just as much a part of AI as are computational metaphors for mind. From this standpoint, AI is less a way of doing psychology and more an engineering practice that relies on the inspiration of human thought for design principles and language. In other words, it makes explicit the animism that has been implicit in other approaches to computation. For example, here is a fairly typical description of the operation of an AI program, selected at random from a textbook:

When FIXIT [a program] *reexamines* a relation, it *looks* for a way to *explain* that relation using all but one of the precedents... [the exception] is omitted so that FIXIT can *explore* the hypothesis that it *provided* an incorrect explanation... (Winston 1992, p390) [emphasis added].

The italicized words above indicate verbs that put the program in the position of an animate actor. Such talk is probably necessary, in that it is the most compact and meaningful way to communicate an understanding of the program to the reader.

AI has always pushed the frontiers of complexity in programming and thus has generated many new metaphors for computational activity, metaphors which are not necessarily linked to the founding metaphor of computation as thought. An example of this would be the blackboard model of problem solving (Reddy, Erman et al. 1973), in which a blackboard metaphorically represents a scratch memory and communications medium that can be shared by multiple computational processes.

The willingness of AI to freely leap between computational and animate language has proved both fruitful and dangerous. The fruitfulness results from the richness of animate language for describing processes, and the inadequacy of more formal language. The danger results from the tendency to take one's own metaphorical language too literally. (McDermott 1987) is a well-known diatribe against the practice of giving programs animistic names like UNDERSTAND rather than more restrained, formalistic names based on the underlying algorithm. Such naming practices, however, are a constituent part of the practice of AI. Constant critical self-evaluation is the only way to ensure that the metaphors are used and not abused.

The subfield of distributed artificial intelligence (Bond and Gasser 1988), in which AI systems are split into concurrent communicating parts, makes more explicit use of anthropomorphic metaphors to represent its parts. Because communication often dominates the activity of such systems, metaphors of communication and social interaction predominate. The metaphors employed include negotiation (Davis and Smith 1983), market transactions (Malone, Fikes et al. 1988), and corporations or scientific communities (Kornfeld and Hewitt 1981). A particularly important thread of work in this field begins with Hewitt's work on Actors, a model of concurrent computation based on message passing (Hewitt 1976) (Agha 1986). The Actor model is a computational theory, in essence a model of concurrent object oriented programming, but the work on Actors has included a good deal of work on higher-level protocols and on ways of organizing communication between more complex actors. This has led to an interest in "open systems" (Hewitt 1986), an approach to thinking about

computational systems that acknowledges their continual interaction with an external environment.

Minsky's *Society of Mind* represents perhaps the most figurative use of anthropomorphic language in AI, perhaps out of necessity as it is a semi-technical book written in largely non-technical language. By using anthropomorphic metaphors to talk about parts of minds, Minsky risks being accused of resorting to homunculi, and indeed has been (Winograd 1991). Dennett has a lucid defense of homuncular theories of the mind:

It all looks too easy, the skeptics think. Wherever there is a task, posit a gang of task-sized agents to perform it — a theoretical move with all the virtues of theft over honest toil, to adapt a famous put-down of Bertrand Russell's.

Homunculi — demons, agents — are the coin of the realm in Artificial Intelligence, and computer science more generally. Anyone whose skeptical back is arched at the first mention of homunculi simply doesn't understand how neutral the concept can be, and how widely applicable. Positing a gang of homunculi would indeed be just as empty a gesture as the skeptic imagines, if it were not for the fact that in homunculus theories, the serious content is in the claims about how the posited homunculi interact, develop, form coalitions or hierarchies, and so forth (Dennett 1991, p261).

3.3.6 Conclusion: Computation Relies on Animate Metaphors

We have seen that animate metaphors and usages play a number of roles in computing: they appear in the discourse of computation as foundational metaphors, in pedagogy as teaching devices, and are taken with various degrees of literalism in interface design and AI. Fundamentally, computers are devices that *act*, and thus we are prone to see them in animate terms.

But in most cases the animism attributed to the computer is of a limited kind, the “rote instruction-follower” kind of animism. In some sense, this is proper and unexceptional—after all, the computer *is* a rote instruction follower, at the level of its inner workings. Students must learn to give up their animism and look at the computer as it “really is”, that is, under the mechanical viewpoint, as a device without goals or feelings, if they are to understand its operations in detail.

But what if our goal is not to teach how machines work, but particular styles of thought? If our interest is in exploring programming as a means of controlling action and of building animate systems, the fact that the language of computation provides only a narrow concept of animacy is unfortunate. With that in mind, the next section looks at the idea of agents as the basis for programming languages that can support a richer notion of animacy.

3.4 Agent-Based Programming Paradigms

This section develops a theory of agent-based programming, that is, a model of programming strongly based on animate metaphors. Agent-based programming begins with the traditional anthropomorphic mapping underlying the idea of computation and changes it in two ways. First, it changes the nature of the mapping so that instead of presenting the computer as a single anthropomorphic entity, it is instead seen as a society of interacting autonomous

agents. Second, it extends the mapping between the anthropomorphic representation and the underlying computational reality to include new characteristics, such as goals and emotions based on the state of goals.

Using multiple applications of the anthropomorphic metaphor in the description of a program is not a particularly new idea. It is present, in varying degrees of explicitness, in the little-person model, in object-oriented programming, in the Actor model of computation, and in the Society of Mind. However, in these systems, the metaphor has for the most part remained in the background, used only for teaching purposes or to illustrate a set of concepts that quickly become detached from their metaphorical roots. The animate parts in such systems are for the most part seen as rote instruction followers.

In agent-based programming, the animate metaphor is taken more seriously. Anthropomorphic characteristics are deliberately brought forward into the language and interface, and the user is encouraged to think of agent activity and interaction in those terms. More importantly, agent-based programming seeks to provide a richer form of anthropomorphism than earlier models. In contrast to the rote-instruction-follower version of animacy found in most of the standard models of computation, the animacy of agent-based systems attempts to emphasize the characteristics of animacy we have found to be the most important: autonomy, purposefulness, and reactivity.

We will see that the single most important technique for realizing this concept of agents is to give them explicit goals—that is, concrete and accessible representations of what it is they are supposed to accomplish. Goals can help crystallize the anthropomorphic metaphor: an agent is not just any sort of person, it's a person with a job to do. Goals can support reactivity (agents react to events that affect their goal) and support detection of conflicts when one agent interferes with another's goal.

3.4.1 Principles of Agent-Based Programming

In the loosest sense of the term, an agent-based system is simply one whose functionality is distributed among active functional modules. If a module can be seen to be taking an action to achieve a purpose, it partakes of agenthood. By this criterion, even the most ordinary programs written in standard procedural languages may be seen as agent-based: agents are procedures, while the communications between them are procedure calls. So this definition does not capture anything very distinctive, although it does allow us to see that the roots of the agent metaphor are to be found in standard programming practice.

How can we design agents that are more likely to be seen in animate terms? We have identified three characteristics in particular that contribute to the feeling of animacy: purposefulness, autonomy, and reactivity. These characteristics do not have simple technical definitions, and are deeply intertwined with one another. The following sections explore how these characteristics can be realized in concrete computational terms. Other criteria must be taken into consideration when designing an agent language. We would like our agent system to be at least as powerful and usable as more traditional languages. This means that we also need to consider tradi-

tional language design issues such as understandability, expressiveness, modularity, and composability.

Agent-based programming, by its nature, emphasizes action-taking rather than computation in the narrow sense. Because of this, it is crucial that agents be embedded in some kind of environment in which they can act on other active or passive objects. The environment, then, will be just as important as the language in determining the nature of agent-based programming, since it determines the kinds of action and interaction that can take place. Chapter 4 treats these issues in detail; for now, we should just keep in mind that agents are involved in interactions with a complex world.

3.4.1.1 Purpose, Goals, and Conflict

When we talk about a “goal”, we mix a thousand meanings in one word.
— Marvin Minsky, *Society of Mind*

In a complex software system, every component is there to achieve some goal of the designer. This goal may be evident to an expert familiar with the global workings of the system and the relationships of the parts, but may not be at all evident to a novice attempting to understand the system from scratch. In fact, even experts often have trouble understanding the purpose of other’s code, and occasionally even their own.

Agent-based programming can make these underlying purposes explicit, by including a computational representation of the state of affairs the agent is supposed to bring about. Agents allow procedural programs to be associated with declarative information about what they do. In the same way that objects in a dynamic language like Lisp can carry around their type with them, agents can carry around their goals. Explicit goals serve a number of different purposes in an agent system:

Control: The most basic use of goals is simply to control and structure activity by triggering procedures at appropriate times. A goal tells an agent when to run—that is, when its goal is unsatisfied.

Verification: Goals allow agents to monitor the success or failure of their actions—successful actions make the goal become satisfied. This can make debugging easier, since parts of programs that fail to achieve their stated purpose are explicitly revealed. The fact that agents can have an idea of success and failure also provides a way to deal with unreliable problem-solving techniques. Agents may create several subagents to help achieve its goal, each subagent having the same goal as its superior, but using different methods of achieving it. Because the agents can tell whether or not they have succeeded, the superior agent can use this knowledge to dispatch among its subagents.

Conflict detection: If an agent achieves its goal, it can continue to monitor that goal and detect if some other agent causes the goal to no longer hold. This is one way to detect inter-agent conflict. Conflict situations are quite common in complex systems, and are a common source of bugs. The presence of explicit goals permits an agent to detect situations in which another agent interferes with its goal, and makes it easier to convey to the user the nature of the problem.

Organization: Agent-based programming can be viewed as a particular way of organizing program fragments, by analogy with object-oriented programming. Just as OOP organizes procedures around the objects that it manipulates, in agent-based systems procedures can be viewed as organized around the goals they are meant to achieve. Each of these forms of modularity provides a degree of encapsulation or black-boxing, that is, a separation between the external interfaces of objects and their internal implementations. In OOP, this means that an object can send another object a message without having to know about the internal structure of the receiving object or the methods that process the message. The analogous property for agent-based systems is that an agent can assert a goal to be satisfied without having to worry about the details of the other agents that will be recruited to satisfy that goal

Visualization: Explicit goals can also make programs more comprehensible to an observer who is reading the program or watching it execute. The goals serve as annotations or comments that explain in a declarative style what a procedure is trying to do. But unlike textual comments, goals are in a machine-usable form and can thus be used for a variety of interface tasks.

Goals thus form the conceptual and computational basis for giving agents explicit representations of their purpose, which also contributes to the realization of the other two major qualities of animacy we seek: autonomy and reactivity.

3.4.1.2 Autonomy

In the standard procedural model of programming, programs run only when they are explicitly invoked by an outside entity, such as a user or another program. Agents, by contrast, should be able to invoke themselves, without the need for external invocation. That is, they should be autonomous in the sense we have been using the word in this chapter—capable of initiating action, or being seen as such. Of course, since an agent is also a constructed mechanism, one which the programmer builds and understands using a more mechanical, causal point of view, this apparent autonomy may only be skin-deep. No action really happens all by itself. So agents will always have to be capable of being seen in at least two ways: as autonomous action takers and as mechanical responders to outside conditions.

The simplest way to realize agent autonomy is to make each agent into a separate process, and let it run in a loop concurrently with all other agents (and other activity, such as user actions). Concurrency in some form is a minimal requirement for agent autonomy. But concurrency by itself is a bare-bones approach to constructing agents. It provides a necessary platform to support autonomy, but no structure for handling conflicts or explicitly dealing with goals.

A somewhat different approach is to allow agents to be triggered by specific events or states. That is, agents specify somehow what states or events they are interested in, and are activated whenever these occur, without having to continuously check for them. This has the same effect as the concurrent approach, but it can be more efficient and the metaphor is different. In some sense triggering is causal or non-autonomous, because the agent is being activated by an outside force, but if we use the monitoring metaphor the agent may still be seen as the source of action.

3.4.1.3 Reactivity

Reactivity means that agents should be responsive to the world. But what does this mean? For one thing, it implies that the agent is operating in a changing world—an assumption at odds with the standard model of procedural programming, where the environment is generally static. It also implies that the agent has some readily available means of sensing the state of the world. More importantly, it means that the image of the agent is different from the sequential executor that the rote-instruction-follower model suggests. While agent-based systems should certainly be capable of executing sequences of operations, it should also be easy for them to respond to changes in the world at any time.

Reactivity is related to the properties of autonomy and purposefulness. An autonomous entity initiates action—but to seem purposeful must have some *reason* for taking action. For simple systems, the reason is usually going to be a change in the state of the world, that is, the entity is *reacting* to something in its environment.

3.4.2 Computational Realizations of Agents

This section examines some of the ways in which the term “agent” has been used to refer to elements of larger computational systems, and analyzes their strengths and deficiencies compared to the criteria outlined above.

3.4.2.1 Agent as Process

The simplest realization of computational agents is as concurrent processes that can execute simple loops. This approach is used by MultiLogo (Resnick 1988). In MultiLogo, agents are processes that can execute any procedure. Agents are thus general purpose entities rather than being specialized for particular functions. Each agent has a graphic turtle of its own, so turtles are controlled by a single agent, but the agents can communicate freely with the sensors and effectors of LEGO creatures, permitting experiments in which multiple agents can be controlling a single creature. Agents communicate by sending messages to one another. Each agent has an input queue for messages and has some control over how messages are inserted into the queue of recipients.

MultiLogo agents, being separate processes, are certainly autonomous. The language provides no special constructs for supporting goals or reactivity, although it is a fairly simple matter to set up procedures with conditionals that provide these kinds of features. There are no built-in facilities for handling conflict: if two agents try and control the same LEGO motor, contention will arise. Agents, however, are protected from conflicting commands by handling them through a queue, which effectively serializes them.

One major difference between MultiLogo’s approach and the agent-based systems of LiveWorld is the sharp distinction in the former between agents and procedures. This follows standard computing practice, but led to some confusion among students learning to use MultiLogo, who had to learn to perform two separate types of problem decomposition. In effect, agents (processes) are a separate facility for concurrent operation that is augmenting a

standard procedural language (Logo). Perhaps the confusion arises because both concepts partake of animacy in different ways, and get conflated in the students' minds.

3.4.2.2 Agent as Rule

Another model for agent is a rule or demon. In this model, agents have a condition and an action to be taken when the condition is met. A collection of such agents, together with a procedure for resolving conflicts, forms a forward-chaining rule system, that is, a system that is driven solely by the current state of the world rather than by goals. While agents may have a purpose, they do not have explicit goals. Telemorphic programming (Nilsson 1994) is an example of a formalism that uses sets of rules to implement agents; for more discussion see Section 5.1.3.

Although the agent-as-process model is more general (since a process can presumably implement a rule using conditionals), the more restrictive format of the agent-as-rule model can simplify conflict resolution and makes possible certain control strategies (such as recency or specificity) and efficient implementations (such as storing rules in a data structure that permits fast triggering (Forgy 1982)).

Rules, considered as a programming metaphor, are quintessentially reactive but are not goal-oriented and are not usually thought of in particularly anthropomorphic or autonomous terms. One way in which rules are anthropomorphized is to think of them as *demons* that are triggered by specific events or conditions.

3.4.2.3 Agent as Enhanced Object

Shoham's Agent Oriented Programming (AOP) formalism (Shoham 1993) is a computational framework explicitly designed as an extension or specialization of object-oriented programming. Objects become agents by redefining both their internal state and their communication protocols in intentional terms. Whereas normal objects contain arbitrary values in their slots and communicate with messages that are similarly unstructured, AOP agents contain beliefs, commitments, choices, and the like; and communicate with each other via a constrained set of speech acts such as inform, request, promise, and decline. The state of an agent is explicitly defined as a *mental state*.

The AOP concept of agent is coarser-grained than that found in LiveWorld's agent systems. Each agent has its own mentality or belief system as opposed to components of a single mind. There is no real opportunity for conflict in AOP, because each agent is responsible for maintaining consistency among its commitments. If given contradictory information a choice must be made, but there is no built-in mechanism for doing this.

Oddly enough, the AOP formalism appears to have no explicit representation of desire or goals. Nor do the languages that implement AOP provide processes or other constructs that support autonomous activity. Agents, despite their mental qualities, are essentially passive like ordinary objects, but communicating at a higher level of abstraction. The agency in AOP is entirely *representational*, rather than active. The roots of agency are seen as having explicit beliefs and knowledge, rather than in being goal-directed or autonomous.

This makes Shoham's concept of agency almost entirely orthogonal to LiveWorld's. They are not necessarily incompatible though—there is the interesting possibility that it might be possible to combine Shoham's representational agency with the purposive, autonomous idea agency developed here. This is an area for future research.

3.4.2.4 Agent as Slot and Value-Producer

Playground (Fenton and Beck 1989) had a rather idiosyncratic view of agents as a combination of a slot and a value-producing form for that slot. An agent in this system is something like a spreadsheet cell that is also part of an object. See Section 2.2.3.2 for further discussion of Playground.

Playground's model of agent seems rather un-agent-like in the context of our discussion, in that it cannot take any action to effect the world outside of itself. Functional agents like these can be anthropomorphized to some extent by describing their computation as a "pulling" of values from other cells in the system to use in its computation of its own value. This is opposed to the usual "pushing" activity of more imperatively-structured languages. This terminology at least puts the agent in an active role, but does not really allow it to act on the world.

A Playground agent is autonomous, in the sense that it continuously updates its value without outside intervention, and reactive for the same reason. It has no explicit representation of goal or purpose. Since there is exactly one agent per slot, there is no real opportunity for conflict to arise.

3.4.2.5 Agent as Behavioral Controller

A wide variety of systems for controlling the behavior of an intelligent creature employ distributed networks of simple modules. Some of these refer to the modules "agents", but it is more common in this realm for the creatures themselves to be called agents. To avoid confusion we will continue to use agent to refer to the *internal* active parts.

These systems arise from a movement within the AI and artificial life fields that focuses on intelligent action rather than the more traditional symbolic problem solving. In part this focus stems from frustrations with the state of work in traditional paradigms, in part from an interest in modeling animal behavior for its own sake, in part from philosophical standpoints about the nature of human intelligence and activity (see section 2.1.4), and in part by a search for alternatives to the dominant planning and reasoning paradigm, which can be computationally intractable and has a limited ability to deal with a dynamic world (Chapman and Agre 1986).

While situated action does not rule out representation and reasoning, it de-emphasizes them. Intelligent action is generated first by responsiveness to current conditions in the world. Systems that work in this way tend to be organized around a functional modularity. Rather than the centralized architecture of traditional AI, these systems are divided up into more-or-less independent, modules each of which is in charge of a particular behavior. These modules are connected in networks so that some control can be exerted, but each module is in charge of managing its own connection to the world through sensors and effectors.

A variety of computational constructs are used to implement the internal agents of these situated creatures. I will survey a few of these systems; for a fuller treatment see (Agre 1995).

The subsumption architecture (Brooks 1986) (Connell 1989) is a design methodology for robot control systems. A subsumption program consists of a number of modules connected in a network, usually arranged in a layered fashion, with each layer capable of controlling action independently. Higher-level layers are capable of preempting lower-level ones using a scheme based on fixed priority gates that make up the network. Each module is an autonomous augmented finite-state machine, which can communicate with the outside world through sensors, to other modules through the network, and to itself through a small set of registers. Modules typically implement fairly simple behavioral control rules, sometimes augmented with state. Goals are implicit rather than explicit, and conflict between modules is handled by hardwired priorities in the connections between the modules and the network.

The Agar system is a software kit for modeling animal behavior (Travers 1988), and its conception of control is based loosely on Tinbergen's ethological models (Tinbergen 1951) (also see section 5.2). In Agar, an animal's behavioral control system consists of a network of agents. An agent is something of a cross between a process and a rule, and has an activation level and a threshold that determines when it can run. Agents in general connect a sensory stimulus to an action, which may be an external action or the activation of another agent. Agents affect one another by passing activation in this manner, and the usual practice is to arrange them into a control hierarchy so that higher-level agents can activate ones at lower levels. There are no explicit goals. Conflict is avoided by hard-wired inhibitory links between agents that conflict.

Maes' Action Selection Algorithm (Maes 1989) (Maes 1990) is somewhat different from the above systems. It too posits a division of a control system into functional modules, with variable activation levels. However, the relations between them are modeled after the relations of classical STRIPS-like planning operators, so that modules are connected to other modules that represent their preconditions and expected results. Unlike the other models, this allows the activation of the system to respond to changing goals as well as to conditions in the world. The action selection algorithm is explicitly non-hierarchical so that there are no managers, however, there is a "bottleneck" in that only one module can actually act at any one time.

The agents of these systems are highly limited, yet can perform complex tasks. As agents in the sense of this chapter, they are somewhat deficient. All three systems offer a high degree of reactivity. Agar's agents and Brooks' behavior modules are semi-autonomous: they run concurrently, and take action on their own. Maes' competence modules are less so, since only one can act at a time. All the architectures implement purposefulness, but only Maes' uses an explicit representation of goals.

3.4.3 Agents and Narrative

Narrative deals with the vicissitudes of intention.

— Jerome Bruner (Bruner 1986)

The agent metaphor allows computational elements to be envisioned using more natural and powerful conceptual tools than those provided by more formalistic approaches. But thinking about program parts in animate terms is only the first step. Once that mapping is created, what other uses can be made of it? One possibility is to explore the traditional methods for expressing the interactions and histories of human agents; that is, of narrative forms. There would seem to be great potential in using the form of stories both to create software agents (through programming-by-example or case-based techniques) and to present the activity of such agents to the user.

This second technique was explored in prototype form in the Nose Goblins system (Travers and Davis 1993). This project included an agent architecture that was one of the predecessors of LiveWorld's, and allowed users to construct constraint-like agents that performed tasks in a graphic design domain. In this architecture, agents related goals and methods, and could detect inter-agent conflict.

When an agent conflict was detected, Nose Goblins created a story about the conflict and expressed the story to the user using a format that combined a comic-strip/storyboard format with clips of animated cartoon characters (see Figure 3.1). The storyboard served to present the temporal qualities of the story, while the animated characters presented the emotional states of the actors, based on the satisfaction state of their goals (happy, unhappy, or angry if in a conflict situation). When a conflict was detected, a partial storyboard was generated with the final panel showing the characters in an angry state; it was then up to the user to resolve the situation. The resolution was stored in a case library and used to deal with future conflicts.

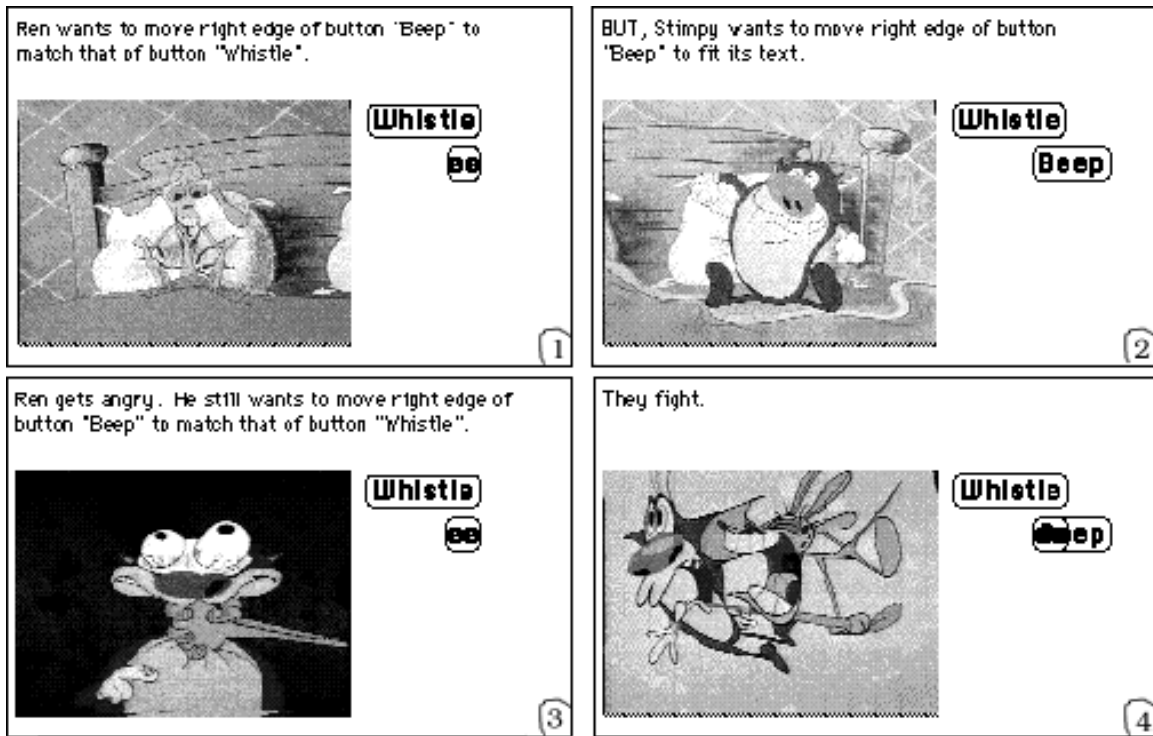


Figure 3.1: Four panels from a Nose Goblins storyboard. The first two frames illustrate the goals of the two agents involved, while the last two illustrate how one agent clobbers another's goal and the resultant conflict.

Narrative is a natural form for describing action, particularly action that involves the interaction of multiple actors. If computational objects are going to be understood anthropomorphically, their actions are going to be best understood through narrative forms.

3.5 Conclusion

This chapter has defined and explored a particular mode of thought and its relationship to computation. I imagine that some readers might have found this discussion to be highly irritating. "Why all this attention to such a marginal concept?", I hear them saying. "Don't you know that anthropomorphism is at best a useful pedagogical device, and not something that really has anything to do with how computers work and are programmed?" Assuming any of those readers have actually made it to the end of the chapter, I would like to answer them. In computation, animism has become what I have called a transparent metaphor, one that is so much a part of the structure of the discourse of the field that we have forgotten that it is there. This chapter is an attempt to see a metaphor that has become close to invisible, with the hope of understanding and extending it.

Is there a need for this treatment? I think that there is, because of the confusion that rages in various segments of the computer world over issues that are deeply tied to the issue of just what it means to be animate and/or intelligent. The current fad of using the word "agent" for software of all types is one aspect of this unease, and it generates a good deal of confusion. Another symptom is the recent split in AI between representationalists and the situated action

approach, reminiscent of the older split between AI and cybernetics. Mature science is supposed to converge on a theory; but instead computational approaches to intelligence and life seem to be diverging, with no consensus on the problems to be solved, let alone the answers.

This chapter, then, is an attempt to address these problems at a conceptual level, by trying to understand how the languages of computation are related to the language used to describe life and action. We found that while animate metaphors are pervasive in computation, being in some sense a founding metaphor for the field, the metaphors offer up only a limited form of animacy that leaves out many of the key properties associated with aliveness. The rote-instruction-follower image affects and infects our thinking and limits our imagination about what computers can do.

Agent-based programming systems set out to bring a fuller form of animacy to the description of computational processes. By picturing the components of programs as animate objects that can have explicit goals, a degree of autonomy, and the ability to dynamically react to their surroundings, languages can be designed that are more suitable for creating animate worlds. The following chapter illustrates some agent systems based on these ideas and explores what they can do.

Chapter 4

LiveWorld

*If the computer is a universal control system,
let's give kids universes to control.*

— Theodore H. Nelson (*Nelson 1974*)

The substrate of LiveWorld, upon which the agent systems were constructed, constitutes a general purpose computational medium, designed for liveliness and ease of construction. LiveWorld consists of a tightly integrated object system and interface, which permits ready manipulation of the various components needed to make animate worlds. In addition to making the animated graphic objects or actors easy to create, manipulate, and control, it was necessary to make the internal parts and operations of the objects and the programs that drove them visible and manipulable as well. While LiveWorld itself is not based on agents or animate metaphors, it provides a world in which languages and programs based on those metaphors can be developed.

Note: this chapter goes into a good deal of detail about the inner workings of LiveWorld. Readers who are interested only in the agent systems may wish to skip over all but the first section of this chapter. The first section is an overview which introduces enough of LiveWorld's terminology to make the next chapter understandable.

4.1 Overview of LiveWorld

LiveWorld's virtual space consists of a hierarchical structure of objects called *boxes* (see figure 1.1). Boxes can be created and manipulated with the mouse; they are concrete and tangible graphic representations of underlying computational objects. The intent of the interface is to make the entire computational state of the virtual world available to the user in the form of boxes.

Boxes are named and can have values. If a box **color** is inside of a box **turtle**, then **color** is sometimes referred to as an *annotation* of **turtle**. Annotations can function like slots of standard object-oriented languages, so in the above example the value of the color box will be interpreted as a slot of the turtle. Boxes are sometimes referred to as *frames*, reflecting their underlying representation.

Some boxes represent graphic objects; these are called *actors*. Actor boxes are contained inside theaters, which present them in two separate views: a *stage* view which shows them in graphical form, and a *cast* view which shows them in the form of boxes. These two views are interconstrained so that changes in one are reflected in the other.

Boxes can inherit properties from a *prototype*. New boxes are often made by *cloning* an existing box, which just means making a new box with the original as prototype. Cloned boxes (also called *spinoffs*) inherit various properties from their prototype, such as values for slots. Inheritance is dynamic, meaning that changes in the prototype may affect the spinoff. The initial

LiveWorld environment consists of specialized objects including theaters, actors, slots, sensors; users usually begin the programming process by cloning and combining these into a world of their own. After an object is cloned the new object may be customized by modifying any of its properties to be different from its prototype.

LiveWorld is programmed using an extension of Common Lisp. Since boxes are named and are in a hierarchy, they can be referred to by paths. There are two forms of box paths. The absolute form gives the name of a box starting from the top, or *root*, of the hierarchy. For instance, the path

```
#!/theater-1/cast/turtle/color
```

names the **color** frame of a **turtle** inside the **cast** of **theater-1**. Relative boxpaths allow references to be made relative to a local box; these are explained in section 4.5.4.2.

A message passing language is defined on frames: The **ask** primitive is used to send a message to a box. For instance,

```
(ask #/theater-1/cast/turtle :forward 10)
```

will cause the named turtle to move forward 10 units. Boxes specify their responses to messages by means of *methods*, which are themselves boxes containing a program.

Specialized boxes called *animas* offer process-like capabilities. An anima repeatedly sends a **:step** message to its containing box, concurrently with other animas and user-initiated actions. Animas can be used as the basis for building agents.

4.2 Design

4.2.1 General Goals

LiveWorld was designed to support certain basic qualities of interaction: all parts of the system should be concretely represented and accessible, the system should feel reactive, it should permit exploratory or improvisatory programming, and there should be a clear learning path that leads a user from simple to more complex modes of use.

4.2.2 A World of Lively Objects

Creating animate systems requires the creation and organization of many different kinds of parts, including worlds, actors (graphic objects, also called creatures), agents (behavior generators). Each of these parts in turn can have parts of its own or aspects affecting its appearance and function. The number of objects can be quite overwhelming unless deliberate steps are taken to present the complexity in a form that a novice can comprehend.

The approach used by LiveWorld is in the tradition of Smalltalk and Boxer, in that it tries to make computational objects *real* for the user. Objects should be *concrete*—that is, they should

appear to have a solid existence, metaphorically rooted in the properties of physical objects, and *tangible*—they should be accessible to the senses and actions of the user.

The techniques used to achieve these goals include a simple yet powerful object-oriented programming system with prototype-based inheritance (arguably more concrete in its operation than class-based inheritance); and the integration of the object system with an equally simple-yet-powerful direct manipulation interface (see Figure 4.1). Together object-oriented programming (OOP) and direct manipulation serve to make computational objects appear as concrete and tangible. These properties allow users to make use of their visual and motor abilities to interact with computational objects.

LiveWorld is designed to support agent-based programming, or programming with an animate flavor. It is not itself constructed from agents¹¹, but is intended to support the creation of agents, both computationally and from a design standpoint. Object-oriented programming is the existing programming paradigm best capable of supporting animate thinking (see 3.3.2), so LiveWorld starts from that basis. Objects are animate in the sense of being capable of acting, but only in response to messages from outside themselves. This is a powerful paradigm, fully capable of representing a wide range of computational activity, but its metaphorical implications may be limiting. OOP presents a picture of objects as passive, taking action only in response to a stimulus from the outside. For LiveWorld, we want objects to seem capable of acting as animate creatures.

As set out in Chapter 3, the three salient characteristics of animacy are purposefulness, autonomy, and reactivity. The agent systems described in Chapter 5 are more directly concerned with these issues, but the underlying level is capable of providing some basic support for the latter two. Autonomy is supported by special objects called *animas*, which provide a facility similar to processes, but in a style designed to support agent-based programming. Animas provide a “power source” which can be combined with other objects to produce independent behavior, running independently from user activity and from each other. They form the underlying basis of some of the simpler agent systems detailed in section 5.1. Reactivity is supported by *sensors* that allow objects to detect other objects or conditions in their environment, and by the more general computed-slot facility that allows values to be automatically updated from changing environments.

4.2.3 Spatial Metaphor and Direct Manipulation

To make computational objects real for the novice, who has not yet developed the skills to mentally visualize them, objects can be given some of the properties of physical objects, such as appearance, location, and responsiveness to action. In other words, they should be brought out to the user by means of a direct manipulation interface (Shneiderman 1983). The interface should disappear as a separate entity; instead the user should be able to think of computational objects as identical with their graphic representation (Hutchins, Hollan et al. 1986, p97).

¹¹Section 6.3.4 discusses the prospect of building a system that is entirely based on agents.

LiveWorld presents itself as a collection of recursively nested graphic objects, arrayed in space and manipulable by the user. This idea is largely borrowed from Boxer (diSessa and Abelson 1986), following the design principle referred to as *naive realism*. LiveWorld retains Boxer's basic metaphor of recursively nested boxes, but differs from Boxer in the way it treats its basic object. In Boxer, boxes are understood in reference to a textual metaphor. A Boxer box is like a character in a text string that can have internal structure, which will be other lines of text that also may include boxes. Boxes have no fixed spatial position, but are fixed in relationship to the text line that contains them. There are no classes of boxes (except a few built-in system types) and no inheritance.

In contrast to Boxer, LiveWorld boxes are to be understood as graphic objects that may be moved with the mouse. While Boxer's interface is rooted in word processing (specifically in the Emacs text editor (Stallman 1981) from which Boxer derives its command set), LiveWorld's is rooted in object-oriented drawing programs. Both systems extend their root metaphor by allowing for recursive containment.

LiveWorld's object system supports the idea that a user should always be able to access internal data, but shouldn't have to see detail that is irrelevant to her needs. Complex objects can be treated atomically as necessary, but opened up so that their workings can be viewed and their parameters can be changed.

4.2.4 Prototype-based Object-oriented Programming

Object-oriented programming (OOP) makes programs more concrete by giving them a feeling of locality. Under OOP, each piece of a program is closely associated with a particular object or class of objects. Rather than an abstract set of instructions, a routine is now a *method*, belonging to a particular object and intended to handle a specific class of communication events. It has a location, and in a sense its function is made more concrete by its association with a class of objects.

Prototype-based languages (Lieberman 1986) (Ungar and Smith 1987) are an alternative to the class-based object schemes used in more traditional OOP languages such as Smalltalk, CLOS, or C++. Prototype-based OOP arguably offers additional concreteness by dispensing with classes. In a class-based object system, every object is an instance of a class, and methods are associated with a class rather than with particular concrete objects. While classes are usually represented by objects, these objects are of a different order than normal objects and are sometimes called *meta-objects* to emphasize this distinction. In contrast, a prototype-based system has no distinction between classes and instances or between objects and meta-objects. Instead of defining objects by membership in a class, they are defined as variations on a prototype. Any object may serve as a prototype for other objects.

The advantages of prototype-based programming are simplicity, concreteness, and the possibility of a better cognitive match between program construct and the thinking style of the programmer. It eliminates whole classes of metaobjects, and simplifies the specification of inherited properties. In a sense it is object creation by example.

For example, the properties of elephants might be defined by a prototypical elephant (Clyde) who has color **gray** and mass **heavy**. Properties of Clyde become defaults for the spinoffs, so all elephants will be gray and heavy unless their default is overridden, as in the case of pink elephants. With prototypes, the specification of defaults and exceptions are done in exactly the same way, that is, simply by specifying concrete properties of objects. In a class system, one would generally have to use separate sorts of statements to express both “all members of the class elephant are gray” and “except for the pink ones”.

There are some indications that mental representation of categories is structured around prototypes (Lakoff 1987). Whether this translates into any cognitive advantage for prototype-based computer languages is open to question. What prototypes do accomplish for programming is to smooth out the development process, by easing the transition from experimentation to system-building.

LiveWorld’s prototype-based object system is a modified version of Framer (Haase 1992), a knowledge representation tool that provides a hierarchical structure for objects, prototype-based inheritance, and a persistent object store. It is based on a single structure, the frame, which can take the place of a number of constructs that are usually separate. Framer is a representation-language language in the tradition of RLL-1 (Greiner 1980). In this type of language, slots of a frame are first-class objects (frames) in their own right. This ability to put recursive slots-on-slots makes it relatively easy to implement advanced programming constructs that are difficult or impossible to realize in conventional object systems, such as facets, demons, dependency networks, and histories. For example, any slot-frame can have a demon added to it, which will be stored on a slot of the slot.

Framer was not designed as an object system for dynamic environments like LiveWorld, and required a few modifications, additions, and speedups. I chose Framer for this task because of its hierarchical structure, which was a perfect match for the Boxer-like interface I envisioned for LiveWorld, because of its simplicity, because of its prototype-based inheritance scheme, and because the ability to recursively annotate slots made it a very flexible vehicle for experimentation. LiveWorld’s graphic representation for the frames is the box, which is described below. From the user’s perspective boxes and frames are identical, and for the most part I will use the terms interchangeably.

4.2.5 Improvisational Programming

A programming system for novices ought to support experimentation and improvisation. The model of use should not assume that the user has a fully worked-out model of the task. Rather, the system should assume that the user is engaged in an activity that consists of incremental and interleaved design, construction, debugging, and modification. This means that each part of the user’s cyclical path between idea, execution, and evaluation has to be short and relatively painless.

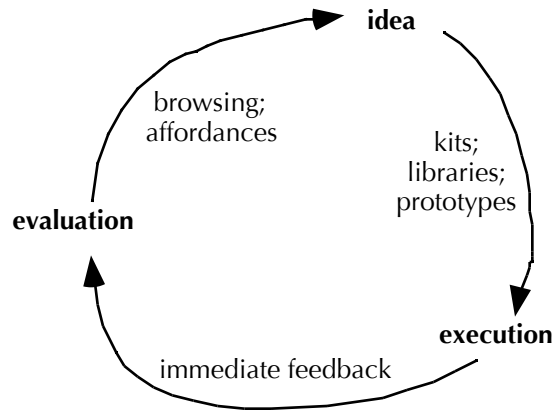


Figure 4.1: How LiveWorld supports improvisational programming.

Improvisational programming relies on the ability to build programs incrementally¹². This facility is an important part of dynamic programming environments for novices such as Boxer or Smalltalk, since novices need the short feedback cycle that incremental programming can provide. From the standpoint of interface design, incremental programming is akin to non-modal interface design: conventional programming requires constant switching between writing and testing, while incremental programming attempts to reduce or eliminate that modal switch.

A graphic browsing and editing interface contributes to the improvisatory feel of a system by laying out all the available parts and options for the user, encouraging combination, modification, and experimentation. Lisp and Logo generally have *not* included such a facility, but Boxer and Smalltalk have made browsing interfaces a prominent part of their design.

The combination of object system, incremental programming, and graphic construction gives the system the feel of a construction kit. A construction kit essentially provides a set of parts that are both more complex than “raw material” and are designed to fit together. Kits support improvisation because they reduce the effort needed to try new configurations.

Prototype-based object systems are arguably better at supporting improvisation than their class-based equivalents. A class-based system makes it hard to create one-of-a-kind objects, so building something is always at least a two stage process (first defining the class, then instantiating an instance). Prototype-based systems eliminate this barrier, freeing users to build concretely without having to first turn their concrete ideas into abstract classes.

The immediate feedback of the system closes the gap between execution and evaluation. Graphic displays that provide affordances for action can involve the user in object manipulation and suggest courses of action.

¹² Sometimes the ability to build programs incrementally is conflated with the distinction between interpreted and compiled languages. While the idea of interactive incremental program construction has its roots in interpreted languages, modern programming environments support incremental compilation, so that the benefits of compilation and incremental construction can be combined.

4.2.6 Parsimony

Also borrowed from Boxer is the idea that a single structure, the box, can be made to serve a multitude of purposes. In most languages, variables, data structures, sets, graphics, classes, buttons, and palettes are very different sorts of things with very different interfaces. In Boxer and LiveWorld all of these structures are replaced by the box, which is flexible enough that it can perform all of their functions. This technique is of great value in reducing the complexity of the interface, since only a single set of manipulation skills needs to be learned.

4.2.7 Metacircularity

As much as possible, the interface of LiveWorld is defined in the language itself, giving the user control over its appearance and operations. Framer's recursive structure makes it easy to use frames to represent information *about* frames, and LiveWorld makes frequent use of this capability. For instance, display information for frame boxes is stored in (normally invisible) frame annotations to each displayed frame (`%box-position`, for instance). If you display these frames, they get their own display annotations, recursively. Because LiveWorld represents its internal processes in the same way as its user-level information, it may be considered as capable of a degree of *computational reflection* (Maes 1987) (Ferber and Carle 1990) in that it is possible for programs within LiveWorld to modify their underlying interpreter and related mechanisms.

4.2.8 Graphic Realism and Liveness

LiveWorld strives for a feeling of real objects interacting in real-time. The principle of naive realism is heightened in LiveWorld by displaying boxes and actors as solid objects that retain their solidity as they are moved around. This simple but important technique serves to heighten the realistic and concrete feel of the medium.¹³

¹³ The ability to do smooth double-buffered dragging was made possible by the Sheet system developed by Alan Ruttenberg.

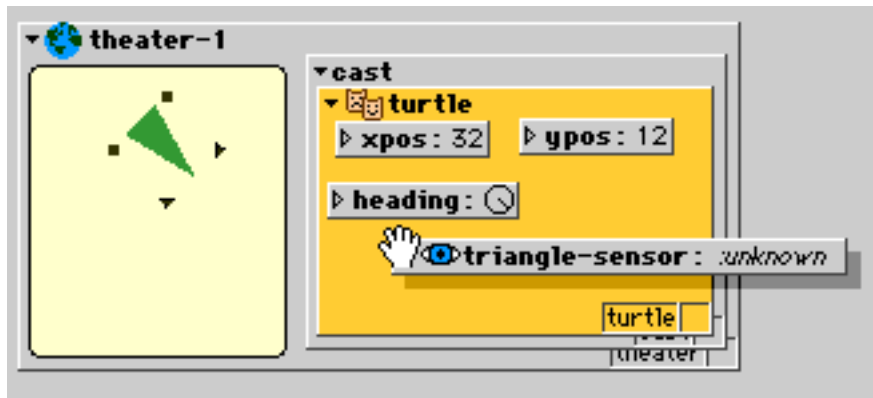


Figure 4.2: A cloned sensor about to be dropped into a turtle actor.

When a box or actor is in transit (for instance, just after being cloned but before being dropped in its home) it appears to float over the world, casting a shadow to indicate its transitional state (see figure 4.2).

The “liveness” of LiveWorld requires that many activities are ongoing simultaneously. This means that screen objects must be able to move concurrently, and also that screen animation can proceed concurrently with user interaction. The feeling of liveness is enhanced by updating displays immediately. For instance, dragging an object will cause any related displays to update continuously, not only after the object is released. Also, autonomous screen objects continue to act during user interaction, and responsive objects may change their behavior as a result of user motion, again even before the operation is complete. The goal of the system is to encourage users to interact with the systems they build.

Many systems that allow interactive editing of a runnable system use global mode switching as a predominant feature of their interface (for example, the NeXT Interface Construction Kit has a big icon of a knife switch for this purpose). These systems impose a barrier between construction and use that should be unnecessary. LiveWorld avoids the need for global mode switching,¹⁴ encouraging users to modify their systems as they run. This has the effect of tightening the debugging loop as well as making the interface less modal.

4.2.9 Learning Path

A programming system for novices should be both easy to learn and powerful. It is too much to expect a powerful system to be learned all at once, so the system needs to provide a learning path consisting of techniques that allow a user to progressively learn and adopt features of the system. Each layer of the system should provide “conceptual scaffolding” that

¹⁴ LiveWorld *does* have a master switch to turn off animation, but its use is optional—editing can take place while things are running.

allows the learning of the next layer to be rooted in existing skills. Complexity needs to be hidden *until it is needed*, at which time it should reveal itself in understandable terms.

In LiveWorld, there are a number of distinct levels of usage that exist within an integrated framework. The intent is that simple modes of use should be easily learnable and also lead the user into the more complex modes. For example, a user whose only experience is in using direct-manipulation graphics programs can approach the use of LiveWorld as she would a drawing program. Unlike a drawing program, however, LiveWorld makes the properties of the object into manipulable objects in their own right, and can thus introduce this novice user to the idea of properties; slots; and values.

The learning path of a novice might follow these steps:

- use of direct manipulation graphics to create pictures;
- opening up the graphic objects to manipulate properties and slots; using objects as prototypes;
- adding behavior to objects by copying agents from a library;
- debugging behavior interactions;
- customizing behaviors by modifying agents.

The coexistence of several modes of use that shade naturally into each other creates a learning curve. Mastery at one level can function as scaffolding for learning the next.

4.2.10 Rich Starting Environment; Libraries

One important stage of use in the scaffolding model is programming by copying and combining pre-built objects and behaviors from libraries. Later, users may open up these objects to see how they work and change them. Alan Kay has termed this style of use “differential programming” (Kay 1981). The idea behind differential programming is to support the common desire to make an object that is almost identical to an existing one, but with a few small modifications.

This is quite different from Logo’s approach (see section 1.3.3), while Boxer is somewhere in between. Logo environments generally begin as blank slates, containing only a turtle. A new Boxer environment presents a blank slate to the user, but the educational community around Boxer has also developed a style in which the user begins with an already-constructed microworld containing both working examples and components that can be used in new simulations.

LiveWorld goes beyond Boxer in that it is designed to permit easy modification and combination of pre-existing components by direct manipulation. Any existing object (from a library or elsewhere) can be cloned, and the clone can then be modified incrementally without affecting the original. If a modification turns out to have an undesirable effect, the modification can be undone by deletion, and the object will revert back to the properties and behavior of the original. LiveWorld’s hierarchical object system allows library objects to be combined into new

configurations: i.e., a user might combine a graphic representation for an actor cloned from a library of pictures and add to it sensors and agents cloned from separate libraries.

Aside from their use as sources of prototypes and components, the libraries also act as sources of examples. That is, each entry in a library may be treated as a closed object, which is used as is, or an open object that can be modified or inspected.

4.3 Box Basics

This section introduces the essential features of LiveWorld's representation and interface.

4.3.1 Boxes Form a Hierarchical Namespace

The LiveWorld environment is made out of boxes or frames. The terms are used interchangeably except in contexts where there is a need to distinguish between the screen representation (box) and the internal structure (frame). From the user's perspective, they are the same. Boxes are named and can contain other boxes as *annotations*. Boxes thus form a hierarchical namespace, with a single distinguished box as the root. The inverse of the annotation relation is called *home*, so that all boxes except the root have exactly one home.

Framer's syntax for referring to frames uses a syntax based on Unix pathnames. `#/` designates the root frame, `#/theater-1` designates an annotation of `#/` named **theater-1**, and `#/theater-1/cast/bug/xpos` designates a frame named **xpos** several levels down in the hierarchy whose home is `#/theater-1/cast/bug`. LiveWorld introduces some variants on this syntax which are explained in section 4.5.4.2. Boxes also can have a *value* or *ground*, which may be any Lisp object, including another box. The value allows a box to serve as a slot.

4.3.2 Inheritance, Prototypes and Cloning

A box can have a *prototype*, from which it inherits values and annotations. For example, the prototype of `#/poodle` is `#/dog`. `#/poodle` will inherit all the properties of `#/dog`, but can override ones in which it differs, such as `#/poodle/hair-curliness`. The inverse of the prototype relation is the *spinoff*, so `#/dog` has `#/poodle` as a spinoff.

By default, a frame's prototype is the annotation of its home's prototype with the same name. For example, if the prototype of `#/dog` is `#/wolf`, then the default prototype of `#/dog/personality` will be `#/wolf/personality`. However, the prototype can be any frame at all, so that while `#/rover` might be a spinoff of `#/dog`, `#/rover/personality` could be changed to be a spinoff of `#/sheep/personality`. Within LiveWorld, new frames are usually created by *cloning* existing frames; see 4.4.4.

Framer provides only single inheritance (a frame can have at most one prototype). But since contained frames can have their own prototypes that violate the default rule, it is possible to simulate some of the effects of multiple inheritance.

4.3.3 The Basic Box Display

LiveWorld's display consists of a single hierarchy of nested boxes, each representing a frame. A typical set of boxes is illustrated in Figure 4.3.

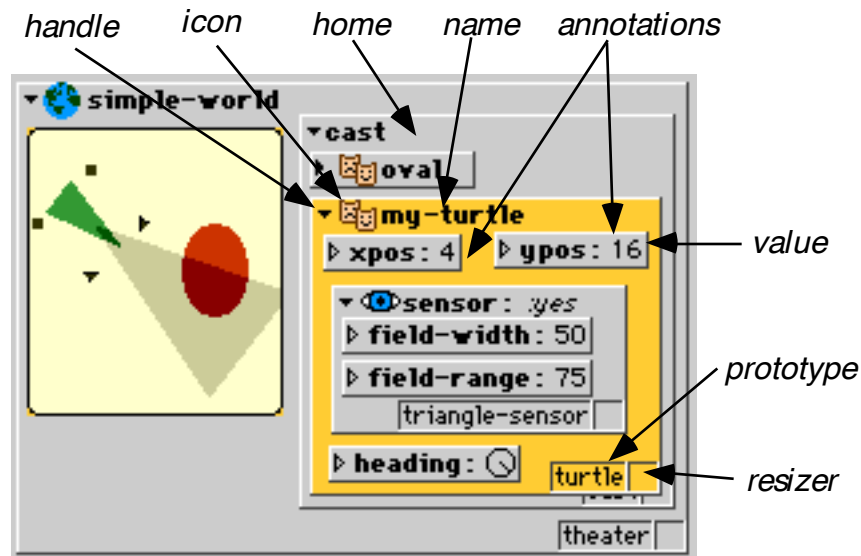


Figure 4.3: Boxes and their parts. The labels denote parts or relations relative to the object **my-turtle**.

Boxes consist of the following graphic elements (from left to right on the top in Figure 4.3):

- *handle* – controls whether the box is open (displaying its annotations) or closed. The handle appears in solid form if there are (interesting) annotations to display, otherwise it appears in outline form.
- *icon* - icons can give a box a more distinctive visual appearance. Icons, inherited like other box properties, serve to indicate broad types of boxes (in the illustration, the icons for worlds, actors, and sensors are visible) and may also be used to indicate the state of the object represented by the box (as in Goal Agents, see 5.1.2).
- *name* – the name of the box. Required; and must be unique within the box's home.
- *value* – the value or ground of the box, if it has one.

The following elements are only visible when the box is open:

- *annotations* – these appear as contained boxes. Only interesting annotations are displayed, see 4.4.6.
- *prototype* – the name of the boxes prototype appears in the lower-right corner of the box.
- *resizer* – used to change the size of the box.

Each of these elements also functions as an affordance for action. For instance, clicking on a box's prototype indicator can select the box's prototype, or change it. Clicking on the name or

value accesses similar functionality. The actions selected by clicks can be customized through click tables; see 4.4.3.

4.3.4 Theatrical Metaphor

Graphic objects are implemented by special frames called *actors* that live in *theaters*. Theaters offer two views of their objects, a *cast* view and a *stage* view. Both the cast and stage are frames themselves, and the actor frames are actually contained within the cast frame and draw themselves in the stage frame. The two views of the objects are interconstrained (so that, for instance, dragging an object will continuously update the relevant slot displays). In Figure 4.3, *simple-world* is a theater with the *stage* and *cast* visible as annotations.

A library of basic actors is provided, which the programmer can clone and customize. These include various shapes, lines, turtle-like rotatable objects, and text-actors. The library itself is a theater and new objects are created by using the standard cloning commands. There is no need for specialized palette objects.

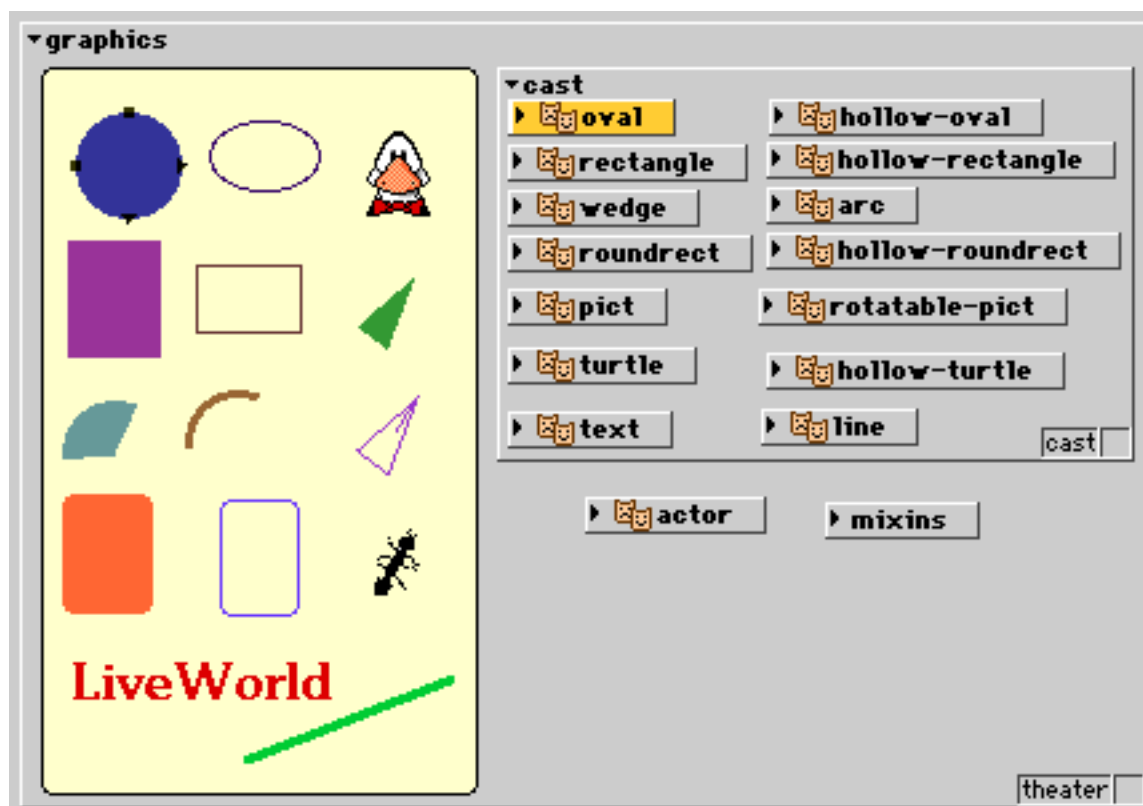


Figure 4.4: The library of graphic objects

Actors respond to roughly the same commands as boxes. Both may be cloned by the same mouse-keyboard gesture, for instance. Both may be dragged, although the meaning of dragging an actor is different from dragging a box. The meaning of gestures may be independently customized for both actors and boxes.

Composite actors can be created by making a separate theater and cloning its stage (see figure 4.5). Instead of another stage, this makes an actor that contains all the actors of the original stage as parts. The “inheritance” of parts is dynamic, which means that changes in the original parts are immediately reflected in the composite.

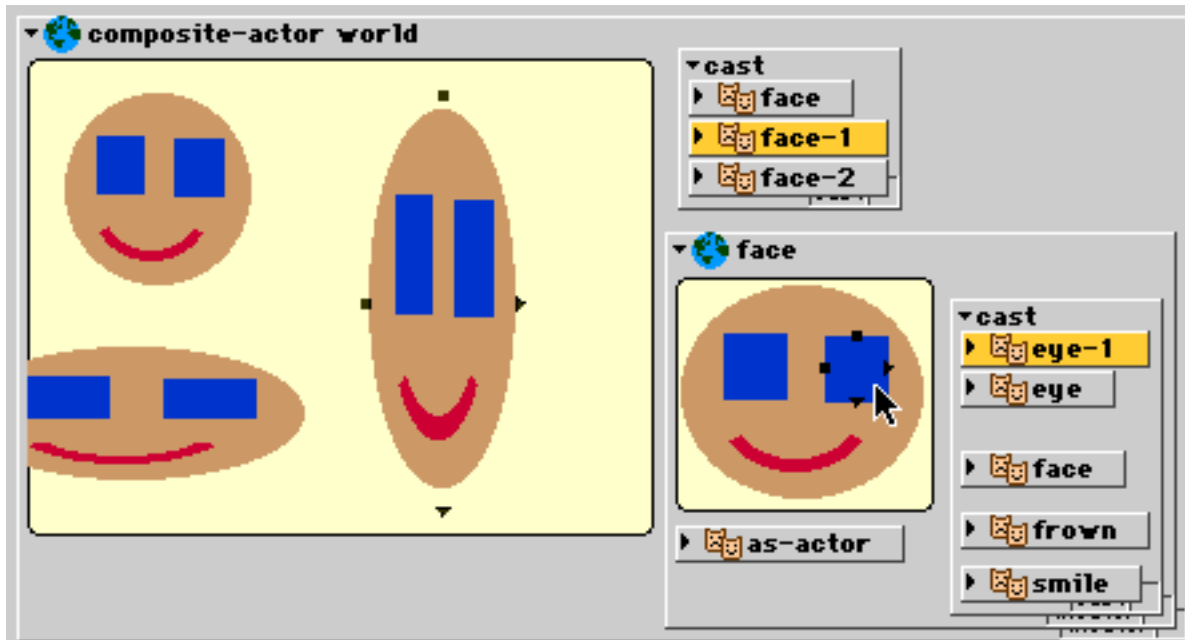


Figure 4.5: Composite actors. The theater on the right allows the parts of the face to be manipulated; while on the left the face is a single actor.

The link between graphic and non-graphic objects may help novices who are experienced users of direct manipulation interfaces make the transition to programming, both by showing them a different view of their actions and by permitting them to interact with computational objects in a familiar way. On the other hand, the link between the graphic and box views is somewhat *ad hoc* and in conflict with the goal of making objects seem concrete by strongly identifying objects and their representation (see section 4.8.1).

4.4 Interface Details

An interface is what gets in between you and what you want to do.

— Carl Haverdine (*Haverdine 1988*)

4.4.1 Selection

LiveWorld’s interface command structure is based on the standard select-then-operate user interface model. Once a box is selected, it may be operated on by menu commands, keyboard commands, or mouse gestures, all of which can be customized for particular boxes and their spinoffs.

Multiple selection is partially supported. Several frames can be selected together and operated on as a group. For instance, if a group of actors is selected they can all be sent a `:grow` message with a single menu selection. However, not all operations make sense for all groups of selected boxes. This is especially true when boxes on different levels are selected. In particular, if a box and one of its containing boxes are both selected, dragging becomes problematic, because the base against which the inner box is moving is itself moving.

4.4.2 Menus

LiveWorld has three menus of its own. The LiveWorld menu contains global controls and items that operate on the selected box. This is about the same set of operations described in the mouse-gesture table below, plus some additional commands for saving and restoring frame structures to files.

The Slots menu shows what slots (annotations) are accessible for the selected box. Both local and inherited slots are included. Slots are organized under the prototype box in which they appear in the regular box view, and the prototypes are arranged in inheritance order, providing self-documentation of the inheritance structure.

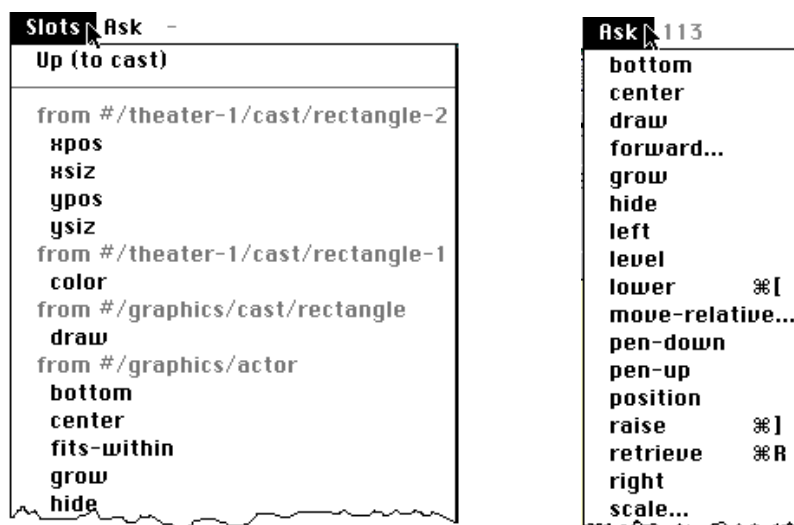


Figure 4.6: Slots and Ask menus, in this case for the box named `#/theater-1/cast/rectangle-2`.

Selecting an item from the Slots menu finds or creates a slot frame as an annotation to the selected frame, declares that slot interesting (see 4.4.6) so that it will be visible, and selects it. Since this has the effect of moving down one level in the box hierarchy, the Slots menu also includes an inverse operation, `Up`, which allows the user to navigate up the box hierarchy by selecting the box enclosing the current selection.

The Ask menu lists all available methods or slots with values for the selected frame. Choosing an item from the menu sends the corresponding message to the selected frame, that is, it is equivalent to a call to the Lisp `ask` procedure (see section 4.5.2). If the message

corresponds to a method that takes arguments, the user will be asked to supply them. The menu items indicate whether or not they take arguments and if they have an equivalent keyboard command (see section 4.5.3. If the ask procedure returns a value it is displayed next to the Ask menu.

Both menus are customized for the selected box or boxes. In the case of multiple selection, the Ask menu shows only methods or slots that are valid for each of the selected boxes. It's more difficult to define a sensible behavior for the Slots menu when more than one box is selected; currently it simply shows the slots for a single selected object.

4.4.3 Mouse Operations

Many operations in LiveWorld can be performed with mouse gestures. A mouse gesture is a combination of a mouse click on some part of a box together with some set (possibly empty) of modifier keys.

The use of modifier keys is somewhat unfortunate, since they are not a particularly easy-to-learn interface technique. Ideally, there would be an affordance (that is, a visible and clickable part) for every possible action. If this were so then there would be no need for modifier keys. But the limitations of screen real estate, together with the large number of operations possible, does not allow this. The combination of separate box regions and modifier keys is a compromise measure. To ameliorate the difficulty this may cause, several rules are adopted:

- All gesture operations can also be accessed from the LiveWorld menu.
- The interface tries to maintain consistency of meaning among the particular modifier keys. For instance, the command modifier usually indicates an editing operation, while the command-option-control modifier indicates a deletion (of a frame, value, or prototype, depending on the region of the click).

	<i>no shift keys</i>	<i>cmd</i>	<i>opt</i>	<i>ctrl</i>	<i>shift-opt</i>	<i>cmd-opt-ctrl</i>
body	drag	edit value	clone	step	move	delete
title	-	edit name	-	-	-	-
value	-	edit value	-	-	copy value (shift-cmd)	delete value
handle	open or close	open or close all	open with internals	-	-	-
prototype	select prototype	edit prototype	-	-	-	delete prototype
resizer	resize	-	default size	-	windowize	-
actor	drag actor	resize actor	-	step	-	-

Table 4.1: Global mouse gestures. An entry of “-” means the behavior of a click on the region is the same as the corresponding click on the body.

Boxes can customize their handling of mouse gestures by changing the value of annotations within a set of frames known as *click tables*. For example, buttons (Figure 4.7) work by providing an entry in the body-click-table for ordinary clicks that calls a user-specified action (and makes control-click take over the ordinary function of drag, so the button can still be moved). A separate click table can be defined for each part of the box (body, resizer, prototype, and so forth). For the purposes of mouse click handling, the graphic representation of an actor is considered a region and has its own table. Click tables also serve as documentation for mouse gestures

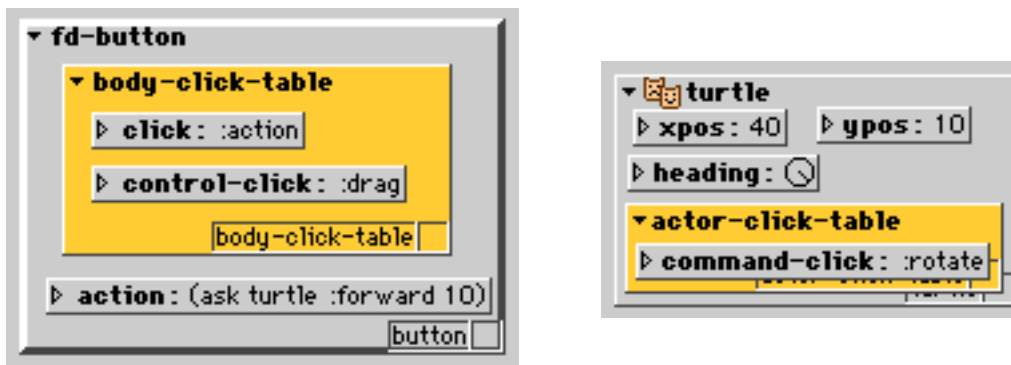


Figure 4.7. Illustrating the use of click-tables. The left-hand box is a button which performs an action when clicked (it can be dragged with control-click). The box on the right is a turtle actor, which will be rotated rather than resized (the default) when given a cmd-click mouse gesture.

4.4.4 Cloning

New objects are usually made by *cloning* existing objects with a mouse gesture. The steps involved are as follows:

- The user clones an existing box with a mouse gesture.
- A new box is made that has the original as its prototype. The new box lives in a special frame called `#/limbo`.
- The new box appears to float over the world, casting a shadow (see figure 4.2). As it is dragged, potential homes are indicated by highlighting them. Some boxes specify that they can only be dropped into particular types of boxes (for instance, sensors can only be dropped into actors).
- The new object is dropped into its home when the user releases the mouse.
- Install methods are run (see section 4.7.4).

4.4.5 Inheritance

LiveWorld objects that are cloned from a prototype maintain a live link back to that prototype, and continue to look up values for local annotations in the prototype until local values are defined. This results in what I call *dynamic inheritance*, meaning that changes in prototypes can affect spinoffs, a fact which is reflected in the interface. This is most noticeable in the case of actors, whose properties can be affected by mouse motions. If an actor with spinoffs is resized, for example, and the spinoffs are inheriting size values from it, they will also change as the mouse moves. This gives a strong tactile feeling for the nature of inheritance. However, it can also lead to expectation violations. An attempt was made to mitigate this by the use of special dragging handles that indicate which objects will be affected by a drag or resize operation (see Figure 4.8).

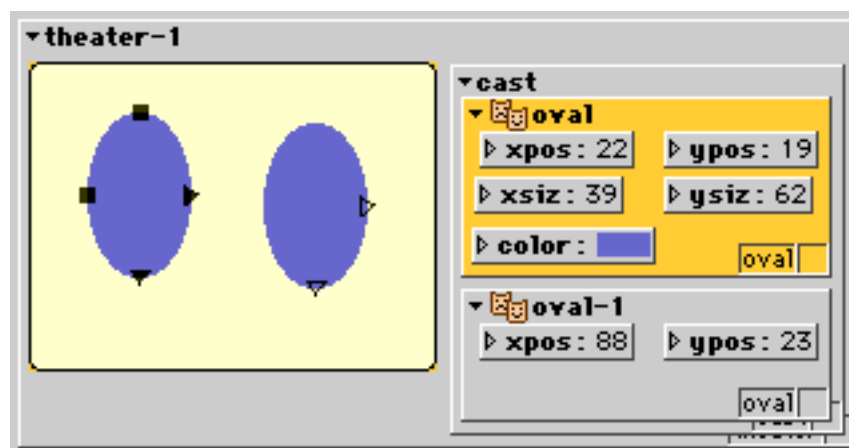


Figure 4.8: Dragging handles. **oval** (on the left) is selected, so it has solid handles that allow the user to change its position or size slots. **oval-1** inherits its **xsiz** and **ysiz** slots from **oval**, so it has hollow handles to indicate that it too will change if those properties are modified in **oval**.

Dynamic inheritance is a useful feature for conveying the idea and feel of inheritance, but it is somewhat in tension with the realist metaphor. Although it is based on realistic cognitive models it does not correspond to the behavior of real world objects. While you might understand your neighbor's car in terms of a "prototypical car" (perhaps your own) you do not expect the neighbor's car to change colors if you should happen to paint your own!

This issue is an example of what (Smith 1987) refers to as "the tension between literalism and magic". There is a tradeoff between adherence to a real-world metaphor and providing access to the useful properties of the computational world that might appear to violate that metaphor. In this case I was particularly interested in exploring the properties of an interface that incorporated dynamic inheritance, so the decision was made to allow this "magical" property into the system. A system aimed solely at novices might choose to adhere more closely to the literal metaphor, or perhaps make dynamic inheritance available as an option.

4.4.6 Interestingness

Framer creates local annotations for slot references. This means, for example, if a rectangle object inherits its color from a parent, it will nonetheless get a local **color** annotation on the first reference. This is done primarily for efficiency—the local annotation serves as a cache that will speedup future lookups. However, it means that there are many slots in existence which are not of direct interest to the user. A newly created rectangle has about 12 such local slots, with only 2 of them (the x and y positions) actually containing information unique to that rectangle.

To deal with this problem, LiveWorld defines an *interestingness* predicate for frames which controls whether or not they are displayed. Essentially a frame is deemed interesting if it is acting as other than a local cache. This is true if it fulfills at least one of the following criteria:

- It has a local value;
- Its prototype is other than the default;
- It contains an interesting annotation; or
- It has been explicitly marked as interesting by the user.

The last clause allows users to make local slots visible, usually for the purpose of modifying their value. For instance, when a user creates a new frame using the Slots menu it is marked as interesting to ensure that it actually appears. Some frames have this property set by the system, such as the **cast** and **stage** annotations of theaters, because such frames should always be visible.

4.4.7 Box Sizes and Positions

The size and position of a box are specified by internal annotations on the corresponding frame. These annotations are *internal*, that is, they are not normally seen by the user (see section 4.7.2). Using the system to represent information about itself has several advantages:

- Sizes and positions are automatically saved and restored along with the rest of a set of boxes when they are closed or written out to a file.
- Box sizes and positions inherit dynamically, just as those of actors do. This means that dragging or resizing a box might change the position or size of a spinoff. This behavior might be disconcerting, but can also serve to reinforce the reality of inheritance relationships to the user.
- A user can choose to make the internal frames visible, which results in a dynamic display of the internal workings of the system.

By default, boxes will automatically resize themselves to be just large enough to contain their contents. This too is done on a dynamic basis, so if an internal box is being dragged, its container will resize itself during the drag. This process is known as “shrink-wrapping”. Of course a user might also want to specify the size of a box by hand, for instance if she was planning to drop a large set of new items into it and wanted to make the space in advance so they could be arranged properly. To deal with this a box has a third internal slot, **preferred-size**, which contains the user-specified preferred size or **nil** if the default shrink-wrapping behavior is desired.

4.5 Language Extensions

This section describes LiveWorld’s extensions to Lisp. The facilities include accessors for boxes and their components, a message passing language defined on boxes.

4.5.1 Accessors for Boxes

Framer extends Lisp with the frame structure and accessors for its components (i.e., **frame-home**, **frame-prototype**). LiveWorld has a slightly modified set of functions for accessing annotations and grounds: names of boxes are given in the form of Lisp keywords, which are preceded with a colon and not evaluated.

(getb *box name*)

Returns the annotation of *box* named *name*, or generates an error if the annotation is not present.

(getb-safe *box name*)

Same as **getb** but returns **nil** rather than getting an error.

(getb-force *box name &optional prototype*)

Same as **getb**, but will create the frame if necessary (as a spinoff of prototype).

4.5.2 Message-Passing with ask

LiveWorld defines a message-passing protocol on boxes. When a box receives a message, which consists of a name and some optional arguments, it looks for an annotation with the same name as the message. This annotation should be a method object, which specifies an action (see the next section). In other words, methods are themselves boxes and are inherited

by the usual mechanism. A single primitive, **ask**, is used both to send messages and to retrieve values from slots. The syntax of **ask** is as follows:

(ask *box message arguments)**

Ask examines the annotation of *box* named by *message*. If it is a method, that method is applied to the box and arguments. If it's an ordinary slot, the value of the slot is retrieved and returned. For example:

(ask turtle :color) Returns the value of turtle's color slot.

(ask turtle :forward 10) Causes the turtle to move forward 10 pixels.

There are also functions **send** and **slot**, which are like **ask** except that they always do a message send or a slot-lookup, respectively. This can be useful if a program wants to access a method object as a value rather than invoke it (something users would not ordinarily need to do).

ask-self is a variant of **ask** that can be used to access a slot or method directly. While **ask** is analogous to message-sending constructs in other languages, **ask-self** is rather unusual. It depends upon the fact that slots and methods are first-class objects and are local to their containing object:

(ask-self *slot-or-method arguments)**

will return the value in *slot-or-method*, or invoke it as a method if it is one. Ask-self is often used to extract a value from a box. Because this operation is quite common, it can be abbreviated as **v** for value. The following constructs are thus equivalent:

(ask *box name*)

(ask-self (getb *box name*))

Setting values is done by sending them a **:set** message:

(ask *slot :set value*)

This allows objects to specify special actions to be performed on sets. For instance, computed slots (see section 4.6.3) get an error if an attempt is made to set their value. Because setting values is such a common operation, it may be abbreviated using the Common Lisp construct **setf** with a value accessing construct such as **ask** or **v**:

(setf (v *slot*) *value*)

(setf (ask *box slot*) *value*)

4.5.3 Methods

Methods are written in Common Lisp augmented with Framer and LiveWorld extensions. There are several different kinds of methods, and it is possible for users to add further kinds.

The system defines the following standard classes of methods:

- *primitive methods*: these methods are built into the system code. Their definitions cannot normally be seen or changed by the user¹⁵. Primitive methods can take arguments.
- *simple methods*: these methods consist of a single box which contains the method definition in the form of a Lisp expression. They cannot take arguments. See Figure 4.9 for an example.
- *complex methods*: complex methods *can* take arguments. A complex method is a box with internal boxes that specify the arguments and body of the method.

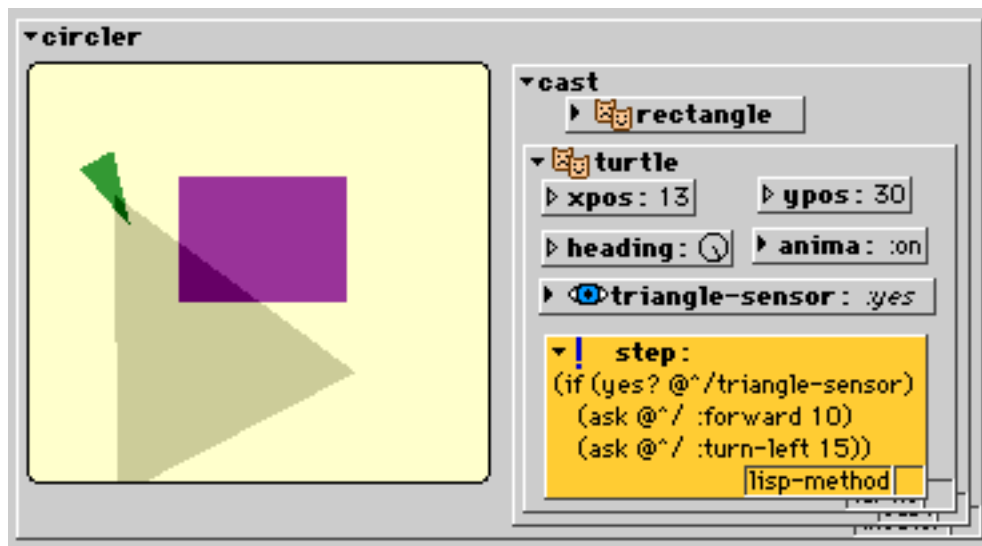


Figure 4.9: A turtle with a simple method for `:step` messages.

Figure 4.9 illustrates a simple method for a turtle that implements a behavior. The method is named `step`, so that `(ask #/circler/cast/turtle :step)` will execute it. The method calls several other methods such as `forward` and `turn-left`, which are inherited from the prototype turtle and are not visible. The `@^/` notation is a boxpath that refers to the turtle (see section 4.5.4.2). `Triangle-sensor` is a sensor (see section 4.6.4). The `anima` causes `step` to be called repeatedly (see section 4.6.1).

The body of a method can access several special symbols that are bound to values that indicate the local context. The symbol `here` is bound to the box in which the text of the expression appears. The symbol `self` is bound to the object which is the method's container. The symbol `call-next` is function-bound to a function that will invoke the method belonging to the object's prototype. This functionality is analogous to `call-next-method` in CLOS (Steele

¹⁵The existence of primitive methods is an artifact of the way in which LiveWorld was bootstrapped, and they could be eliminated.

1990), although the way in which next-methods are found in LiveWorld is very different. If a method calls **call-next**, the system follows the method's prototype link back until an appropriate next method is found. This allows any object to specialize a method while making use of the default behavior, and it also allows behaviors from one object to be imported to another, while preserving appropriate inheritance and calling paths.

Methods can have other slots besides their body and arguments. Among the standard defined slots are **documentation**, whose value can be an explanatory string, and **command-key**, whose value is a character which will make the method invocable by a keyboard command when the parent object is selected (see Figure 4.6).

4.5.4 Relative Box Reference

Methods often need to refer to their containing object or to other boxes. Absolute box names are usually not suitable because they will break if the method is cloned. Methods thus need to refer to boxes in relation to their own location.

4.5.4.1 Self

The simplest way to handle relative reference is through a special variable that refers to the local object. In many other OOP languages this variable is called **self**, and LiveWorld continues that tradition. In the body of a method, the name **self** can be used to refer to the object that contains the method. However, LiveWorld's hierarchical object system generates needs for more complex forms of relative reference. Also, the name **self** is somewhat confusing in the context of a hierarchical object system: it might just as logically be taken to refer to the box in which it occurs, rather than the box for which it is a method, which might be several levels up in the hierarchy.

4.5.4.2 Boxpaths

To solve these problems, I defined a new syntax for **boxpaths**, which provides a more flexible facility for relative reference to boxes and their values. Boxpaths specify a relative path from one box to another. Boxpaths are always specified relative to the box in which they occur, removing the ambiguity of **self**. Boxpaths also allow reference based on types. For instance, a boxpath can specify paths that say "go up to the nearest enclosing **actor** box and then get the value from its **prey-sensor's other** slot."

Boxpath syntax is based on Framer's default syntax. An @ introduces a boxpath, and following characters specify navigation paths through the box hierarchy, either up or down. A caret (^) means "go up", that is, access an ancestor of the current box. A boxpath can go a fixed number of steps up the container hierarchy, or it can search upwards for a particular kind of box. Downward navigation is expressed by the standard frame syntax (see 4.3.1) but with an extension that allows box values, as well as boxes themselves, to be accessed.

This extension is based on Unix directory naming conventions, with a box treated analogously to a directory. A final / in a boxpath indicates that the path is to return the final

box in the path, while the absence of a final / indicates that the final box's value should be returned.

Some examples of how boxpaths express common patterns of box reference:

@/	The local box (the box in which the boxpath appears).
@	The value of the local box.
@^/	The immediate container of the local box.
@^/ food-sensor	The value of the box named food-sensor that is a sibling of the local box.
@^^/	The container of the immediate container.
@^ actor / prey-sensor / other	The example above: "go up to the nearest enclosing actor box and then get the value from its prey-sensor 's other slot."

Boxpaths are essentially a syntactic abbreviation for complex sequences of box accessing functions. For example, the last example above expands into the Lisp code:

```
(ask (getb (lookup-box *here* :actor)
           :prey-sensor)
     :other)
```

Where ***here*** is bound to the box in which the boxpath appears. **Lookup-box** is a function that searches up the box hierarchy for a box of a specified type.

4.5.5 Demons

The standard **:set** method includes a facility for establishing *demons* on a slot. These are procedures that get executed whenever the value of a slot changes. While similar effects can be accomplished by defining a local **:set** method, the advantage of demons are that they can be easily added dynamically, and if there are more than one they will not interfere with each other (that is, a box can have many demons, while it can have at most one local **:set** method). LiveWorld uses demons for many internal functions, such as invalidating computed slots (see 4.6.3) and updating the display of graphic actors if one of their parameters should change.

Demons are stored in an internal annotation (name **%demons**) and are inherited by clones of the parent object. To avoid the potential of infinite recursion, the **%demons** slot itself has a **:set** method for that prototype that avoids the usual demon mechanism.

The demon mechanism can generalize to handle other types of events besides value changes. For instance, boxes can also have *annotation demons* that get called whenever an annotation is added to or removed. A typical use is for a sensor to put an annotation demon

on the cast box of a theater, so that it can be informed whenever a new actor appears in the world.

4.5.6 Global Object

The inheritance scheme of Framer was modified to include the concept of a global default prototype. Named **#/global**, this box serves as a last resort when looking up slots. This object provides the default methods for common operations such as **:set**, the default annotations for interface operations such as click tables, and default templates for universally shared slots, such as **documentation** and **%demons**.

4.6 Specialized Objects

4.6.1 Animas and Agents

Animas are objects that provide an interface to the background execution facility of LiveWorld. An anima causes its enclosing object to be sent messages repeatedly. Animas run concurrently with user actions and provide the underlying ability to create autonomous and reactive objects. Animas are the foundation on which the agent systems of Chapter 5 are built.

When the animas are running, all objects containing active animas are sent **:step** messages repeatedly. Any object can have a **:step** method, but they are most commonly found in method objects. The effect of sending **:step** to a method object is to apply the method to its container. A method of zero arguments can thus be converted into an agent of sorts simply by dropping an anima into it (see Figure 4.9 for an example). Animas are thus conceptually similar to processes, although the execution of animas are not truly concurrent (Chapter 5 goes into more detail about how animas can be modified to provide better concurrency as well as conflict resolution facilities). The execution of animas is interleaved with each other and with user action.

Animas may be turned on or off individually, and there is also a “master switch” in the LiveWorld menu that can disable all animas. When this master switch is off, the entire system can be single stepped by the user, which means that each active anima gets sent a single **:step** message. This is useful for debugging.

It is interesting to note that the metaphorical description of the relationship between processes and procedure is inverted. In more traditional systems, processes are established that execute procedures inside themselves, while in LiveWorld, the process mechanism lives inside the procedural object that it calls. This inversion of the usual relationship is one part of LiveWorld’s attempt to change the metaphors used to describe computation into a form that suggests a greater degree of animacy and autonomy (see Chapter 3).

4.6.2 Specialized Slots

LiveWorld defines some slot objects that are specialized for particular kinds of values, such as colors or points, that have special display requirements. The prototypes for these objects live in a library named `#/slots` (see Figure 4.10). For example, `#/slots/color-slot` defines special methods that cause it to display its value as a color swatch, and to invoke the system color picker when its value is edited. These slots are incorporated into other system objects (i.e., most actors have a color-slot, and turtles and some sensors use an angle-slot to specify their headings).

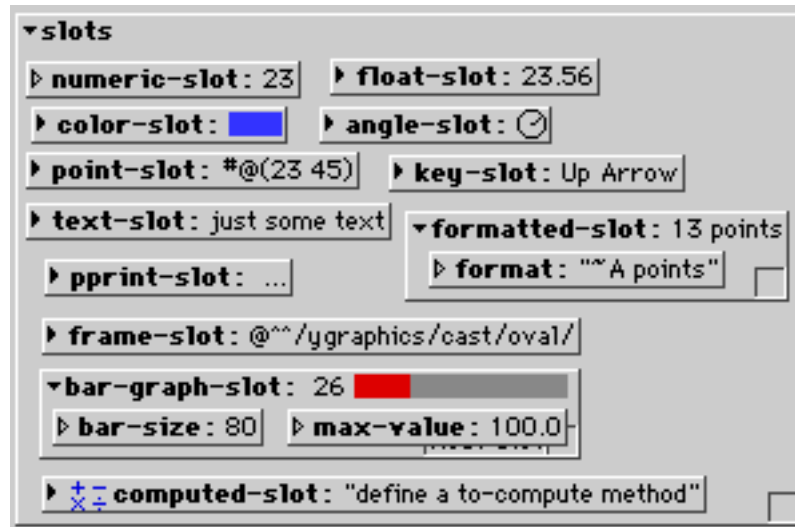


Figure 4.10: The slot library.

Some specialized slots take advantage of Framer's recursive annotations to have parameters of their own. In the figure, `bar-graph-slot` and `formatted-slot` are open to show their parameter slots.

4.6.3 Computed Slots

Computed slots are slots that dynamically compute their value from an expression, in the manner of spreadsheet cells. The value of a computed slot is defined by a **to-compute** method defined as an annotation to the slot. For example, the computed slot `area` in Figure 4.11 computes the area of its containing actor as the product of the actor's regular `xsiz` and `ysiz` slots. Note the use of boxpaths in the method, which allows the area slot to be cloned into other actors.

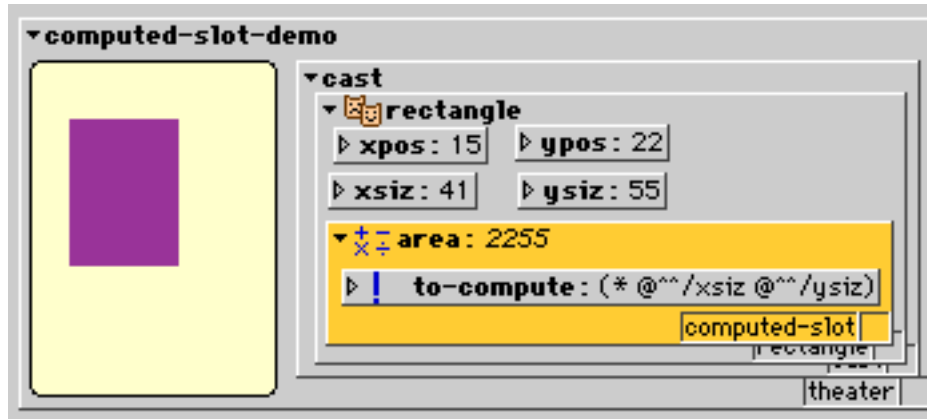


Figure 4.11: An example of a computed slot.

A computed slot's value is accessed using normal mechanisms, but trying to set it will cause an error. The value is displayed in an italic font to indicate that it is not settable.

There are actually several varieties of computed slots. The simplest just computes its value on demand (that is, whenever the rectangle is asked for its area, the multiplication specified in the **to-compute** method will be run). The more complex varieties cache their value and only recompute it when necessary. This requires that the cached value be invalidated whenever any of the values that it depends upon change. Since these values are in other slots, this can be accomplished by putting demons (see 4.5.5) on the depended-upon slots which invalidate the cached values (a technique developed in (Fisher 1970) to support what were called "continuously evaluating expressions"). These demons are installed automatically by monitoring slot accesses during the evaluation of the **to-compute** method.

Figure 4.12 illustrates some of the internal structures and mechanisms that implement computed slots. Every cached computed slot has an internally stored value and a flag that

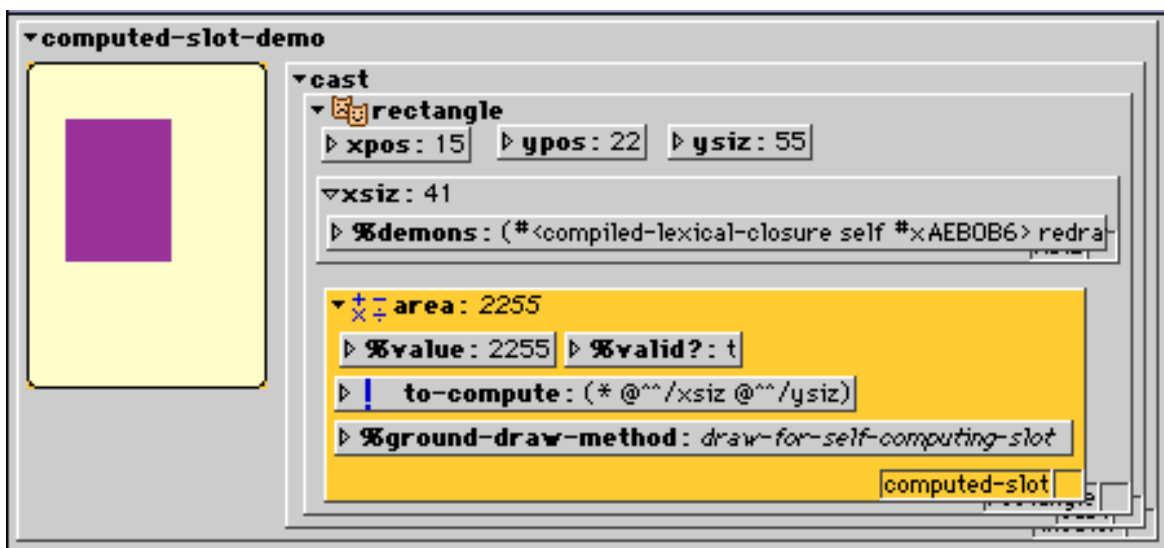


Figure 4.12: The same computed slot, with the internal slots that comprise the computational mechanism revealed.

specifies whether or not that value is valid. Each slot upon which the computed slot depends has a demon added, which causes the computed slot to become invalid whenever the value of the depended-upon slot changes. In this case the demon appears as a compiled Lisp closure.

4.6.4 Sensors

Sensors are objects that allow actors to obtain information about their surroundings, and are implemented specialized versions of computed slots. That is, they are computed slots that compute their value based on the actors' world, and are designed to be annotations of actors. Sensors can also provide a graphic representation of themselves as an add-on layer to the actor they are part of. An example of a sensor may be seen in Figure 4.3. This triangle-sensor is so called because its field is in the shape of a triangular region, and is sensitive to objects that enter that field.

Sensors make particularly effective use of Framer's hierarchical representation structure, which lets them be seen as both slots and objects with slots of their own. Sensors contain slots that specify their parameters or details about the results they are computing (the to-compute method can set other slots besides returning a value). An example is shown in Figure 4.13. **Shape** is a computed slot itself, that depends upon **field-heading**, **field-width**, **field-range**, and

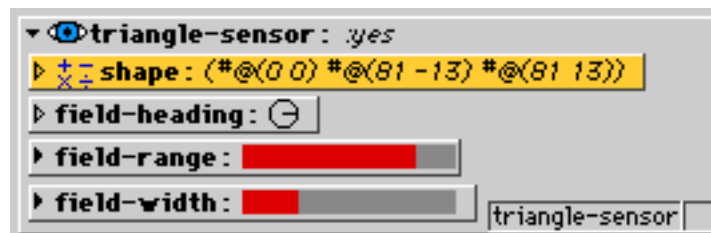


Figure 4.13: Inside a **triangle-sensor**.

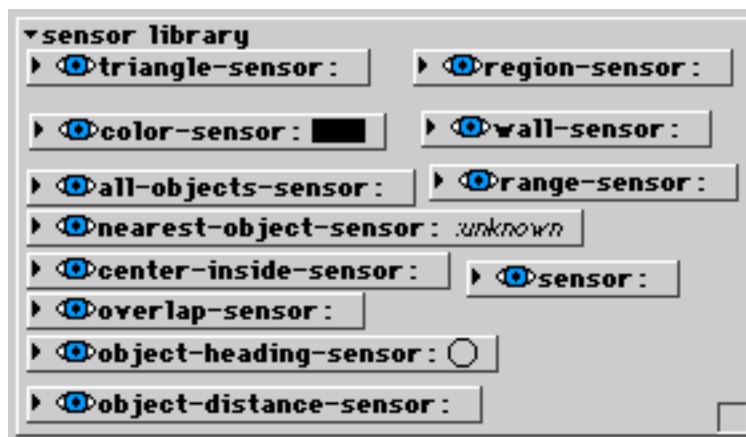


Figure 4.14: The sensor library

on the **heading** of the containing actor. The effect of these dependencies is to let the shape of the sensor be computed automatically when the actor moves or when the user changes the values of one of the sensor's slots. Other parameters allow the sensor to be customized to be sensitive to particular kinds of objects (for instance, only ovals).

There are many kinds of sensors (see Figure 4.14). Some, like the triangle-sensor, have a predicate value. Others have numerical values (for instance, the range-sensor, which monitors the distance of a single object) or return the detected actor or actors themselves (such as nearest-object-sensor). While most sensors respond to actors, some, like wall-sensor, respond to other features of the environment.

4.6.5 Multimedia Objects

LiveWorld's object system makes it straightforward to provide specialized objects for a variety of media, and also to provide libraries of samples. The two most useful and most fully supported media forms are pictures and sounds. Pictures are simply a form of actor. Sounds are specialized boxes with **:play** and **:record** methods. Both pictures and sounds objects are implemented as encapsulations of the corresponding Macintosh system objects, and can be imported from external applications.

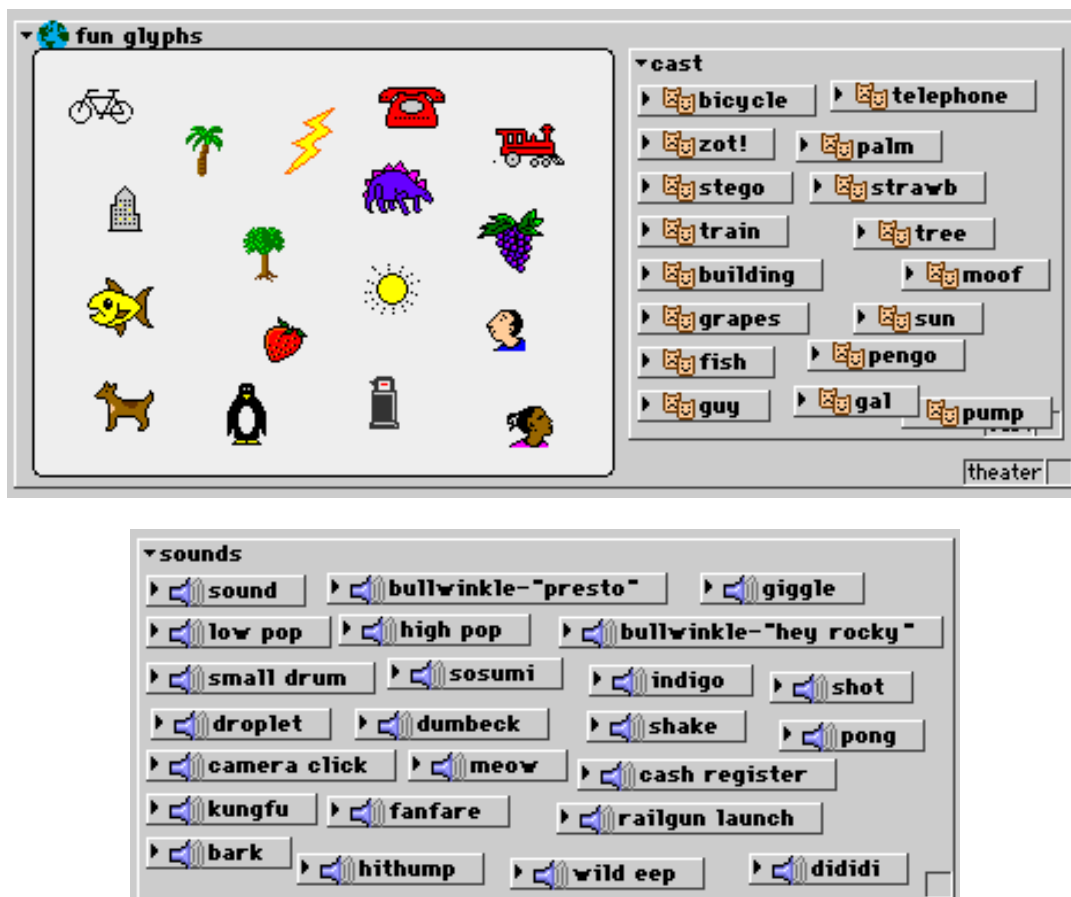


Figure 4.15: Libraries of pictures and sounds.

Sound objects can be used to provide auditory responses for creatures or sound effects for games, but also have a special use for debugging, a technique sometimes called “program auralization” (DiGiano, Baecker et al. 1993) by analogy to the well-established techniques of program visualization.

LiveWorld supports audioization by providing a special kind of object, called an **auto-sound**, which can be dropped into a method. The effect of this is to cause the sound to be played whenever the method is called. This technique not only supports program audioization, but makes it easy to add sound effects to games and simulations.

4.6.6 K-lines

Frames can also be used to implement a form of memory loosely based on Minsky’s K-line learning mechanism (Minsky 1980). In *Society of Mind*, a K-line records a partial mental state represented as the activation state of a set of agents. In LiveWorld, a K-line can record the state (that is, the values) of a selected set of boxes. A K-line is a box created in the usual way. To specify that a slot is to be recorded by the K-line, the user makes a clone of the slot and drops it into the K-line. The K-line thus contains spinoffs of all the frames that it covers and responds to two basic messages, **:store** and **:recall**, which have simple implementations. Store forces each spinoff to have the *current* value of its prototype, while recall copies the value of the spinoff *back* to the prototype. An additional message, **:add-object**, will cause the K-line to cover all of the interesting slots of an object.

LiveWorld’s K-lines were constructed mostly to see how far the prototype relationship could be extended into other uses (for a similar experiment, see the discussion of ports in 4.9.1). While these K-lines are not powerful enough to be of much use in mental modeling, they have proved to have some utility in recording initial states for applications like animations and games.

4.7 Miscellaneous Issues

4.7.1 Shallow vs. Deep Cloning

A question arises when a box is cloned about which of its internal parts should be cloned along with it. For example, in the case of an actor with no complex internal parts, there is no need to clone any of its slots—the values will be inherited. But if the actor has sensors, the sensors need to be copied so that the new creature has its own perceptions, rather than inheriting those of its parent! So, in some cases internal parts *do* need to be cloned.

Which objects require this treatment? Ordinary slots, in general, do *not* need to be copied by this mechanism, since their values are inherited. The objects that need to be copied explicitly are those that cause some side-effect, such as creating an image on the screen (actors) or compute a value based on local information (sensors).

LiveWorld provides a mechanism that allows any box to specify that it must be cloned whenever a box that contains it is cloned. Such boxes are flagged with an internal annotation, **%copy-as-part**. Whenever any box is cloned, all of the original box's annotations are examined via recursive descent for the presence of this flag, which if present causes the box containing it to be cloned into the appropriate place. Since the appropriate place might be several levels deep in the new box, the intervening container boxes must also be cloned.

The same mechanism is used to maintain dynamic inheritance relationships after an object is cloned. For example, if **turtle-1** is cloned to form **turtle-2**, and then **turtle-1** is given a sensor, **turtle-2** should receive a clone of that sensor. The **%copy-as-part** flag also controls this propagation.

4.7.2 Internal Frames

LiveWorld uses annotations to record information about its internal functioning (for example, the screen position of a frame's display is stored as an annotation on that frame). Showing these to the user would be confusing. LiveWorld therefore allows certain frames to be marked as *internal*, with the result that they are not normally displayed to the user in the box view or in menus. The convention used is that internal frames are those whose names begin with the % character¹⁶. An advanced user can choose to display these frames, by holding down an additional key during a command that opens a box or displays a menu. A frame with its internal frames exposed may be seen in Figure 4.16. Internal frames serve some of the same purposes as *closets* do in Boxer.

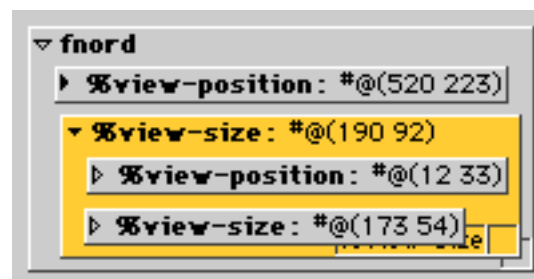


Figure 4.16: A box that has (some) of its internal frames showing. One of them, **%view-size**, also has its own internal frames visible. The values in these slots are Lisp point objects that denote x-y coordinate pairs.

Aside from the display information shown in the figure, internal frames are used to store data for internal mechanisms like demons, internal state like interestingness (see 4.4.6), and slots whose value would be meaningless to the user, such as pointers to Macintosh sound objects.

¹⁶ It would be more elegant to use an annotation to convey this information. The % convention is an artifact of LiveWorld's development history, and has been retained for reasons of efficiency.

4.7.3 Deleting Boxes

Deleting objects can be problematic in a prototype-based object system. One issue is the question of how to deal with objects that inherit from the object being deleted. Framer's built-in handler for deletion simply disallows deletion of a frame with spinoffs. LiveWorld extends Framer by allowing the user to specify a way to proceed when a problematic deletion is detected.

The user's options are to:

- abort the deletion;
- delete all the spinoffs as well; or
- try to preserve the spinoffs by *splicing out* the deleted frame.

Splicing out a frame involves removing it and its annotations from the inheritance chain so that it may be deleted. Each spinoff of the spliced-out frame (and its annotations, recursively) must have its prototype changed to point to the prototype of the spliced-out frame, and if the spliced-out frame provides a value that value must be copied into each spinoff. This complex-sounding procedure retains the old values of slots and retains as much of the inheritance relationship as is possible.

Giving the user choices is fine for interactive use, but does not deal with the problem of deletion by a program in a live simulation. For instance, in a dynamic biological simulation, animals will reproduce by creating spinoffs of themselves. If a parent is eaten by a predator, we don't want the child to be affected, but we also don't want the system to stop and ask the user what to do.

There are also some fundamental problems with splicing out prototypes. Consider the case where there is a prototype ant, and many spinoffs of it (and possibly spinoffs of the spinoffs). Say further that there is an anteater in the simulation with an ant sensor. Sensors often specify the type of object they are sensitive to. If the prototypical ant gets spliced out and deleted, the ant sensor will be pointing to a deleted frame. Even worse, the ants will have no common prototype that allows them to be considered to be a class of objects¹⁷. Furthermore, the ants now each have their own copy of whatever properties were defined by the prototype (such as the complement of sensors and agents) so it is no longer possible to change these characteristics without changing them individually for each ant.

A better solution to the deletion problem, which makes good use of LiveWorld's hierarchical structure, is to keep prototype objects in a separate theater (see Figure 4.17). These prototypes are then outside of the simulation, inactive, and so in no danger of being deleted.

¹⁷It might be argued that using prototypes to specify classes is a bit inelegant anyway. If the sensors were truer to their real-world models, they would not specify a type of object to be sensitive to, but instead would use a detectable characteristic of the object (i.e. the ant sensor would look for objects that were small and black). This is certainly implementable within the framework of LiveWorld, but would be considerably less efficient than the current scheme.

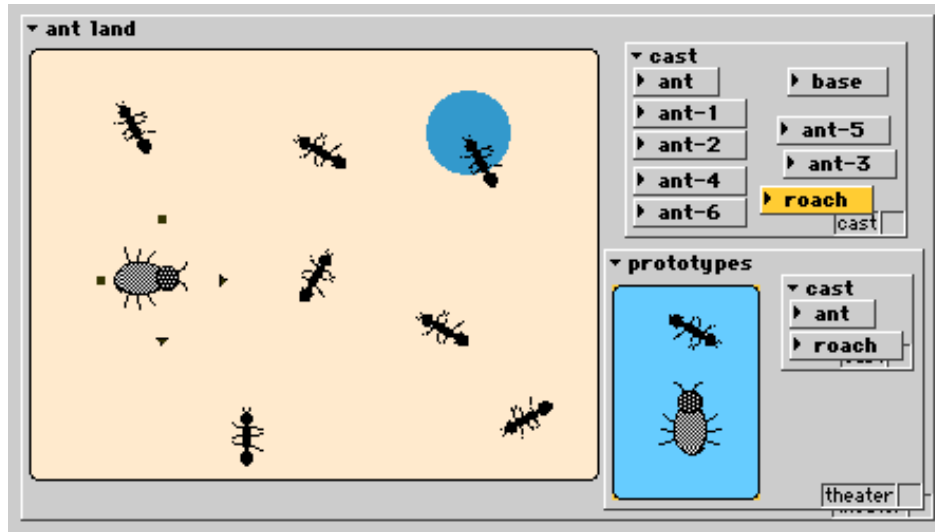


Figure 4.17: Illustrating the use of a separate theater to hold prototypes.

The deletion problem is one that is not found in class-based object systems, where there is no danger that the definition of a class of objects will be taken as an element of that class. Some other prototype-based object systems, such as Self (see section 4.9.2) avoid this problem by using static rather than dynamic inheritance. In Self, spinoffs inherit properties from prototype at the time they are created and do not maintain a pointer back to their prototype, so deleting the prototype is not problematic.

4.7.4 Install and Deinstall Protocol

LiveWorld defines an install/deinstall protocol on frames. Whenever a frame is created, deleted, or moved it is sent `:%install` and `:%deinstall` messages as appropriate (a move will generate both). Methods for these messages handle operations such as installing demons for computed slots or creating the graphic representation for actors.

4.7.5 Error Handling

Any programming environment must be able to deal with unexpected conditions. LiveWorld utilizes Common Lisp's condition handling system to provide an error handling facility with a box-based interface. When an error is detected, it generates a Lisp condition object which is then encapsulated into a box. For any type of error, the system can provide specialized error boxes that provide user options or other information.

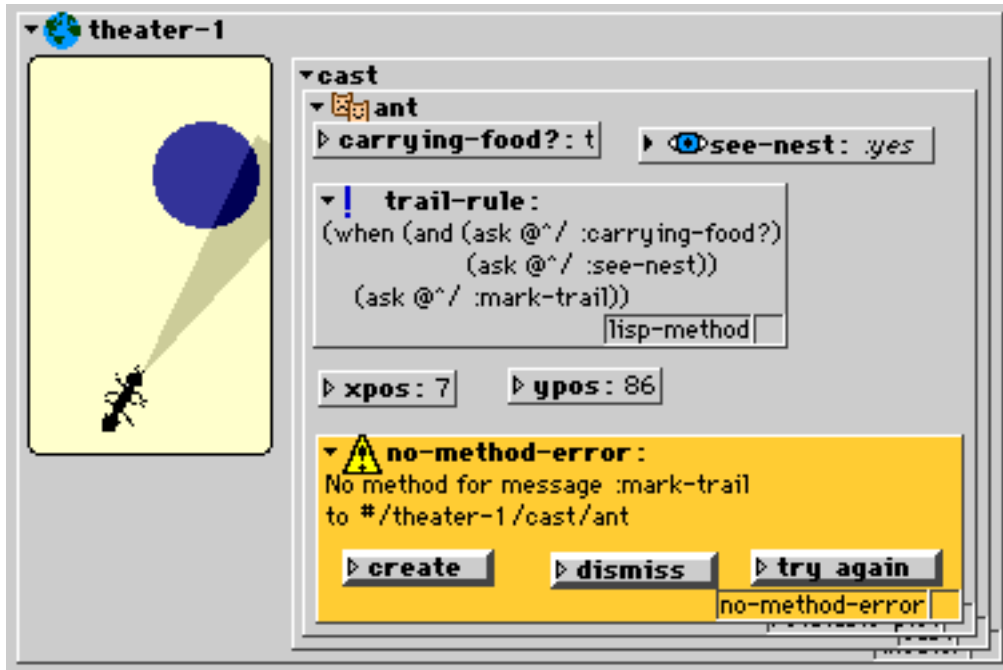


Figure 4.18. An error has occurred in the execution of the **trail-rule** method, generating the error object **no-method-error**.

An example is shown in the figure. The error box contains three buttons that specify restart options. Clicking the **create** button, for instance, causes a method object of the appropriate name and type to be created, after which the user can fill in its slots.

4.8 Some Unresolved Issues

LiveWorld in its current state has a few shortcomings and inconsistencies in its design, which are discussed here. Solutions to some of these problems are also discussed in Chapter 6.

4.8.1 Multiple Views of Actors

The workings of the theater/stage/cast mechanism is somewhat *ad hoc* relative to the rest of the user interface metaphor. These boxes have special relationships between each other that are not found elsewhere in LiveWorld and are not capable of being explained by the basic mechanisms of the system. This magical mechanism is motivated by the need to have multiple views of actors. They must be capable of being seen both as graphics and as structured objects, and so the interface goal of having a perceived identity between object and screen representation must be relaxed.

This is not a severe problem for the interface; the relationship between the views is not hard to understand. Nevertheless it is annoying that a special mechanism is required in this case. Furthermore, the need for multiple views may not be confined to graphic objects, and this

suggests that the solution to the *ad hoc*-ness of the theater mechanism should be a *general* mechanism for supporting multiple views.

A related issue has to do with the need for actors to be able to change their appearance. LiveWorld allows graphic properties of actors to be changed, but not the general form, which is determined by the actor's prototype. Some other actor-based systems (i.e. Rehearsal World (Finzer and Gould 1984)) separate out an object's appearance into a separate object called a *costume*, which can be attached to an actor and easily changed.

One solution to these problems is to introduce the idea that an object is distinct from its graphic appearance and that it can have multiple appearances. Both costumes and the dual views of actors and boxes can be handled by a more general mechanism. This view mechanism might even be able to subsume other aspects of the system such as composite actors and specialized slots. The problem with this solution is that it significantly increases the complexity of the system by introducing new objects, and it might interfere with the feeling that users are manipulating real objects rather than representations of them.

4.8.2 Uniform Interfaces Have a Downside

LiveWorld's box was designed to fulfill many functions with a single uniform construct. This makes learning the system easier, but may impose certain cognitive penalties as well. The elements of the LiveWorld system appear rather uniform, both in terms of graphic appearance and at the functional level. For instance, objects and slots look the same—they are both boxes—and so it is possible, during the course of an interaction, to confuse one for another. The ability for boxes to have icons that indicate their type or states was a late addition to LiveWorld and helps ameliorate the uniformity problem. However, there is still a tension in the design between the desire for uniformity and particularity.

The deeper issue is one of abstract vs. concrete constructs. The box, which is designed to be capable of being a great many different things, is necessarily abstract in comparison with, say, a button that serves a single purpose and has only one operation it supports. While constructing a broad class of functions out of a small set of structural elements is elegant, esthetically pleasing, and reduces the number of things to learn, it can also cause difficulties for novices (diSessa 1986). The difficulty arises because the minimal elegant elements are necessarily abstracted from the concrete functional goals of the user. In contrast, systems in which each component is specialized by an expert for a particular use are less flexible, but they can be carefully designed so that each component is distinguishable, as are, for instance, the various elements of the standard Macintosh user interface (Apple Computer 1987).

4.8.3 Cloning and Dependent Objects

In the early versions of LiveWorld, sensors had to be contained directly within an actor. This caused a problem when trying to define libraries of drop-in behaviors. The behaviors were dependent upon the presence of the appropriate sensors, so cloning the behavior alone from the library would not, by itself, produce the needed behavior. Somehow the behavior needed

to bring the sensor along with it. A number of solutions were considered (and some were implemented). This design issue, while fairly minor, is typical of many such issues that arise in an environment like LiveWorld.

- Give the behavior an **:%install** method that created the needed sensors. The effect for the user would be that cloning an agent would result not only in the agent appearing in the destination, but a sensor appearing alongside it. This scheme has the problem that two behaviors might try to create two different sensors with the same name. Another, more serious problem is that the install mechanism is intended to be below the user's level of awareness, meaning that users cannot (easily) write new behaviors with included sensors.

- Equip the model behaviors in the library with actual sensor objects, that would somehow, when cloned, migrate upwards to the proper place with the actor. This solves the problem of having to write **:%install** methods but not the name conflict problem. It also introduces a rather magical new form of object movement to the interface.

- The third (and best) approach is to modify the implementation of sensors to allow them to live anywhere within an actor rather than requiring them to be directly beneath. This allows prototype behaviors to carry their sensors along with them in the natural way, eliminates the need for migration, and also eliminates the potential for name conflicts. The only disadvantage is that sensors will now no longer be so directly associated with their actors (i.e., it is no longer possible to gain access to all of an actor's sensors by opening up only the actor).

4.9 Relations to Other Work

4.9.1 Boxer and Logo

Boxer was a central influence in the design of LiveWorld. As explained in section 4.2.3, LiveWorld retains Boxer's emphasis on the concrete, its use of hierarchical nesting to hide detail and provide a uniform medium, and its use of a single structure to perform a multiplicity of computational and interface tasks. LiveWorld extends Boxer's ideas by making Boxes into objects with inheritance relationships and a message-passing language, and by providing a direct manipulation interface.

Boxer's interface is based around a generalization of text editing. While the benefits of LiveWorld's object-orientation and direct manipulation are many, there is at least one advantage to Boxer's approach: the text-based model provides for the ordering of the parts within a box, making it easier to express constructs like lists and arrays.

Boxer includes a port mechanism that allows references across the box hierarchy. A port is rather like an alias for a box that can be placed in a remote part of the system, and in most respects functions like the original box. The port displays the value in the original box, and setting the contents of the port changes that of the original box. Ports are an alternative to referring to remote boxes by names or by complicated access paths. For the most part a facility like this has not been needed in LiveWorld. However, ports can be mimicked in LiveWorld

using the prototype relationship. A port is just a spinoff with a specialized `:set` method that changes the value of the original frame rather than the local value. It's somewhat more difficult, though not impossible, to handle the other operations on frames in a port-like fashion, such as adding annotations.

Logo, a direct ancestor of Boxer, was also influential in many ways, but primarily at a conceptual rather than design level (see sections 1.3.3 and 3.3.3). The turtle, of course, derives from Logo.

4.9.2 Self

Self (Ungar and Smith 1987) is the prototypical prototype-based object-oriented language. Its chief feature is an extremely simple object and inheritance model, which makes it easy to teach and implement. From the standpoint of object-oriented programming, LiveWorld's (really mostly Framer's) major contributions over and above what Self provides are in providing slots that are first-class objects, and dynamic inheritance. Another difference is that in Self, an object generally does not inherit values from its prototype, but from a separate object (called the parent). While this has certain advantages, Framer's method is in many respects simpler to understand, and it is easier to represent the inheritance relationships in a dynamic graphic environment like LiveWorld.

Self has an exploratory graphic interface (Chang and Ungar 1993) which has much in common with LiveWorld's interface. Both systems emphasize animation, but use it in very different ways. The Self interface animates its object representations (akin to LiveWorld's boxes) in a fashion designed to make them seem real to the user (i.e., when a new object is made, it does not simply appear, but swoops in from the side of the screen), but this animation is outside the realm of the user's control. LiveWorld is more concerned with allowing the user to control the animation of graphic objects, leaving the more abstract, structural representation of objects unanimated.

4.9.3 Alternate Reality Kit

ARK, or the Alternate Reality Kit (Smith 1987), is another environment with goals similar to that of LiveWorld. In this case the commonality is in providing a lively, reactive feel, and in integrating graphic objects with computational objects. Other shared properties include prototypes (ARK's prototypes work more like those of Self), step methods to drive concurrent objects, and a special limbo state (called *MetaReality* in ARK) for objects in a state of transition.

Both ARK and Rehearsal World, as well as many other visual environments, rely on a separation of user levels. Components are built by sophisticated users, generally outside of the visual environment, while more naive users are allowed to connect these components together graphically but not to open them up or modify them. LiveWorld, on the other hand, tries hard to make every object openable and accessible.

4.9.4 Rehearsal World

The Rehearsal World system (Finzer and Gould 1984) explored the use of theatrical metaphors as an interface to an object-oriented programming environment. Rehearsal world presents objects as “players” which come in “troupes” and can be moved onto “stages”, objects are controlled by “cues”, and putting objects through their paces is, of course, done via rehearsal, which in practice is a weak form of programming-by-demonstration (Cypher 1993) in which objects can record user actions and play them back on command.

LiveWorld borrows from Rehearsal World the theatrical metaphor, and the idea of providing a large set of existing players (a troupe) which users can use as components and starting points for further development. The purpose and feel of the two systems also have many points of commonality: both aim to provide a friendly and dynamic world where programming and direct manipulation are integrated. The main differences are LiveWorld’s unusual object system and its emphasis on concurrent reactive objects. Rehearsal World’s programs, in contrast, are generally “button-driven”, that is, driven directly by user actions.

LiveWorld also incorporated a form of “rehearsal”-based programming, where objects could record a sequence of message sends for later playback. This facility was not developed very far, however.

4.9.5 Ágora

Ágora (Marchini and Melgarejo 1994) is a platform for modeling distributed simulations based on graphic objects, concurrent iterated programs, and communication between them. One of Ágora’s more innovative ideas is the notion of a *communication medium*, which provides a metaphor for broadcast style of inter-actor communication. In this model, any actor can connect itself to one or several communication media. Objects communicate with each other by sending messages to a medium which then rebroadcasts it to the other objects present.

A medium can correspond to various sensory modalities such as vision or smell. The various media can have different behaviors and properties that determine the duration and disposition of messages. Mediums thus take the place of LiveWorld’s sensors. In LiveWorld, sensors compute values obtained directly from the world. The sensors themselves are trusted to look only at slots that make sense given the sensor’s designed-in locality. Having media objects encapsulates this constraint into a separate object. By reifying the idea of a medium, Ágora presents an alternative to the traditional one-to-one image of message passing and provides a conceptual basis for thinking about object interaction in a variety of different circumstances.

4.9.6 IntelligentPad

IntelligentPad (Tanaka 1993) is a “media architecture” visually quite similar to LiveWorld. The basic unit is the pad, graphic objects which can be given various appearances and

behaviors and pasted within other pads to form hierarchical structures. Pads when pasted establish message-passing links between them. Pads serve as displays, buttons, containers, and can represent complex simulation objects like springs and gears.

While the pad environment allows end users to manipulate and combine given sets of pads, it does not support full programming capabilities. Users cannot modify the behavior of the basic pads nor create new kinds of objects. In this respect its design philosophy resembles that of Agentsheets (see section 6.2.2).

4.10 Conclusion

I wish that I hadn't needed to spend the time I did building LiveWorld. In fact, I was both annoyed and somewhat puzzled that there was no system like this available already—an object-oriented system coupled with a powerful underlying language, with an interface that could support multiple moving objects and concrete viewing. This seemed like something that should, in this day and age, be already commercially available and in the hands of children. The absence of such systems is, I think, a reflection of the fact that we don't have languages that are oriented towards programming in such environments. And of course the absence of suitable languages is in part due to the absence of environments! I chose to break this no-chicken-and-no-egg cycle by building the environment first, partly to avoid falling into the common language-design trap of elegant languages that can't do anything tangible. Because of this, LiveWorld is oriented towards agents but not itself based on agents. Nonetheless, it has unique features which derive from the agent-based point of view. While the language and environment may be formally separable, the design values that inform them are the same.

Chapter 5

Programming with Agents

Putting the control inside was ratifying what de facto had already happened—that you had dispensed with God. But you had taken on a greater, and more harmful, illusion. The illusion of control. That A could do B. But that was false. Completely. No one can do. Things only happen, A and B are unreal, are names for parts that ought to be inseparable...

— Thomas Pynchon, *Gravity's Rainbow* p34

Earlier chapters showed that the discourse of computation is founded on a diverse set of metaphors, with anthropomorphic or animate metaphors playing a particularly central role. This chapter presents a series of agent-based systems and languages that are explicitly designed to exploit animate metaphors. Several implementations of agents within LiveWorld are presented here, ranging from extremely simple agents that have only implicit goals, to somewhat more complex agent systems that allow agents to have explicit goals and thus monitor their own activity, and finally to Dynamic Agents, a system which allows agents to dynamically create new goals and agents. This system is flexible enough to build agent systems that operate with a variety of control structures and can perform a variety of tasks in the problem domains set out in Chapter 1.

5.1 Simple Agent Architectures

In this section we develop and examine a couple of “toy” agent architectures that illustrate some of the issues involved in using agents in a dynamic environment. In the first system, Simple Agents, goals are implicit and agents are simply repeating procedures. Trivial though it is, the Simple Agents system still illustrates some of the issues involved in agent concurrency and conflict handling. In the second system, Goal Agents, agents are more structured and have explicit goals that can be used for a variety of purposes. Both systems offer an essentially static agent model, that is, agents are not created or deleted during the course of activity. Systems that allow for the dynamic creation of agents will be the subject of the next section.

5.1.1 Simple Agents

The simplest form of agent that we will consider is essentially just a procedure that can arrange for itself to be invoked repeatedly, or in other words, a procedure that can be readily converted into a process. These agents have no explicit goals or declarative aspect, nor do they have any convenient way to communicate with other agents. Their agenthood rests solely with their ability to carry out their function independent of outside control.

LiveWorld provides a way to turn any method of zero arguments into an agent by dropping in an anima (see section 4.6.1). Thus, the Simple Agents system is really just a straightforward extension of the existing LiveWorld method structure. Animas are hooks for

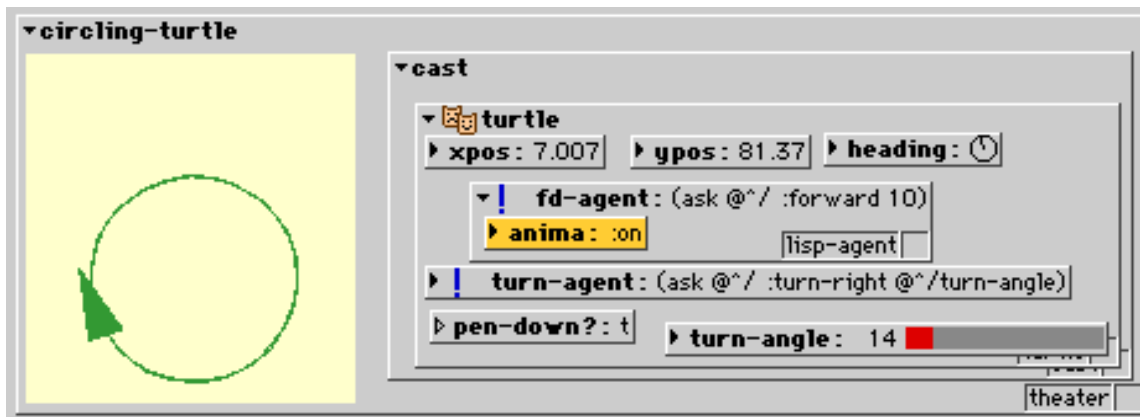


Figure 5.1: a turtle executing a classic turtle circle using two simple agents, one to go forward, and one to turn by a variable amount.

LiveWorld's activity driver, so to understand the action of simple agents in more detail, we will need to look at the anima driver mechanism, and modify it to handle some of the issues raised by agents.

The default anima driver is a simple loop which calls the function run-animas repeatedly. Each call is considered an anima cycle. This driver is shown in Listing 5.1. Note that this driver ensures that anima-driven agents run in lock-step with each other, in the sense that each gets to execute exactly once during each clock cycle. This form of concurrency makes agents synchronized by default, allowing them to be simpler than they would have to be if they needed to achieve synchronization through explicit constructs.

```
(defun run-animas ()
  (dolist (a *running-animas*)
    (send a :step)))
```

Listing 5.1: The very simplest anima driver.

5.1.1.1 Simulating Concurrency

When several animas are active, issues of concurrency and conflict can arise. The default anima driver shown in Listing 5.1 basically ignores these issues. Each agent simply executes in turn, creating problems of unpredictability and order-dependency. The unpredictability comes from the fact that the ordering of agents in this round-robin execution is arbitrary, so that two seemingly identical agent systems might exhibit different behavior if the animas happen to be listed internally in different orders. Additionally, the execution of one agent may affect the subsequent action of another. Ideally, all agents would execute concurrently, so that the results of one could not affect the other and the results of execution would always be predictable.

A form of simulated concurrent execution can be accomplished rather easily in LiveWorld by modifying the anima driver. The basic idea is this: all effects of agents are ultimately expressed as side-effects to the values of boxes. So if all side effects are delayed until the end of an agent cycle, the effects of concurrent execution can be simulated. To do this, the agent

driver needs to be changed. Rather than simply stepping each anima in turn, it needs to operate in two phases: one to compute the new values, and another to make those new values current.

The ability to annotate boxes makes this simple to implement. All that is necessary is to modify the global `:set` method so that instead of actually altering the value of a box, it will store the new value in an annotation on the box. Then this action is made conditional on the value of a variable `*phase*`. The master anima driver now will step all animas with `*phase*` set to 1, then go back and update the actual values of any changed boxes. The implementation of this is shown in Listing 5.2. This code basically says that when `*phase*` is 1, the new value is stored on the annotation named `:new-value` of the given box, and the box is saved in a global list for later processing in phase 2. If `*phase*` is 2, a normal set is done using `set-slot-1`, the primitive function for changing a slot's value. The anima driver is changed so that it drives the two phases alternately.

```
(defvar *phase* 0)
(defvar *phase2-boxes*)

(def-lw-method :set #/global (new-value)
  (case *phase*
    (1 (progn
        ;; use lower-level mechanism to save new value in annotation
        (set-slot-1 (getb-force self :new-value) new-value)
        (pushnew self *phase2-boxes*)
        new-value) ; return the new value
      ((0 2) (set-slot-1 self new-value))))

  (defun run-animas ()
    (let ((*phase* 1)
          (*phase2-boxes* nil))
      (dolist (a *running-animas*)
        (send a :step))
      (setf *phase* 2)
      (dolist (f *phase2-boxes*)
        (send f :set (slot f :new-value))))))
```

Listing 5.2: two-phase slot setting in Simple Agents.

In addition to properly simulating concurrency, this two-phase-clock technique has an additional advantage and one severe drawback. The advantage is that it also gives us the hooks to deal with conflict in a systematic way, a topic dealt with in the next section. The drawback is that it significantly changes the semantics of the languages. Under the two-phase clock regime, if a method sets a slot value and then uses that slot value in further computation, it will no longer obtain the right answer. This serious problem is somewhat offset by the fact that agents are usually quite simple and rarely need to examine the value of a slot that has been set in the same cycle.

5.1.1.2 Handling Conflict

Delaying the operation of `:set` to achieve concurrency also gives some leverage for dealing with conflict. A conflict occurs when two or more agents try to specify different values for the same slot. If this happens under the system as described above, we will again suffer order-dependencies among the agents (since the last agent to run during phase 1 will be the one that

ends up supplying the new value). To deal with this, we can extend the above scheme to keep track of *all* the **:sets** done to a slot during the course of a single agent cycle (during phase 1), and then during phase 2, figure out for each slot which value will prevail.

```
(def-lw-method :set #/global (new-value)
  (case *phase*
    (1 (progn (push (list new-value *agent*)
                    (slot self :proposals))
              (pushnew self *phase2-boxes*)))
    ((0 2) (set-slot-1 self new-value))))

(defun run-animas ()
  (let ((*phase* 1)
        (*phase2-boxes* nil))
    (dolist (a *running-animas*)
      (let ((*agent* (box-home a)))
        (send a :step)))
    (setf *phase* 2)
    (dolist (f *phase2-boxes*)
      (send f :resolve-set))))

(def-lw-method :resolve-set #/global ()
  (let ((proposals (slot-or-nil self :proposals)))
    (case (length proposals)
      (0 (error "No proposals for ~A" self))
      (1 (send self :set (caar proposals)))
      (t (send self :resolve-conflict)))
    (setf (slot self :proposals) nil)))

(def-lw-method :resolve-conflict #/global ()
  (let* ((proposals (slot self :proposals))
        (best (maximize proposals
                    :key
                    #'(lambda (proposal)
                        (let ((agent (cadr proposal)))
                          (or (ask-if agent :urgency)
                              0))))))
    (send self :set (car best))))
```

Listing 5.3: Conflict handling in Simple Agents.

To make this work, we need some way to arbitrate between conflicting agents. This is accomplished by equipping them with an **:urgency** slot. When an agent sets a slot's value, a pointer to the agent gets saved along with the proposed new value, so the phase 2 code can compare the urgency values of conflicting agents (we assume that no agent sets the same slot twice on the same cycle).

In this version of the system, during phase 1 each slot collects *proposals* for what its new value should be. Proposals consist of the new value and a pointer to the agent that is responsible for suggesting it. During phase 2, the proposals are evaluated and resolved by the **:resolve-set** and **:resolve-conflict** methods. **maximize** is a utility function that will select an element from a list (in this case the list of proposals) that yields the maximum value of a key function (in this case, the urgency of the agent that generated the proposal). The code above is not as efficient as it could be, since there is no real need to store all the proposals and find the

maximum later, but we will see that keeping the proposals around will be useful for generating user displays and for establishing alternative conflict resolution strategies.

Since urgency values are just slots, they can be dynamically updated by additional agents (for instance, the urgency of a find-food agent might increase with time and decrease when food was found).

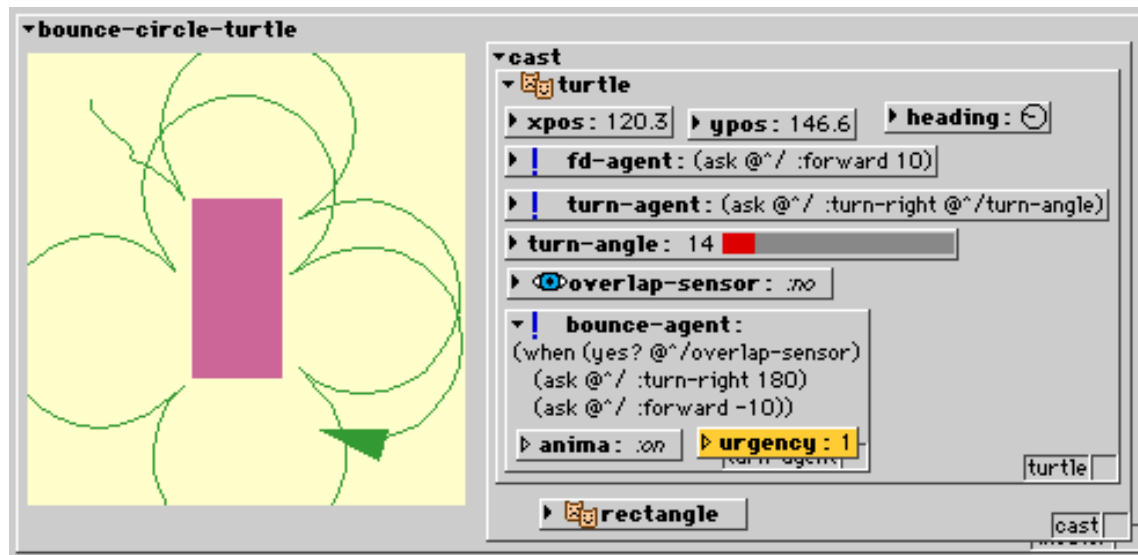


Figure 5.2: Three simple agents, with urgency.

An example where agents conflict is shown in Figure 5.2. This is the same as the previous example with the addition of a new agent, **bounce-agent**, a sensor, and a new rectangle actor. **Bounce-agent** checks to see if the turtle overlaps this rectangle, and if so it makes it back up and turn around. The **:urgency** slot of **bounce-agent** is 1, allowing it to supersede the other agents, which have the default urgency value of 0. The result is that the normal circling behavior prevails except when the turtle is in contact with the rectangle, when **bounce-agent** takes charge.

A few other things to note about this example: it is obviously necessary for **bounce-agent** to move the turtle as well as changing its heading, since otherwise, the turtle would still overlap the rectangle and the **:turn-right** action would be repeated on the next cycle. Less obvious is the fact that the two **asks** in **bounce-agent** are effectively done in parallel, because of the delay built into slot-setting in this agent system. That is why the turtle must back up rather than simply turning around and then going forward. Essentially, no sequentially-dependent operations can appear inside the action of a single agent when the two-phase-clock method is used. The two **asks** can appear in either order without changing the effect of the agent.

Note that this method of conflict resolution can result in part of an agent's action being carried out while another part is overridden by another agent with a higher urgency. In other words, conflicts are resolved at the slot level rather than at the agent level. Whether this is desirable or not depends on how one interprets the anthropomorphic metaphor. A partially-successful action might be interpreted as a compromise.

```

(def-lw-method :resolve-conflict #^/cast/turtle/heading ()
  (let* ((proposals (slot self :proposals))
        (average (/ (reduce #' + proposals :key #'car)
                    (length proposals))))
    (send self :set average)))

```

Listing 5.4: An alternative conflict resolution method.

Also note that other methods of resolving a conflict are possible. For instance, for a numerical slot like **heading**, the values of all proposals could be averaged rather than selecting one of them. These alternate resolution methods can be implemented on a per-slot basis by specializing the **:resolve-conflict** method. Listing 4 shows an example in which proposals to set the heading of any turtle will be averaged.

5.1.1.3 Presenting Conflict Situations to the User

The ability of the simple agent system to capture conflict can be exploited to generate displays for the user. This allows the user to specify the resolution to conflicts, and can automatically generate agent urgency values.

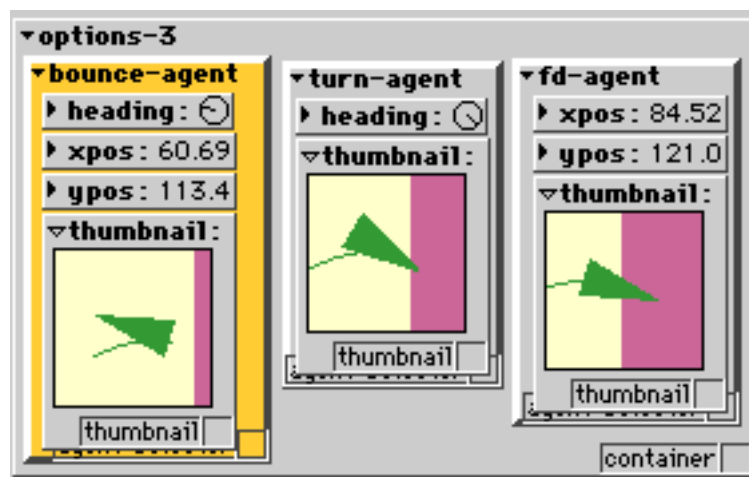


Figure 5.3: Presenting the user with an agent conflict situation.

An example of this is shown in Figure 5.3. In between phase 1, when each agent has made its attempt to set slots, and phase 2, when actions must be selected and performed, the system surveys the changed slots for agent conflicts. If conflicts are found that cannot be arbitrated by the urgency mechanism (because the agents involved have equal urgency), the system generates a display illustrating the effects of each agent and presenting the choice of agents to the user. Each agent generates a corresponding box containing copies of the slots it effects, and a thumbnail snapshot of the world as it would be if that agent alone was allowed to execute. Thumbnails are based on a special kind of slot whose value is a picture. In this case, the conflict is a complex one: **bounce-agent** is in conflict with **turn-agent** over the **heading** slot, and in conflict with **fd-agent** over the **xpos** and **ypos** slots.

The user then has to select some combination of agents that avoids conflicts. In the figure, **bounce-agent** could be selected by itself, or **turn-agent** and **fd-agent** could be selected

together, since they don't conflict with each other. The system can then adjust the urgencies of the selected agents so that this particular conflict will be arbitrated the same way automatically in the future.

In a more complicated system (for instance, one with multiple active actors) there are likely to be multiple sets of conflicting agents. This situation could be handled with multiple sets of thumbnails to choose among, although this could get confusing rather quickly.

5.1.1.4 Behavior Libraries

The simpler agent systems lend themselves particularly well to a mode of programming in which behaviors are selected from a library and cloned and dropped into an actor. An example of such a library is shown in Figure 5.4. This library contains both simple agents and goal agents (which are described below). Library agents generally refer to other objects through relative boxpath references (see 4.5.4) so that their clones will work properly in any context. Additionally, library agents may refer to slots of their own, which allows them to be parameterized (see behavior **get-to** in the figure for an example).

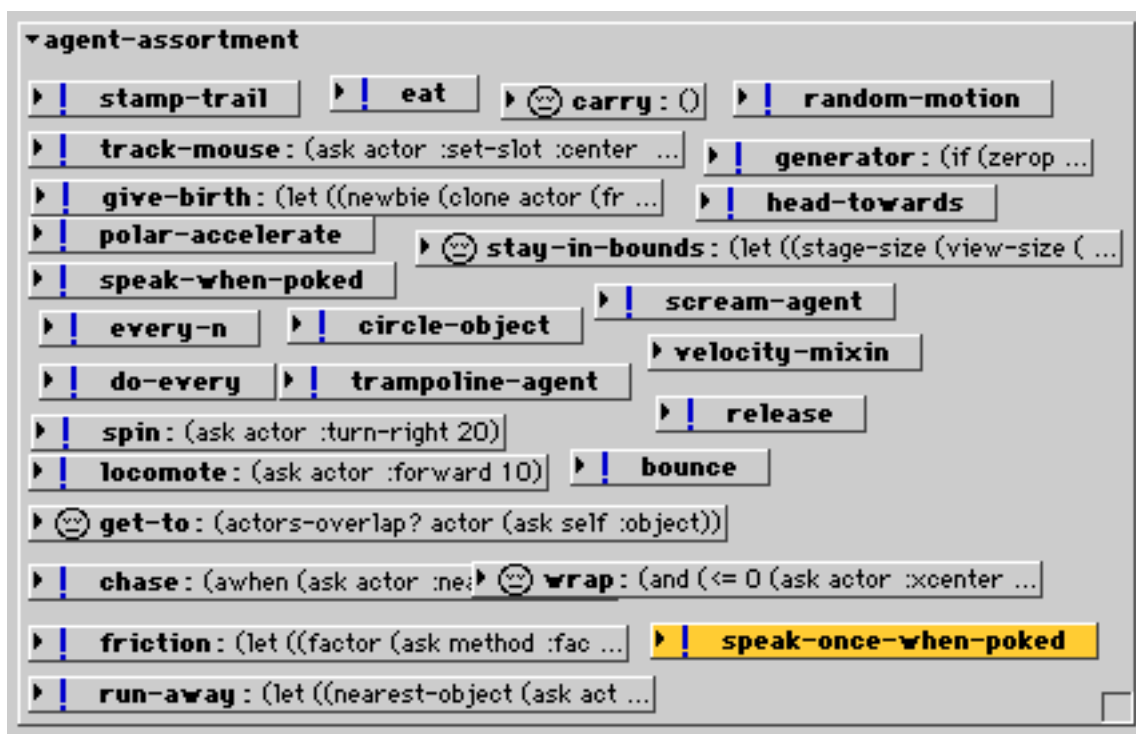


Figure 5.4: A library containing a variety of drop-in behaviors.

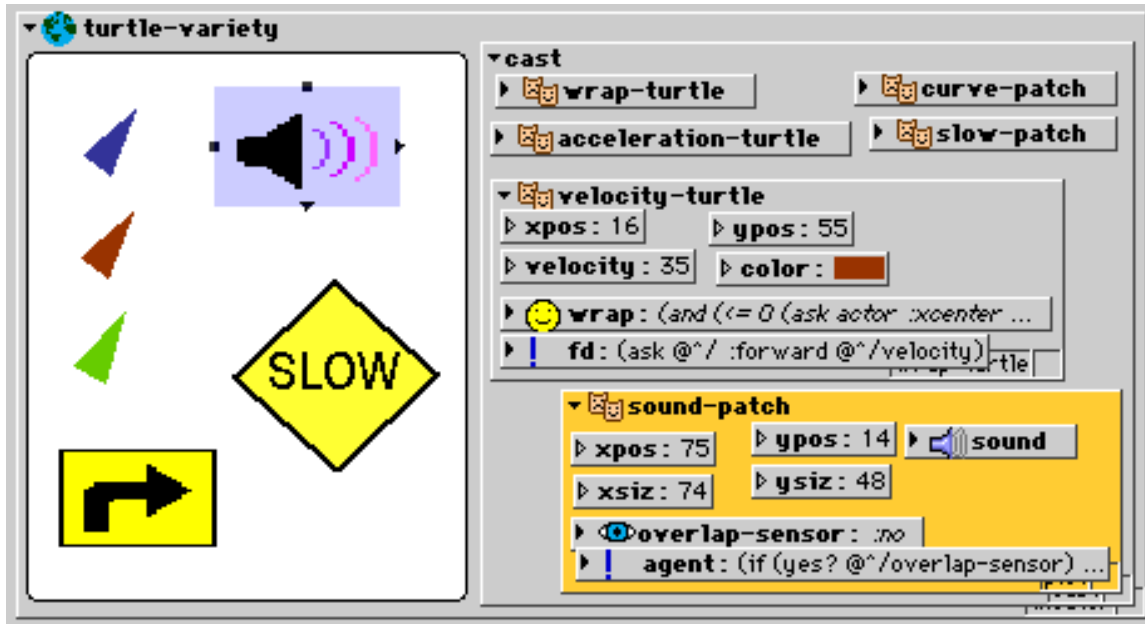


Figure 5.5: A library of objects containing agents.

Figure 5.5 illustrates how agents can be embedded into a library of other objects. This library contains a variety of turtles and patches (actor objects that are stationary but do something when another object contacts them). **Velocity-turtle**, for example, has a simple agent, **fd**, that gives it a constant forward velocity, and another Goal Agent (see the next section) that implements the wrapping behavior seen in most Logo environments. The **sound-patch** object plays a sound when another object passes over it (the sound, like the agent that generates the behavior, is an internal annotation to the patch object itself). The other patches, **curve-patch** and **slow-patch**, will change the motion of turtles that pass over them.

5.1.2 Goal Agents

This section describes a slightly more sophisticated form of agent-based programming called *Goal Agents*. This system illustrates several additional aspects of agents. Specifically, it shows how agents can be based around goals, and how this further enables the use of simple anthropomorphic metaphors to illustrate agent activity.

A goal-agent consists of a box whose value is a goal in the form of a Lisp predicate expression. The agent's task is to keep this goal satisfied. In addition to the goal itself, the box display includes an icon that indicates the state of the agent. A satisfied goal displays a smiley face icon, while an unsatisfied one produces a frown. Goal agents have a few other properties which are stored as internal annotations. Of these, the most significant is the *action*, which specifies how the agent is to attempt to satisfy its goal.

In Figure 5.6 we see an instance of a satisfied goal agent inside an actor (Opus). The agent's goal is that the overlap-sensor have a value of **:no**, or in other words that its parent actor should not be in contact with any other actor.



Figure 5.6: A satisfied goal-agent.

In Figure 5.7 the turtle has moved, causing the agent to become unsatisfied, as indicated by the changed icon. The agent has been opened to reveal the action within. This action will execute repeatedly until the agent is once again satisfied. In this case the action moves the actor 20 pixels in a randomly chosen direction, so eventually (assuming the turtle is static) Opus will elude the turtle and the goal will once again be satisfied.

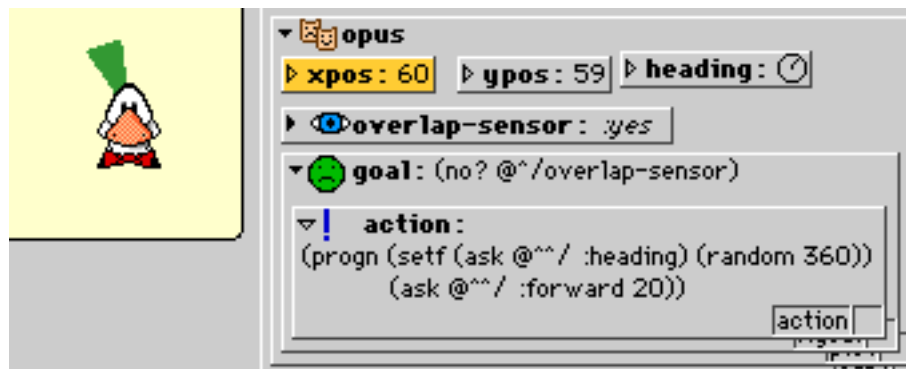


Figure 5.7: The goal-agent, unsatisfied and open.

Goal-agents act somewhat like an iterative construct of a procedural language, such as the **repeat...until** construct of Pascal, running the action until the test is satisfied. Goal Agents are also somewhat similar to situation/action rules, with the difference that instead of specifying a triggering condition and action, a goal-agent contains the *negation* of the triggering condition as its goal, and acts only when this goal is unsatisfied, or as the ethologist Hinde put it, “The concept of a goal refers always to a situation which brings a sequence of behaviour to an end.” (Hinde 1978, p. 624).

It’s worthwhile noting that this agent (and all the other types of agents presented in this chapter) operate concurrently with user interaction. For example, the above agent allows the user to chase Opus around by dragging another object. It is this property, the ability to interact with running agent programs, that gives LiveWorld its feeling of liveliness.

Goal-agents can also have precondition, urgency, and satisfied-action slots. The **precondition** slot contains another predicate and determines whether or not the agent is

allowed to run its action. **Satisfied-action** is an action to take if the goal is *satisfied*, as opposed to the normal action which is done if the goal is unsatisfied. The satisfied-action slot is included mainly for the sake of symmetry, although since it is somewhat in conflict with the goal-seeking metaphor its use is discouraged.

```
(def-lw-method :step #/goal-agent ()
  (if (ask self :precondition)
      (if (ask-self self)                                ; are we satisfied?
          (progn (ask self :satisfied-action)
                  (setf (ask self :icon) *smile-icon*))
          (progn (ask self :action)
                  (setf (ask self :icon) *frown-icon*))))))
```

Listing 5.5: Step method for goal agents.

Goal Agents (as described so far) can be implemented by equipping a LiveWorld object with the **:step** method shown in Listing 5.5. Implementing goal-agents also requires installing defaults for the slots involved, that is, goal agents must have a default **:satisfied-action** method that does nothing, and a default **:precondition** that is always true. Some other mappings into emotional icons are possible; for instance, an agent whose goal was unsatisfied but could not act because of a precondition might appear as frustrated.

As described so far Goal Agents don't really have much more computational power than Simple Agents, since simple agents can simulate the goal/precondition/action mechanism of goal agents with ordinary Lisp conditionals. However, Goal Agents provide some additional *structure*, in that the separate parts of the conditional are now separate objects and can thus be accessed separately. This means that the system now has the ability to know more about the state of agents. This property is what enables the anthropomorphic icon display. It also enables some alternative control strategies. Agents can test the satisfaction condition of other agents, and conflict resolution strategies can be used that take into account the knowledge about which agents need to run, based on the state of their goals. This permits agents to be arranged in sequences, albeit awkwardly (see the next section).

A useful variant of goal agents can be made by replacing the goal with a condition. The result is an *if-then-agent*, which consists of three clauses corresponding to the usual parts of a conditional statement: if, then, and else (see section 5.4.1 for an example). Sometimes this is a more natural way to formulate an agent, but because it is not based on a goal it is less amenable to anthropomorphization.

5.1.3 A Comparison: Teleo-Reactive Programming

It is instructive to compare Goal Agents with other goal-based formalisms for controlling action. The teleo-reactive program formalism (Nilsson 1994) was developed to address one of the same issues that motivated LiveWorld: the fact that sequential programming is not always suited to the control of agents that are in a continuous relationship with their environment. Other languages at this level include GAPPs (Kaelbling 1988), Chapman's circuit-based architecture (Chapman 1991), and universal plans (Schoppers 1987).

The teleo-reactive program formalism is particularly simple to understand. A teleo-reactive program consists of an ordered set of rules, rather like production rules. Unlike traditional production systems, however, actions are not discrete events (such as adding a token to a list) but durative actions like *move*. Durative actions persist as long as the triggering condition remains the first satisfied condition in the program. Nilsson suggests thinking of the program in terms of a circuit, which is constantly sensing the world and turning on a specific action. Equivalently, one can think of the T-R program as being called repeatedly in a loop whose cycle time is short compared with the typical length of time of actions. Rules are tested in order until one of the left-hand sides is true, and when one is found its right-hand side action will be executed. Then the process is repeated. For instance, here is a sample program (from Nilsson) that is intended to get the actor to navigate to a bar and grab it:

```

is-grabbing    → nil
at-bar-center & facing-bar → grab-bar
on-bar-midline & facing-bar → move
on-bar-midline → rotate
facing-midline-zone → move
T              → rotate

```

The equivalent program in Lisp would be something like this, with the understanding that the program is enclosed in a rapidly-repeating loop:

```

(if is-grabbing
  nil
  (if (and at-bar-center facing-bar)
    grab-bar
    (if (and on-bar-midline facing bar)
      rotate
      (if ...

```

While this language is rather simple, there are a few nonintuitive peculiarities about it. One thing to note is that actions are not directly associated with their goals. Instead, the action of rule n is intended to achieve the goal of rule $n-1$. Another is that considered as a plan for action, it reads backwards: the last action to be performed, *grab-bar*, comes first.

To express the equivalent program with Goal Agents requires creating an agent corresponding to each rule as follows (the notation G_n is short for “agent n ’s goal is satisfied”):

n	<i>goal</i>	<i>action</i>	<i>precondition</i>
5	is-grabbing	grab-bar	G_2
5	at-bar-center & facing-bar	move	$G_3 \ \& \ \sim G_1$
5	on-bar-midline & facing-bar	rotate	$G_4 \ \& \ \sim G_1 \ \& \ \sim G_2$
5	on-bar-midline	move	$G_5 \ \& \ \sim G_1 \ \& \ \sim G_2 \ \& \ \sim G_3$
5	facing-midline-zone	rotate	$T \ \& \ \sim G_1 \ \& \ \sim G_2 \ \& \ \sim G_3 \ \& \ \sim G_4$

Table 5.1: A teleo-reactive program expressed in goal agents.

This transformation makes for a more natural modularity (actions are associated with the goals they achieve) but necessitates explicit preconditions, rather than having them be implicit in the order of productions. There is no simple way to express any ordering of the goals.

One problem with this is that the preconditions get rather complicated for long action chains. Agent n has to check that agent $n+1$ has been satisfied *and* check that all agents $1...n-1$ have *not* been satisfied! The reason for this is our refusal to have any sort of central arbitrator or serializer that decides which agent should run. Instead we want each agent to figure out for itself if it is applicable, but that means that each agent has to know about all the other agents involved that come before it in the sequence. This is hardly modular.

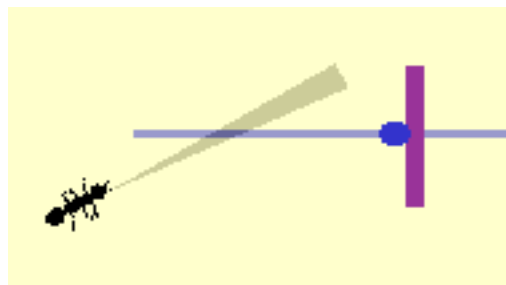
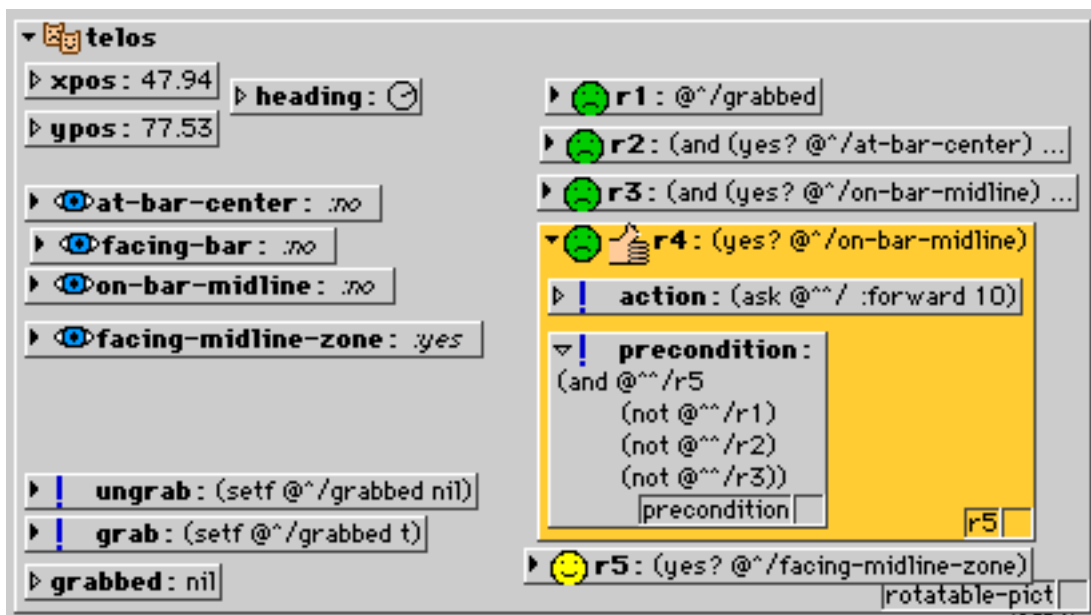


Figure 5.8: A teleo-reactive program expressed as Goal Agents. The long visible sensor of the ant is the **facing-midline-zone** sensor; other predicates are implemented with overlap-sensors. The bar is the thick rectangle, the bar midline is visible as a guideline.

The Goal Agents system thus provides a richer way to allow agents to interact than does Simple Agents, but still does not make it very natural to talk about sequences of action. This issue (and others) are addressed by Dynamic Agents.

5.2 Dynamic Agents

The agent-based programming systems introduced so far are quite limited in their expressive power. While agents can pursue goals, they are limited to a single fixed method for achieving them. Agent systems have fixed sets of agents and cannot readily modify their goals during execution. There is no easy way to specify sequences of action or lists of goals to be achieved sequentially. This essentially means that actors driven by simple agent systems are limited to a single fixed response to a given situation.

Dynamic Agents (DA) was designed to overcome some of these limitations, and is the most powerful agent language implemented within LiveWorld. In contrast to earlier systems in which agents were basically static entities, in Dynamic Agents, new goals can be created on the fly and agents created to realize them. As in Goal Agents, agents are responsible for a single goal or task. But to carry out their task, they can create new tasks which in turn results in the creation of new, subordinate agents. The agents in a DA system thus form a hierarchy, but a hierarchy that can dynamically change over time.

Imagine an agent as a person. It has a job to do, which can involve either keeping a goal satisfied or carrying out an action. It is created and activated by its superior, but otherwise acts autonomously. Its actions can consist of direct changes to the world, or it can recruit subordinate agents to pursue subtasks that contribute to its own success.

In terms of the anthropomorphic metaphor, the hierarchy consists of managers and workers. Managers can have goals, which they monitor but take no direct action to achieve. Instead they create and supervise the activity of other agents. Worker agents actually make changes to the world. Managers have some flexibility in the manner in which they activate their subordinates. For example, an agent to stack blocks might have a subordinate in charge of finding blocks, another in charge of moving the hand to the block, another to grasp the block, another to move the hand so that the block is on top of the stack, and another to release the block from the hand. This agent would want to activate its subagents sequentially and in a fixed order. An agent which maintains a set of graphic constraints would have a subagent for each constraint, but would typically not care about the order in which they were achieved. It would, however, need to make sure that one agent did not interfere with the goals of another agent. The language for specifying agents has to be flexible enough to accommodate these different sorts of managers.

	little-person model	Tinbergen model	dynamic agents
<i>basic model</i>	procedural	reactive	task-driven
<i>units</i>	“little people” (procedure invocations)	drive centers	agents
<i>persistence</i>	dynamic	static	dynamic
<i>goals</i>	no support	implicit	explicit
<i>action</i>	executing procedures	releasing energy	performing tasks
<i>communi- cation</i>	LPs ask other LPs to perform procedures	passing energy	agents post tasks which generate new agents
<i>conflict</i>	no model	intra-layer inhibition	slot conflict and goal conflict
<i>multiple methods</i>	one LP per procedure call	one unit per behavior	can have multiple agents for a task
<i>failure handling</i>	none	none (except repetition)	agents can fail, allowing other methods to be tried
<i>state</i>	running, dormant, waiting	blocked; activation energy	dormant, [un]satisfied, failed, succeeded, determination
<i>concurrency</i>	exactly one LP active at a time	one active unit per layer	multiple active agents
<i>top</i>	“chief LP” initiates activity	top-center	God agent organizes activity
<i>bottom</i>	primitive procedures	motor units	worker agents

Table 5.2: Programming models compared.

Two earlier forms of hierarchical control system have influenced the design of the agent hierarchy underlying DA. The first of these is the hierarchy of procedure calls generated by the activity of an interpreter for a procedural programming language, and the little-person metaphor sometimes used to illustrate it (see 3.3.2.1). The notion of a society of experts performing tasks on demand is retained, but extended in several significant ways. A little person executes a procedure once then disappears, but a dynamic agent is a persistent entity. This allows agents to have significantly greater capabilities. Agents operate in parallel, they can monitor goals as well as carry out procedures, and there can be more than one agent created to accomplish a

given task. Because agents can have goals, there is a basis in the model for thinking of agents as satisfied or unsatisfied, successful or failed, and for handling conflict.

The other major model for DA's hierarchy of agents is Tinbergen's model of animal behavior control systems (Tinbergen 1951), and computational attempts to realize it such as (Travers 1988) and (Tyrrell 1993). Like Tinbergen's drive units and releasing mechanisms, agents in a DA hierarchy are constantly checking their state against the world, and so can take control when necessary, possibly overriding other agents. This flexibility is not found in the standard procedural models of programming. In some sense, DA can be understood as an attempt to integrate the reactive hierarchies of Tinbergen with the formal simplicity and power of the procedural model.

Table 5.2 shows a point-by-point comparison of dynamic agents with the little-person model of procedural programming and Tinbergen's model of animal behavior. The basic metaphor behind procedural programming is serialized following of instruction, while in the Tinbergen model it is reacting to environmental stimuli. Dynamic Agents attempts to integrate these by basing its model of an agent on tasks which can specify both sequences to execute and goals to be achieved. From the LP model, DA takes the idea that agents can dynamically create subagents as part of their activity. From the Tinbergen model, DA takes the idea that agents while arranged in a hierarchy have a measure of autonomous control, can run concurrently, can constantly check on the state of the world and can seize control from each other. The possibility of combining the best features of these two rather different models of control was the inspiration behind the design of Dynamic Agents.

5.2.1 Overview

This section presents a technical overview of the structures, concepts, and operation of the Dynamic Agents system. The DA system works by building a hierarchy of agents (see Figure 5.8) based on user specified tasks. In a cyclic process, the agents are activated and may take actions which consist of either a direct action which changes the world, or the creation of subtasks and subagents. Agents may come into conflict by attempting to perform incompatible actions or by interfering with another agent's goal. The system attempts to resolve conflicts and schedule agent activity in such a way as to satisfy as many agents as possible.

Each agent is in charge of a single task, which determines its type and behavior. *Primitive tasks* specify a direct action to take in the world. Other kinds of tasks can specify goals to be achieved, or complex procedures to perform that invoke a number of subtasks. Tasks are specified using a vocabulary of built-in control constructs and user-defined templates. For instance an **and** task specifies a number of subtasks, all of which must be satisfied simultaneously for the **and** task to be satisfied. A **script** task, on the other hand, activates its subtasks sequentially. Agents have a numerical *determination* value which corresponds roughly to how hard they try to take their action in the face of conflicts.

At the top of the agent hierarchy is an agent with a special task called (**god**). This agent serves to organize the activity of the next lowest layer of agents, which embody top-level tasks.

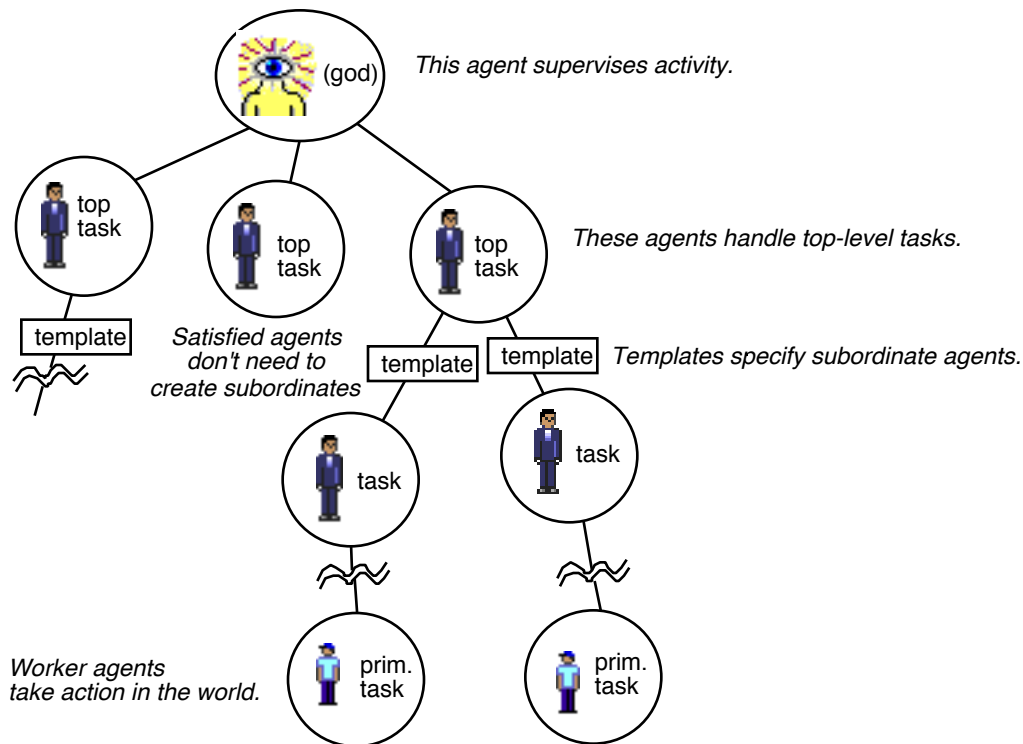


Figure 5.9: An agent hierarchy.

These tasks can be specified explicitly by the user in task objects, or arise implicitly from user actions. They can also be part of objects in a simulation or library.

Agents can be in a number of different states. Agents whose task is a goal can be *satisfied* or *unsatisfied*. Agents that carry out an action can be *successful*, *failed*, or *waiting* for a subordinate. Worker agents succeed when they are able to carry out their action. They can fail for a variety of reasons, including coming into conflict with other agents. Manager agents generally succeed or fail based on the success or failure of their subordinates. For example, the failure of a single subordinate will cause an **and** agent to fail, while a similar subordinate failure under an **or** agent will instead cause a different subordinate to be activated.

The agent interpreter begins its process of activating agents from the top agent and works its way downwards. At each level, the task of an agent determines which subagents, if any, will become activated. For instance, a **script** agent will only activate one subagent, whereas an **and** agent will activate all of its subagents. An agent that is satisfied needs to take no action and will not activate any subagents.

Eventually this process bottoms out in worker agents who attempt to execute primitive tasks. The actions of these agents are deferred so that concurrency and conflict resolution can work right, in a manner very similar to that used in simple agents (see 5.1.1.2). The conflicts are resolved using the determination values of the agents involved. Agents that prevail (or are not in conflict) are marked as successful, while losers are marked as failed. Success and failure propagate upwards in a manner determined by the tasks involved. The successful actions are

realized, that is, their proposed changes for LiveWorld slots become the current values. This may cause other agents to become unsatisfied. Agent determination values can change as other agents fail and succeed, so that conflict resolution is not permanent—the “balance of power” can shift. This cycle is repeated either indefinitely or until all top-level tasks are satisfied.

5.2.2 Structures

This section explains the major structures used in dynamic agents: tasks, agents, and templates. To briefly summarize the relationship between these three separate but closely related concepts:

- A *task* is an expression that specifies a job to do;
- An *agent* is a structure created to carry out a particular task;
- A *template* is a structure that specifies, for some set of tasks, how to create agents to carry them out.

Templates extend the kinds of tasks that can be handled and thus in some sense constitute the program specification of an agent system. To extend this rough analogy with traditional procedural programming, tasks correspond to procedure calls, while agents correspond (roughly) to the stack frames or other internal structures used to keep track of procedure invocations. However, agents are longer-lived, keep more information, and have more complicated associated control mechanisms than stack frames.

5.2.2.1 Tasks

A *task* is an expression in Lisp syntax that represents either a goal to be achieved or an action to be taken. While tasks are in the form of Lisp structures, they are not (in general) interpreted as Lisp expressions. The first element of a task specifies the general type of task, and may be a special symbol such as **and** or **do**, or the name of a domain task. Because tasks and agents are closely coupled, we will use the terminology for tasks and agents interchangeably (for instance, we might refer to the type of agent, which is simply the type of its task). There are a number of different general classes of task:

- *goals* are tasks that have an interpretation as a predicate (in other words, they are satisfiable. Some special tasks (such as **and**-type tasks) are also goals, and tasks that are in the form of user or system-defined predicates are also goals.
- *actions* are tasks that are *not* predicates.
- *primitive tasks* are action tasks that take direct action in the world.
- *manager* tasks include all non-primitive tasks. Agents with manager tasks take no direct action on the world. Instead they specify and control some number of subordinate agents.
- *special tasks* are manager tasks whose type is a member of the vocabulary of control constructs presented below. They generate their subordinate agents according to rules presented in section 5.2.4. In general, special tasks have one or many subforms, each of which is a task and generates a single subordinate agent.

- *domain tasks* are manager tasks with a user-defined, domain-dependent type. Domain tasks generate subagents according to templates that match the task.

To summarize: tasks can be split along several dimensions into disjoint pairs of classes. Goals and actions are one such pair, as are manager and primitive tasks; and special and domain tasks. For instance, an **and** task is a special task, a goal, and a manager task.

Some sample tasks are presented here. More detailed examples may be found later (see especially section 5.4).

```
(= #^/rect/xsiz #^/rect/ysiz)
```

This task is a goal since it has a predicate value. The task it denotes is to keep the values of two slots equal (it ensures that the **#/rect** object is a square). An agent for this task would take no action if the task was satisfied. If it was unsatisfied, the agent would create subagents to try to achieve satisfaction.

```
(go-to #^/ant #^/food-3)
```

This is a user-defined goal task. The goal is satisfied when the **ant** is in contact with **food-3**, and when unsatisfied will create subagents that will cause the ant to move towards its goal. These subagents will manage the ant's direction and movement.

```
(do (ask #^/turtle :forward 20)
```

This is a primitive task. An agent with this task will make the turtle go forward when it is activated.

```
(script (find-free-block @^/marker/)
        (go-to @^/hand @^/marker/)
        (grasp @^/hand))
```

This script task from the blocks world instructs the agent to make three subagents and activate them serially. The first instructs a marker object to find a free block, the second instructs the actor representing the hand to go to the marker, while the third instructs the hand to grasp whatever object it finds. All three of these subtasks are high-level domain tasks that generate manager agents.








5.2.2.2 Agents

An agent is a structure created to carry out and monitor a task. Agents are arranged hierarchically as described above. Internally agents are represented by low-level Lisp structures for efficiency, but they also have representations in box form for presentation to the user.

The most important property of an agent is its task. An agent also contains an assortment of additional state variables that are used to control its activity. This includes its precondition, its current subordinate agent (for script tasks and others that activate only one subagent at a time), its determination, and its success or failure state including information about lost conflicts. Agents also contain the necessary pointers to maintain the agent hierarchy. If an agent was

generated by a template, it contains a pointer back to that template along with the variable bindings that resulted from the match of the template's task-pattern to the agent's task.

Agents are intended to be understood anthropomorphically as the active and interested components of the system. To that end, a set of icons are used to represent the state of agents in terms suggestive of human emotion. In addition to the icons for goal satisfaction used in goal-agents (see 5.1.2), dynamic agents can express the state of their action (succeeded, failed, or waiting) and the presence of conflict. The icons and their interpretation are:

- inactive (asleep) 
The agent is inactive.
- satisfied (happy) 
The agent is active and its goal is satisfied.
- unsatisfied (sad) 
The agent is active and its goal is not satisfied.
- in conflict (angry) 
The agent *was* satisfied, but was made unsatisfied by another agent.
- failed (thumbs-down) 
The agent failed.
- succeeded (thumbs-up) 
The agent succeeded.
- waiting (tapping fingers) 
The agent is waiting for a subordinates to complete.

Roughly, an iconic face is used to indicate the satisfaction state of the task, while an iconic hand is used to show the state of any action taken to achieve that task. These are somewhat independent, since an agent's task can become satisfied without direct action by the agent.

5.2.2.3 Templates

A template specifies a method for constructing an agent to handle a particular class of domain task. The relationship between templates and agents is similar to the relationship between a procedure specification and a procedure invocation in a standard procedural language: that is, the template is a static, generalized structure that serves as a pattern for dynamically creating other structures (agents) to deal with particular instances of tasks. Thus, templates correspond to what is normally thought of as "the program", that is, they contain the specifications of what the computer is to do.

When an agent with a domain (non-special) task is expanded, the system matches its task against all¹⁸ templates. If exactly one template matches, then a single subagent is created using that template's information. If more than one template matches the task, an intervening ONEOF task is created to manage the multiple agents that result.

Templates consist of a number of fields. Some of these have values that are patterns used to match and generate tasks. The most important of these are the task-pattern and the action-pattern, which specifies a new task. The semantics of a template are (roughly) that if there is an active task that matches the task-pattern of the template, then a subagent should be built that incorporates the new task specified by the action-pattern. Templates can also specify preconditions for the agent and other relevant agent properties.

In this document, templates are shown defined by Lisp forms. It is also possible to create templates in the form of box structures within LiveWorld, by the usual method of cloning a specialized object and filling out the appropriate slots.

Patterns are specified using a pattern-matching language based on the one in (Norvig 1992). A pattern consists of a Lisp list in which some of the elements are pattern variables. Pattern variables are symbols beginning with the **?** character. For example, the pattern **(= ?a ?b)** would match the task **(= @^/rect/x @^/rect/y)**, binding the variables **?a** and **?b** to the appropriate matched values. This information is used to substitute the values for pattern variables in the other patterns in the template, such as the action.

Here is an example of a template from an animal-behavior simulation:

```
(deftemplate (get-food-forward ?a)
  :precondition ^ (and (yes? (ask ?a :left-food))
                      (yes? (ask ?a :right-food)))
  :action ^ (repeat (do (ask ?a :forward 10))))
```

The task-pattern of this template is **(get-food-forward ?a)**, and the effect of the template is to say that if a task of that form is asserted (for some actor that will match the pattern variable **?a**), and the two food sensors are both returning **:yes** values, then go forward. The pattern variable **?a** will be bound to the actor when the template is matched against outstanding tasks, and the precondition and action are evaluated in a context that includes the resultant bindings of the pattern variables.

In the above example, the pattern matching facility does not add much power over and above the agent systems already seen. However, the next example uses multiple pattern variables to establish templates for tasks that involve relations between slots:

```
(deftemplate (= (+ ?b ?c) ?a)
  :name +left
  :precondition ^ (settable ?b)
  :action ^ (= ?b (- ?a ?c))
```

¹⁸ Actually the implementation organizes templates into buckets based on their first element in order to minimize the relatively expensive process of pattern-matching. This is an implementation detail.

This template specifies one method for satisfying a particular form of arithmetic constraint by reducing it to a simpler constraint. This template is part of the system's general arithmetic capabilities. As this example suggests, there can be more than one template matching a task. When a task matches with multiple templates, the system will generate multiple agents under a **oneof** agent (see 5.2.4.4).

The fields of a template specification are:

- The *task pattern* (the header or second element of the deftemplate form) is used to match the template against a task. It can contain pattern variables that get bound to elements in the original task.
- The *action pattern* (specified by the :action keyword) generates a new task based on the values of pattern variables bound in the goal pattern. This task becomes the basis for a subagent of the original task's agent. This field must be supplied (or calculated from the satisfied field, see below).
- The *precondition pattern* specifies a precondition that also becomes part of the newly created agent. The default value is t, that is, by default agents are always able to run.
- The *satisfied* field allows a predicate and action to be defined together (see below).
- The *name* field assigns the template a name that uniquely identifies it for purposes of debugging and redefinition. The default value of the name is the goal-pattern. This means that if multiple templates are being defined for a single goal-pattern, they had better have unique names so the system does not interpret them as redefinitions of one template.
- The determination-factor field is a number used to scale the determination value assigned to the agent (see 5.2.7). The default value is 1.

Users can define their own goals by using the **:satisfied** parameter of templates. This essentially defines both the declarative and procedural parts of a high-level goal simultaneously. For instance,

```
(deftemplate (square ?a)
  :satisfied (= (ask ?a :xsiz) (ask ?a :ysiz)))
```

This template definition both defines a one-argument Lisp predicate **square** with the obvious definition, and creates a template that if given a square task will attempt to achieve it by posting the appropriate equality task. This process involves converting the task-pattern into a Lisp function specifier and argument list, and for this reason a template that includes a **:satisfied** field must have a task-pattern that is of the form (**<symbol> <pattern-var>***).

The pattern-matcher is actually more powerful than described so far. Most of its capabilities (for instance, matching a variable to a segment of arbitrary length) have not been used. One extension that has proved useful is syntax that allows patterns to specify that a pattern variable must match a box of a particular type. For instance, the pattern

```
(yes? (?spinoff ?sensor #/sensors/overlap-sensor))
```

will match any task of the form (**yes? foo**), if and only if **foo** is a spinoff of **#/sensors/overlap-sensor**.

5.2.3 Control

From the implementation standpoint, agents do not really operate independently but instead are controlled through a centralized agent interpreter. This interpreter is in charge of creating agents for the initial tasks, finding templates to match tasks and creating the new tasks specified by the templates, and so on recursively. The interpreter is also in charge of determining what subset of these agents are to be active at any time.

The interpreter is cyclic, that is, it consists of a process that repeatedly examines and updates the structure of agents. There are a number of different ways such an interpreter could work. For instance, it could pick a single agent to activate on each cycle, or pick a single agent on each level, or activate as many agents simultaneously as it can. To be true to the distributed form of the anthropomorphic metaphor, we have chosen the latter, even though the single-agent-at-a-time methods might be more tractable. So a single major cycle of agent activity consists of activating as many agents as are appropriate, and, if any actions are proposed, resolving conflicts and executing them. This section describes the internal mechanics of agent activation and control. For an extended example of the consequences of such activity, see section 5.3.5.

5.2.3.1 Activation and Expansion

Activation is the process of making an agent check its goal (if any) and take necessary action. Agents are activated in a top-down process, starting from the top-agent and working down to the worker agents. Activating one agent may result in the creation of subagents and their subsequent activation.

When an agent is activated, it first performs several checks. First, if the agent has already completed, nothing is done. This does not necessarily mean that each agent can run only once, since the completion state of an agent can be reset (see below). If the agent is not complete, it checks whether it may already be satisfied (if it is a goal) and whether or not its precondition is met (if it has one). After passing these checks, an agent has determined both that it needs to run and that it can.

At this point if it is a primitive agent it runs its action. Otherwise it expands itself and marks its subordinate agents for later activation. *Expansion* is the process of creating subagents for an agent. This involves different processes for different types of agents. For agents with domain-specific tasks, expansion involves finding templates that match the task and using them to create subagents. For special tasks, the process varies but most typically involves making one subagent per clause of the task. For the **god** task, expansion involves collecting all the top-level tasks and making a subagent for each. Each agent is expanded at most one time.

5.2.3.2 Cycles

The process of agent interpretation is essentially iterative. The system activates some set of agents, they perform their tasks, and the process is repeated. In the earlier systems this process was relatively simple, with an anima-based driver that clocked each agent independently, with a global conflict resolution phase. The dynamic agents system is

significantly more complex and requires a more complex driver. I have found it useful to think of three different levels of cycle for the different levels of agent activity:

- A *microcycle* is the smallest-scale cycle, and a single microcycle consists of activating a single agent.
- A *major cycle* or *millicycle* is a round of microcycles that starts from the top agent and proceeds downwards until the leaves of the agent tree are reached, which will be worker agents executing primitive tasks. This is followed by a conflict resolution phase and the actual state change. A major cycle is like a single unit of real-time, in which all the agents can act if they need to.
- A *macrocycle* consists of initializing the agent tree and performing millicycles until the top agent successfully completes. Note that a complete macrocycle is possible only if all the top-level tasks are able to be completed.

Microcycles are useful mostly for the purpose of user stepping, and macrocycles only make sense for constraint-like problems where all the top-level tasks are goals. The millicycle is really the fundamental unit of dynamic agent activity, and when the system is being driven by an anima (see 4.6.1) it is essentially performing a millicycle per anima clock.

5.2.3.3 Success and Failure

An agent can succeed or fail at various points in its activity. Success or failure halts the agent activity and signals the agent's boss agent, which will take action dependent upon its task.

Agents succeed when:

- a worker agent successfully executes its action.
- an agent with a goal-task is activated and finds its goal already satisfied.
- a subordinate agent succeeds and appropriate task-specific conditions are met.

Agents fail when:

- a worker agent gets a Lisp error.
- a worker agent loses a slot conflict to another agent.
- an agent's precondition is unsatisfied.
- no templates can be found for a domain-specific task.
- a subordinate agent fails and appropriate task-specific conditions are met.
- subordinate agents succeed but the agent's goal is still not satisfied.

Lisp errors may occur due to bugs in the specification of primitive tasks, of course, but they may also be generated by general templates being applied in situations where they don't work. For instance, an agent that attempts to satisfy the task ($= a (* b c)$) by setting **b** equal to ($/ a c$) will get a Lisp error (and thus fail) if **c** happens to be zero. When this happens, the Lisp error is caught by the agent interpreter, the agent is marked failed, and other methods are tried. This means that the user can be rather free in specifying methods.

The “task-specific conditions” mentioned above refer to the way in which agents react to success or failure for different special tasks. For instance, the failure of an agent whose boss is an **and** agent will cause the boss to fail as well. In the case where the boss is an **or** agent, however, the failure will cause a different subordinate of the boss to be activated during the next millicycle. What happens next depends on the parent’s action type. For instance, if a child of an agent with an **and** special action (which we will call an **and** agent for short) fails, then the parent agent will fail as well. If a child of a **script** agent succeeds, it makes the next sequential child the current one so that it will be activated on the next agent cycle.

Agent success and failure persists across cycles but may be reset in some circumstances, allowing the agent to attempt to run again. These circumstances include when a successful agent’s goal becomes unsatisfied (a goal conflict, see 5.2.6.2) or when a failed agent has its determination increased due to sibling failures.

5.2.4 Special Tasks

This section describes the vocabulary of control constructs that are used to create special and primitive tasks, and how they are interpreted by the agent system. The interpretations of other types of tasks (that is, domain tasks) are specified by templates. Special tasks allow tasks to be combined in different ways, and in some sense represent different “managerial styles” or ways of accomplishing a complex task by parceling it out to subordinate agents. Some tasks activate all their subordinates at once, while others activate them one at a time. Of the latter, some specify a fixed order of activation while others don’t care. Some tasks are goals that can be satisfied or not, while others are just actions. These distinctions are somewhat confusing, but each task represents a different strategy for accomplishing a complex task. Examples of special tasks in use may be seen in section 5.4 and elsewhere.

5.2.4.1 Combining Tasks

These special tasks serve to combine several subordinate tasks in various ways.

(AND goal*), (OR goal*)

These special task types are duals of each other in many respects. **And** and **or** tasks are goals, and their satisfaction is a function of the satisfaction of their subtasks (which must also be goals). An **and** agent activates all of its subagents simultaneously, and expects them all to be satisfied eventually. If any of them fail, the **and** agent fails. An **or** agent activates its subagents one at a time, in an unspecified order, choosing another if one fails and only failing if all subagents fail.

(ALL <task>*), (ONE <task>*)

All and **one** are similar to **and** and **or**, but are actions rather than goals, and can have actions as their subtasks. An **all** agent activates all of its subagents, in parallel, and completes successfully when all of the subagents complete successfully. A failed subagent will cause it to fail. **one** acts like **or** in activating subtasks one at a time in arbitrary order, but requires only one

successful subtask to succeed. In other words, whereas **and** and **or** pay attention to satisfaction, **all** and **one** only care about success.

(SCRIPT <task>*)

Script tasks are actions. A script causes its subordinate tasks to be activated one at a time, in the order specified. Each task must complete successfully before the next task is begun. **Scripts** succeed when the last task is done and fail if any subtask fails.

(TRY <task>*)

Try is similar to **script**, in that subtasks are activated one at a time and in the specified order. However, only one of the subtasks needs to succeed for the **try** task to succeed. In other words, **try** tries running each subtask until one succeeds, at which time it succeeds itself. Roughly, **try** is to **script** as **or** is to **and**.

(AND-SCRIPT <goal>*)

This special task combines the semantics of **and** with the sequential properties of **script**. Like the former, **and-script** tasks must be satisfiable goals and their subtasks must be as well. But unlike **and**, and like **script**, they activate their subordinates one at a time, in the order specified by the task.

The difference between **and-script** and **script** is that the former requires that its tasks remain satisfied after they are completed. That is, once a subagent of a **script** is finished, it no longer cares about its goal, but all steps of an **and-script** up to the current step must remain satisfied even after they are completed. If they become unsatisfied, the **and-script** fails (there is no **or-script** because its behavior would be identical to the more general **try** construct).

5.2.4.2 Control Tasks

(REPEAT <task>)

REPEAT tasks are actions that never complete, and thus cause their inner task to remain activated indefinitely. Generally, **REPEAT** tasks are used as the expansion of goals, so that the inner task will be executed only until the goal is satisfied. **REPEAT** is most useful for actions that achieve a goal only after multiple iterations. For an example, see the template for **GET-TO** tasks in section 5.2.5.3.

The **REPEAT** construct allows for iteration at the agent level. Iteration in primitive actions (that is, at the Lisp level) is discouraged, because it blocks other agents from running. Using **REPEAT** allows the iteration to be interleaved with other agent actions.

5.2.4.3 Primitive Tasks

(DO <lisp>*)

Primitive tasks are specified using the **do** task type. Primitive tasks are handled by worker agents that have no subordinates and always complete immediately. They can fail due to Lisp errors or by coming into conflict with a more determined agent. While primitive tasks can specify the execution of arbitrary Lisp programs, they are expected to be simple and fast, and will typically involve nothing more complicated than sending an object a message, or changing the value of a slot or two.

5.2.4.4 Internal Special Tasks

Some special tasks are for the internal use of the system, and are not meant to be used in user-specified tasks and templates.

(ONEOF task*)

oneof tasks are used only internally: they cannot appear in user-specified tasks or templates. They act essentially like **one** tasks, but are generated by the system when multiple subagents are required for a single domain task. So for instance, a goal of (= @/a @/b) would generate two subtasks corresponding to the two obvious ways to make an equality constraint be true:

```
(do (setf @/a @/b))
(do (setf @/b @/a))
```

Because domain tasks can have only one subagent, a **oneof** task and subagent is created to intervene between the equality constraint and the two methods. Here an outline notation is used to show the relationship between an = constraint, its subordinate **oneof** task, and two further subordinates:

```
a1: (= @/a @/b)
  a2: (oneof <a3> <a4>)
    a3: (do (setf @/a @/b))
    a4: (do (setf @/b @/a))
```

Note that since **oneof** tasks are used only internally, they are able to refer directly to their subagents. In actuality the treatment of equality is somewhat more complicated; see section 5.2.5.1 for more details.

(GOD)

The GOD special task is used internally by the system for the top agent. It gathers together all the top-level tasks, makes them into subagents, and otherwise acts essentially like an **all** task.

5.2.5 Domain Tasks and Templates: Examples

Domain tasks are interpreted according to templates; that is, the user specifies an interpretation for them. This section presents some detailed examples of domain tasks and the templates that interpret them.

5.2.5.1 Numerical Constraints; Equality

Numerical constraints are expressed as algebraic relations. For example, one way to express a task to maintain the relationship between Fahrenheit and centigrade temperature representations looks like this:

```
T1: (= @^/Fahrenheit
      (+ 32.0 (* 1.8 @^/centigrade)))
```

Equality constraints are usually handled by templates that propose two subagents: one that proposes to make the equality constraint hold by making the expression on the left-hand side be equal to the one on the right, and the inverse. These tasks are expressed by means of the `<==` operator, known as *one-way equality*. This operator has the same declarative meaning as `=`, that is, it is satisfied if its two arguments are numerically equal. Its procedural meaning is that equality can only be achieved by altering left hand side to match the right. A similar construct, `<=equal`, is available for non-numeric values. The basic definition of one-way equality is provided by the following set of templates:

```
(deftemplate (<== ?a ?b)
  :satisfied (= ?a ?b)
  :precondition `(settable? '?a)
  :action '(do (setf ?a ?b)))

(deftemplate (= ?a ?b)
  :name =left
  :action '(<== ?a ?b))

(deftemplate (= ?a ?b)
  :name =right
  :action '(<== ?b ?a))
```

The temperature conversion task would result in the two subtasks:

```
T3: (<== @^/Fahrenheit
      (+ 32.0 (* 1.8 @^/centigrade)))

T4: (<== (+ 32.0 (* 1.8 @^/centigrade))
      @^/Fahrenheit)
```

If the left-hand side of a one-way equality constraint is settable, it can be expanded into a worker agent by the first template shown above. Settable-ness is defined by the `settable?` function, and is true of anything that can be a legitimate first argument to the Common Lisp `setf` macro. `@^/Fahrenheit` is settable, so T3 can be expanded into:

```
T5: (do (setf @^/Fahrenheit
             (+ 32.0 (* 1.8 @^/centigrade)))
```

T4, however, has a left-hand side that is not settable. This does not mean the task is hopeless, however! Other templates can match it and transform it into subtasks that will eventually allow action to be taken:

```
(deftemplate (<== (+ ?b ?c) ?a)
  :name +right
```

```
:action `(<== ?c (- ?a ?b)))
```

This template says that a one-way equality task that is trying to change the value of a sum ((+ ?b ?c)) can be transformed into a new task that sets one of the items being summed (in this case, ?c). The template will match T4 and generate a new subtask:

```
T6: (<== (* 1.88 @^/centigrade)
        (- @^/Fahrenheit 32.0))
```

In effect, the template has performed an algebraic transformation on T4. This process of transformation continues, controlled by other templates which have forms similar to the **+right** template, until it results in a runnable worker agent. In this case, the penultimate task, which will expand directly into a worker agent, is:

```
T11: (<== @^/centigrade
        (/ (- @^/Fahrenheit 32.0) 1.8))
```

A rewriting system like this poses a danger of generating a non-terminating chain of transformed tasks, either through looping or through infinite expansion. The built-in set of templates for handling numerical constraints avoids this problem by ensuring that templates always generate new tasks whose left side is a sub-part of the left side of the original task. Since each transformation must produce a new task whose left side is strictly smaller than the starting task, the process of transformation must eventually terminate.

5.2.5.2 Geometry

Geometrical constraints ultimately turn into numerical constraints of the type shown above. The combination of the template pattern language and LiveWorld's object facilities allow them to be expressed as higher-level relationships.

For example, the (**rectangle <actor>**) form of goal is heavily used in graphic constraint problems such as the graphic version of the temperature converter. The dimensions of LiveWorld actors are directly controlled by the four slots **xpos**, **ypos**, **xsiz** and **ysiz**. The rectangle constraint defines two additional slots, **bottom** and **right**, and enforces geometric consistency among the resulting set of six slots. **Rectangle** does not by itself put any constraint on the appearance of an actor, instead it allows other constraints to be placed on any one of a rectangle's sides or dimensions. It should also be noted that **rectangle** constraints can be applied to any actor, or indeed anything with the appropriate slots. For instance, it makes sense to give ovals a rectangle constraint. The constraint actually means "enforce consistency on an object's rectangular bounding box."

The rectangle constraint is defined by the following template:

```
(deftemplate (rectangle ?r)
  :satisfied
  (and (= (ask ?r :right) (+ (ask ?r :xpos) (ask ?r :xsiz)))
        (= (ask ?r :bottom) (+ (ask ?r :ypos) (ask ?r :ysiz)))))
```

This means that an unsatisfied rectangle task will expand into an **and** task which will in turn expand into two equality tasks.

Geometric constraints can also express inter-object relationships. The **within** constraint specifies that one object should lie entirely within another:

```
(deftemplate (within ?a ?b)
  :satisfied (and (>= (ask ?a :xpos) (ask ?b :xpos))
                 (<= (ask ?a :right) (ask ?b :right))
                 (>= (ask ?a :ypos) (ask ?b :ypos))
                 (<= (ask ?a :bottom) (ask ?b :bottom))))
```

A **within** task expands to an **and** subtask with four subtasks of its own. Each of these enforces a numeric relationship. When \geq and \leq tasks are unsatisfied, they are expanded into equality tasks. That is, if the task $(\geq a b)$ is encountered and unsatisfied, the system will attempt to enforce it by trying to equate a and b . This technique obviously is incomplete, in that it only tries a very small subset of the possible ways to make the constraint true. However, it produces reasonable interactive behavior in the case of the **within** constraint. If the user drags an object constrained to be within another, the agents will do nothing as long as the constraint is satisfied. If the object goes outside the boundaries of the surrounding object, the surrounding object will be moved as if bumped and dragged.

Note that the two objects constrained by a **within** task must also have **rectangle** constraints defined so that their **bottom** and **right** slots are defined and maintained properly.

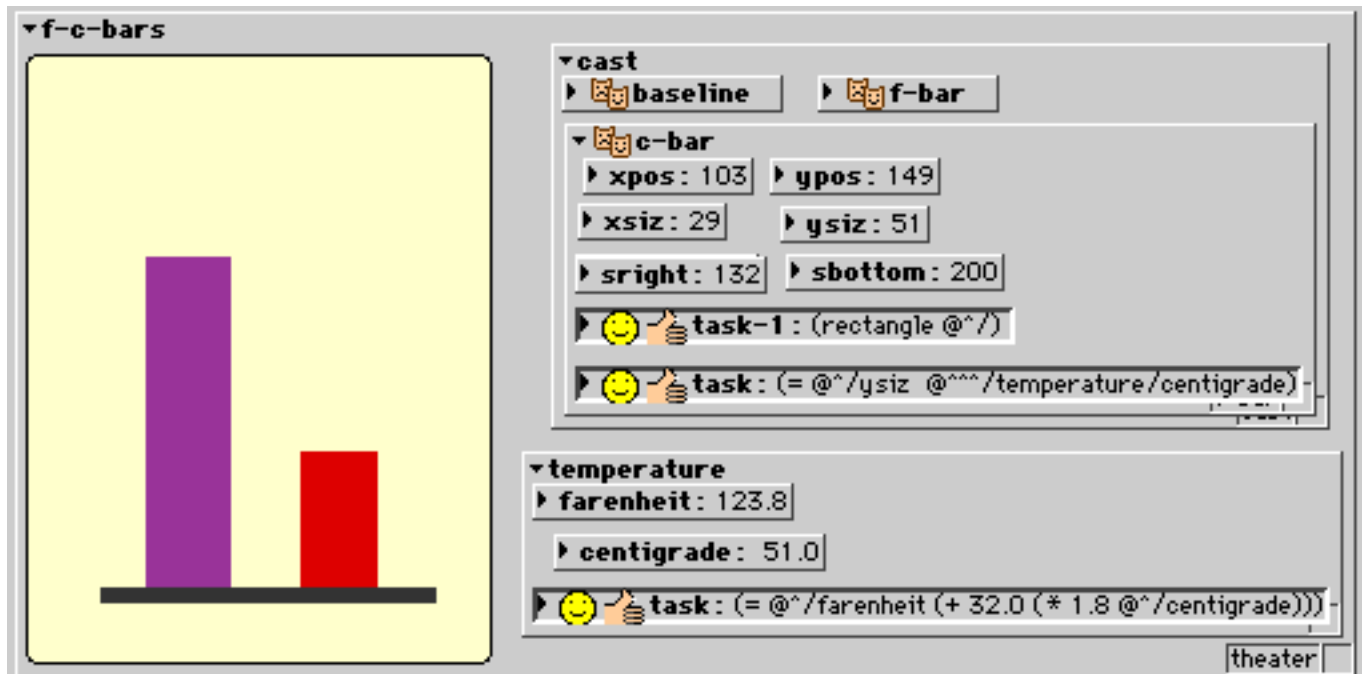


Figure 5.10: A world combining numerical and geometrical tasks. The conversion task (T1 in the text) appears in the task object within temperature. Within the centigrade bar, **task-1** asserts that the bar is a rectangle while **task** constrains its vertical size to be equal to the computed centigrade temperature.

5.2.5.3 Behavior

Tasks and templates from behavioral domains tend to make more use of the **repeat** construct, reflecting the fact that the actions that satisfy goals in this domain generally do not satisfy immediately, as they do in the more abstract domains above. Instead, they achieve their goal only by repeated application. A simple example:

```
(deftemplate (get-to ?creature ?place)
  :satisfied (eq (ask (ask ?creature :overlap-sensor) :other)
?place)
  :action '(repeat
            (script
              (head-towards ?creature ?place)
              (do (ask ?creature :forward 10))))))
```

This template defines a goal called **get-to**, that expects to have actors match its pattern variables. The goal is satisfied when its two actor arguments are in contact, as defined by the creature actor's `overlap-sensor`. This template generates a subagent with a **repeat** task. This means that the innermost task (the script) will be repeated until the **get-to** task is satisfied. The script first attempts to satisfy the **head-towards** goal, and when it is satisfied will proceed to activate the **do** agent, which will move the creature forward. The **script** task ensures that forward motion only happens if the **head-towards** goal is satisfied, that is, only if the creature is facing the target. The fact that the repetition is outside of the script ensures that the **head-towards** goal will remain monitored and if the target object moves, the creature will realign itself before moving forward. By contrast, a goal of the form:

```
(script
  (head-towards ?creature ?place)
  (repeat (do (ask ?creature :forward 10))))
```

would adjust the heading once, and then go forward indefinitely with no further course corrections.

5.2.6 Conflict

There are two types of conflict that can occur during the execution of a DA system. The first, *slot conflict*, occurs when two or more agents attempt to set a slot to different values. The second, *goal conflict*, occurs when a successful action results in another agent's goal becoming unsatisfied. These two types of conflict represent different constraints on agent activity. Slot conflict models the physical impossibility of an object being in two states at once or taking simultaneous actions that are physically incompatible (for instance, an actor cannot move in two different directions at once). Goal conflict, on the other hand, reflects the fact that agents will often interfere with each other's goals, and that either compromises must be reached or some agents will remain unsatisfied.

5.2.6.1 Slot Conflict

Each major cycle might result in many worker agents attempting to take action. This raises the possibility that they might come into conflict. As in Simple Agents, all the actions of worker

agents are ultimately expressed as changes to the values of slots, and so slot conflict is defined as more than one agent trying to set a slot to incompatible values. The DA system uses techniques similar to the two-phase clock technique used in the simpler agent systems. Under this technique, the actions of worker agents are not actually performed when the agent is activated but instead get deferred and turned into proposals for each slot.

At the end of each major cycle, therefore, a conflict resolution procedure is required to determine which slots have conflicting proposals and which agents shall prevail. This involves collecting all the slots that have proposals, all the worker agents that have made proposals, computing sets of agents that are in conflict, and for each set, picking an agent to prevail. This is done by comparing the determination of the agents in the conflicting set. The agent thus selected gets to make its proposals take effect, and is marked as successful, while the losing agents fail.

In some cases, it is important to detect conflicts between agents that do not necessarily get activated during the same major cycle. This is the case for agents that are jointly underneath an **and** agent, for instance.

When an agent's proposal is realized, the slots that are changed record which agent provided the value, and the current clock tick. When the **persistent-slot-values** switch is on, this data is used in *future* cycles. In this case, in addition to any proposals suggested by agents running in the current cycle, the prevailing value and the agent that set it are also compared in the conflict resolution stage. This essentially means that in order to change a slot value, an agent must have a higher determination than the last agent to successfully set that slot's value.

5.2.6.2 Goal Conflict

Goal conflict occurs when the action of one agent causes another agent, previously marked as satisfied, to become unsatisfied. When this happens, the newly-unsatisfied agent is *reset*, that is, its completion state is revoked so that it can run again. In addition, it is marked as the victim of a conflict, which causes no additional processing but causes the graphic representation of the agent state to be an "angry" icon rather than the less descriptive "unsatisfied" icon.

The intent of the goal-conflict handling mechanism is to allow these "angry" goals to have another chance to satisfy themselves. In the most usual case, the goal was originally satisfied without taking any action, so it had not asserted its control over any slots. Now that the conflicting agent has run, when the original agent runs again it is likely to encounter slot-conflicts with the agent that caused the goal-conflict. The slot-conflict mechanism is more specific, and gives the conflicting agents a chance to search for non-conflicting methods of attaining their goals.

5.2.7 Determination

Each agent has a determination value between 0 and 1 that is used to resolve conflicts. The metaphorical interpretation of determination is a measure of how much the agent wants to be

successful, and how strongly it will “press its case” when coming into conflict with other agents. An agent’s determination value can depend on a number of factors:

- the determination and type of its superior agent;
- how many siblings exist, and how many of them have failed;
- the determination scaling factor of the creating template.

The default determination value is 1. Agents that have subordinates pass the determination down to them according to their type. **and**-type agents, that require all of their subordinates to be successful, will give each subordinate the same determination as itself. **or**-type agents (which include **or** and **oneof** agents) will split their determination among their subordinates.

More precisely, an OR-type agent’s determination is split up among those subordinates that have not failed. When a subordinate does fail, the determinations of its remaining siblings are recalculated.

The effect of this determination calculation in constraint-type domains is roughly to implement local propagation of known states. A fully known state, in this interpretation, is a slot that is being held to a value by an agent with a determination of 1. A constraint agent that supervises several different methods will initially give each method a determination less than 1. If a method tries to set a slot that has been locked (that is, set by a highly-determined agent) it will fail. If all methods fail, the system tries to determine which agents have failed permanently (for reasons other than conflict) and which still have a chance for succeeding if their activation is increased. Then each of the latter class is given a chance to run with the full determination of its parent.

Figure 5.11 illustrates a typical sequence of events that make up a propagation. When slot1

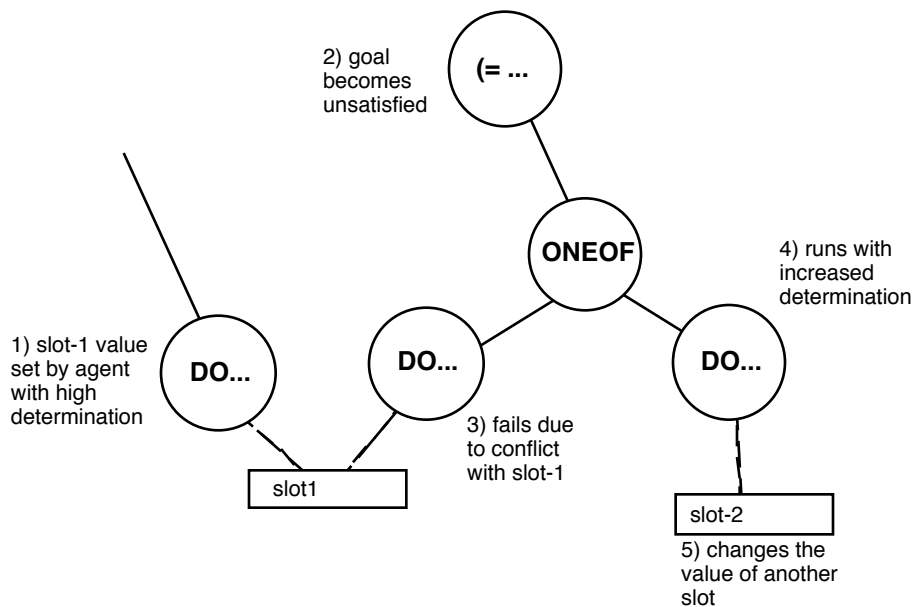


Figure 5.11: How agents implement local propagation of constraints.

is set by an agent with a high determination (step 1), it causes the goal agent to become unsatisfied (step 2). The goal agent activates its subordinates, and while the first one now fails where previously it might have succeeded (step 3), the second one succeeds (step 4) because it alters a different slot, which might result in further propagation. For a detailed example of how this method of propagation works in practice, see section 5.3.5.

5.3 Interface

We have described how agents work but have not yet described how the user specifies and controls them. The most important need for the interface is to allow the user to create and remove top-level tasks. In addition the user needs to be able to exert some control over the agent interpreter, such as cycling manually for debugging or controlling certain parameters that affect the operation of the interpreter.

5.3.1 Top-Level Tasks

Top-level tasks are those specified directly by the user. They are used to create the initial population of agents directly subordinate to the top agent. Top-level tasks can be created in a couple of different ways: they can be explicitly specified by the user by using *task objects*, or they can be automatically generated as a result of user actions such as dragging, which allows the user's intentions to be represented to the agent system as tasks.

Explicit top-level tasks are specified by means of task objects (see Figure 5.10 for some examples). Like other LiveWorld objects, these are created by cloning a prototype. The task is entered as the value of the object. Task objects also incorporate a toggle button, which allows them to be turned on or off, and icons that indicate the task's status. Relative box paths (see 4.5.4.2) may be used in the specification of tasks, which makes it possible to incorporate task objects into other objects which can then be used as prototypes for further objects (that is, the task objects will be cloned along with their container objects, and the new tasks will properly refer to the new object).

5.3.2 Auto-tasks and Locks

In order for DA to be used as an interactive constraint system, the user's actions must somehow be represented within the system. This is done by translating user actions such as dragging, size changing, or setting the value of a slot into goal tasks.

Auto-tasks generally have the form:

`(<== (v <slot>) <value>)`

That is, they use one-way equality to specify that the slot is to hold a certain value. In general auto-tasks persist only briefly, and are removed as soon as the user takes another action. For example, during a drag, each time the user moves the mouse, two tasks are created,

one to hold the x position at a constant value and similarly for the y position. During the next move, these tasks are removed and new ones will take their place.

Lock tasks are similar in form to auto-tasks, in that they specify that a slot or slots are to be held to constant value. Lock tasks, however, are created explicitly by sending a slot a `:lock` command. This command simply creates the appropriate task and marks the slot with an icon.

5.3.3 The Agent Display

Agent activity may be monitored through an agent display that shows the status of every active agent in the system. The graph uses LiveWorld's hierarchical box display to convey the hierarchical structure of the agent tree. This also allows the user to control the level of detail by closing uninteresting boxes. Each agent shows its state by means of anthropomorphic icons (see 5.2.2.2). When active, the agent display is dynamically updated as agents change state.

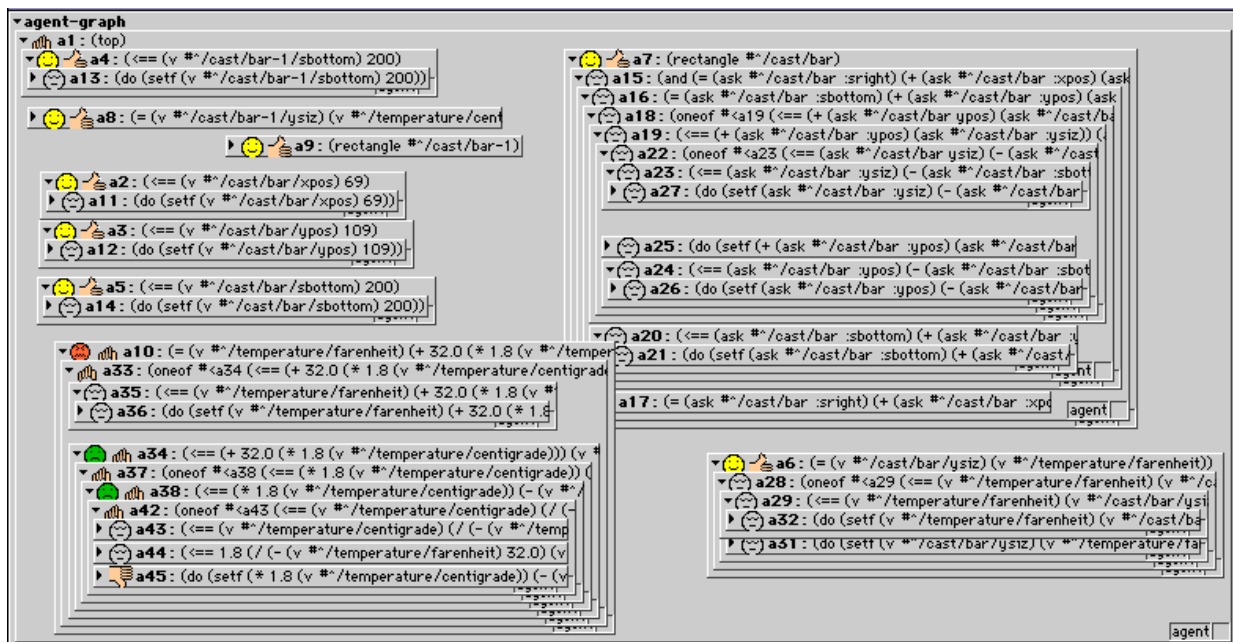


Figure 5.12: The agent display.

5.3.4 Controlling When and How Agents Run

The agent system is designed to run concurrently with user interaction. There are several ways to do this, the best method being dependent upon on the type of problem being modeled. One technique is to have the agent system driven by an anima. This is best for domains in which the modeled system is basically independent of the user, such as an animal behavior simulation. For constraint domains, where objects do not move except in response to the user, a smoother interface is obtained by triggering the agent interpreter explicitly after each user action.

A number of user-settable switches are available to control the details of agent activation. These switches appear in the agent control panel (see Figure 5.13).

- **agents-on?**
When this switch is on, agents will be activated automatically after every user action. That is, after every user operation (such as dragging an object—each individual mouse movement counts as an operation) the system will execute a macrocycle.
- **auto-tasks?**
This switch controls whether or not user actions get translated into auto-tasks.
- **eager-locks?**
This switch, when on, causes lock-type tasks (including auto-tasks) to execute early on in each macrocycle. This makes constraint-solving faster by ensuring that all locked slots have their value asserted early on. When the switch is off, lock agents must first become unsatisfied before they can run, which leads to extra cycles of activity.
- **hide-activity?**
This switch controls when redisplay happens, but does not affect agent activity. When off, redisplay happens after every millicycle, that is, after every agent operation. When on, redisplay only happens after every *macrocycle*, that is, it happens only after the agent system is quiescent with all satisfiable agents satisfied. Turning this switch on can make constraint solving seem “smoother” at the cost of hiding the processing from the user.

The remaining switches in the control panel control how much information is displayed but do not affect agent operation. **show-auto-tasks?**, for instance, control whether or not auto-tasks are turned into visible task objects. These switches exist only for the sake of allowing the user to save on both time and the consumption of screen real estate.

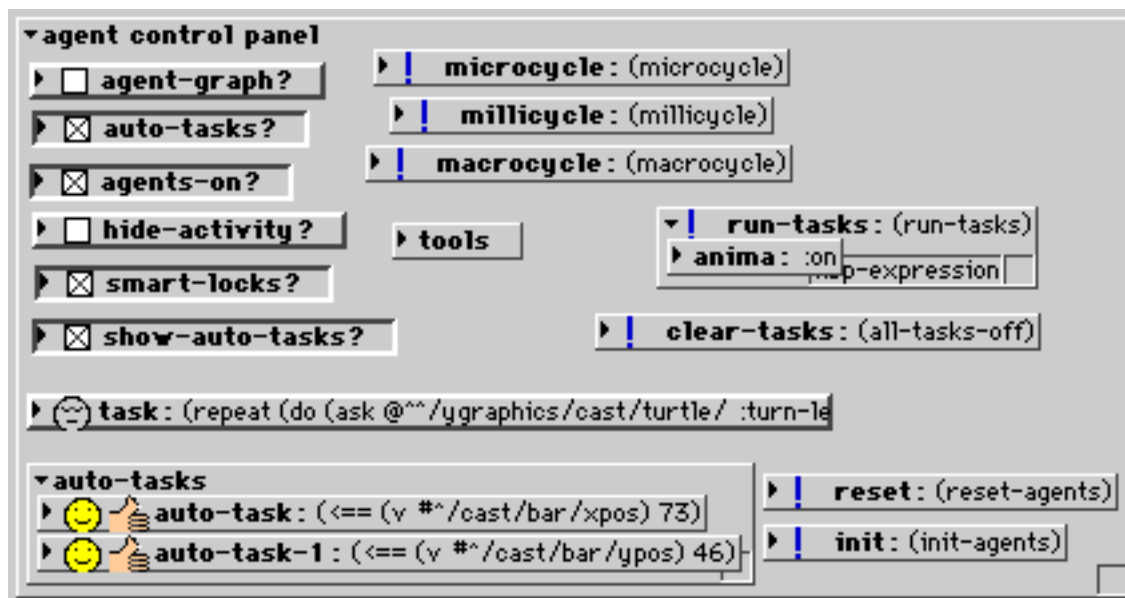


Figure 5.13: The agent control panel.

The control panel also contains methods that allow the user to manually initiate agent cycles, an anima-based agent driver (**run-tasks**), the prototype for task objects, and other agent-related miscellany.

5.3.5 Agent Displays and Storyboards

The operation of a system of agents is a complex affair, with possibly hundreds of agents acting and interacting. It is a challenge to present this activity to the user in a comprehensible manner. Similar problems are present in any programming system, of course, but the concurrency and distributed control model of DA makes following a sequence of action even more difficult than it might be otherwise. The anthropomorphic metaphor underlying agents suggests that narrative forms might be useful for the purpose of conveying the underlying structure of activity (see section 3.4.3). Users of Dynamic Agents can generate *storyboards*, or comic-strip-like graphic representations, as one method of visualizing agent activity.

Storyboards have an advantage over other methods for representing temporal processes (such as animation) in that a storyboard maps time into space, so that a number of different moments of time may be seen together and compared visually. The comic-strip form, defined as “juxtaposed images in deliberate sequence” (McCloud 1993), has a rich vocabulary for expressing both temporal and spatial relations, at a variety of scales. Comics are also a narrative form, meaning that they lend themselves to expressing the “vicissitudes of intention” in Bruner’s phrase. In plainer terms, stories deal with characters with goals, their attempts to realize those goals, and their success or failure—a format which nicely supports the activity of agent systems.

The storyboards automatically generated by DA only scratch the surface of the representational and narrative language available. A storyboard is composed of a sequence of *panels*, each of which represents some segment of time in the evolution of the agent system. Panels can contain a variety of relevant types of information, such as graphic snapshots of the changing world that the agents are manipulating, textual description of the action (modeled after comic-strip captions), or other representations of activity.

In general, there is too much going on in an agent system to put every agent and every action in a storyboard. Thus the problem of storyboard generation requires *selectivity* in terms of the information displayed, as well as the ability to translate events into suitable graphic representations. The system must select which moments to translate into panels, and what details of those moments are significant enough to take up space in the storyboard.

The strategy used to generate DA storyboards is to emphasize change and the agents responsible for it. Each panel of a storyboard represents a moment in which the world changes, and the panel includes both the results of that change (in the form of a snapshot of the relevant parts of the world) and information about the agents responsible for the changes. There are too many agents involved in a typical step to display them all, so further selectivity is called for. The generator chooses agents from the top and bottom of the agent hierarchy that reflect the why and how behind the actions.

This selectivity is in contrast with the more direct, lower-level representations used in the full agent display (see 5.3.3). These take different approaches to illustrating the same thing, namely agent activity. The agent graph display is intended to be a direct, transparent, and literal picture of the agents, while the storyboard is intended to tell a story *about* agent activity. Thus the displays are quite different, albeit with certain common features.

Figure 5.14 is an example of a fairly simple storyboard, illustrating a sequence of action from the graphic temperature conversion problem (see section 5.2.5.1). Each panel of a storyboard represents a major cycle, a reasonable choice given that a major cycle corresponds to a unit of world time. However, a storyboard panel time does not include a panel for every major cycle of the agent interpreter. Only major cycles that actually result in change to the world are included, or in other words, major cycles that include no successful executions of worker agents are excluded. This is to conserve space, to keep the narrative interesting (by omitting panels where nothing visible happens) and to limit the display to the most salient information. For example, the sample storyboard shown has 6 panels, but would have 14 if all major cycles were included. The omitted major cycles include agent activity, but in these cycles all agents fail and so take no action in the world.

This policy has interesting narrative consequences. As it stands, each panel of the storyboard represents a moment of change in the world. If the excluded steps were included, the story being told would be more “internal”, reflecting not just the actions of the agents, but their internal starts, pauses, proposals, and failures. This level of detail might be of value under certain circumstances, but for the present purposes, the presentations of such internal detail will be minimized (as implemented, the storyboard generator has user-settable switches that control the level of detail presented).

Each panel includes a *snapshot* of the changed parts of the world, which may include both changes to graphic actors or changes to the values of other boxes. Since the picture is not enough to convey the reasons and sources of change, panels must also include some information about the agents responsible for the action in the panel. Listing all agents involved in a single cycle would be too much detail. The system selects the agents that best convey the *how* and the *why* behind the action of the panel. In practice, this means including the worker agents that actually performed the action (the *how*) and the top-level agents that the workers are working for, which convey the purpose (the *why*) of the action. The effect is to include both the ultimate goal or purpose of the action and the concrete physical events that were taken to realize it, while omitting the intermediate stages of computation.

Also included are any top-level agents that have experienced a change in their state as a result of the action of the major cycle, even if they themselves have not taken action. In practice this means agents that were satisfied but have become unsatisfied due to the

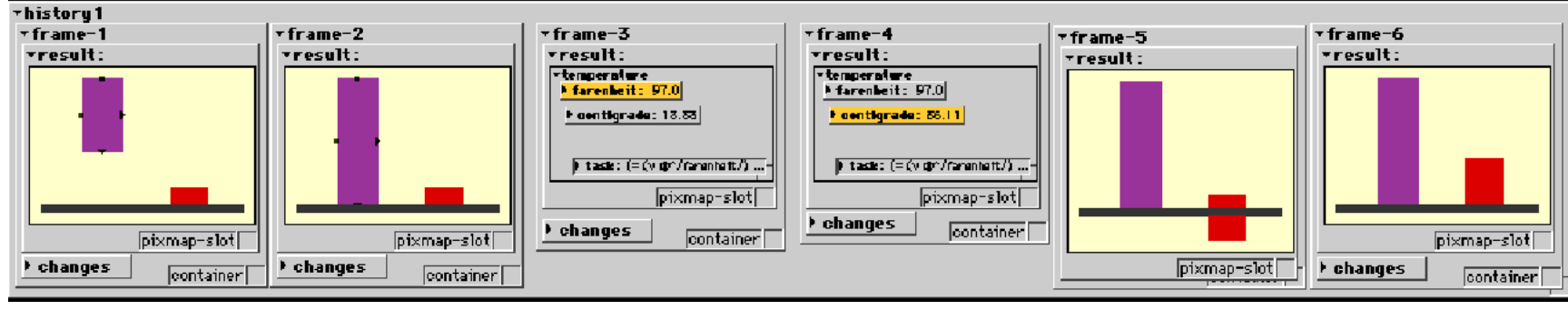


Figure 5.14: A storyboard.

action of another agent (that is, there has been a goal conflict; see section 5.2.6.2). The narrative rationale for including these is to illustrate the further effects of the panel's action, beyond the immediate physical effects. Generally such agents will become the active agents of the next panel, so this is also a device for linking the panels of the storyboard together.

Panels illustrate changes in agent state by using a variant of the usual agent icon notation. Each agent that is shown in a panel is shown with a *pair* of icon sets, separated with an arrow, illustrating the state of the agent before and after the action of the panel. Typically this information shows that some agents went from being satisfied to unsatisfied, or the reverse.

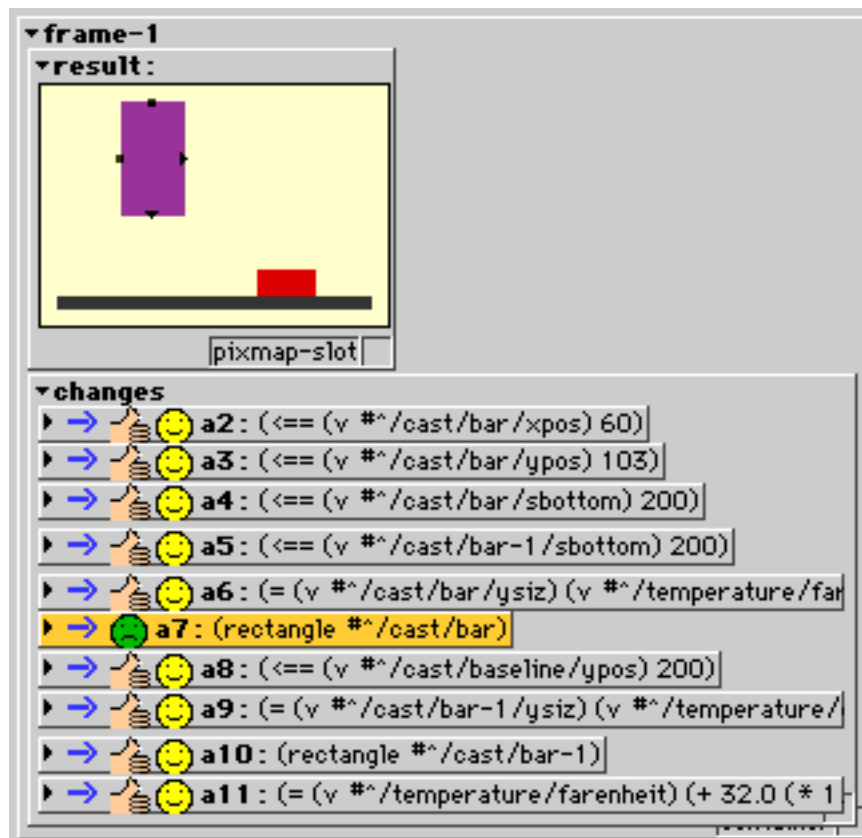


Figure 5.15: The first panel of the storyboard.

There are six panels in this example storyboard, which is a recording of activity in the constraint system shown in Figure 5.10. The action begins just after the user has made a change (by moving the position of the first bar) but before the agents have had a chance to respond to the change. This state is shown in **panel-1**. The “changes” shown reflect the first activation of the various top-level task agents involved in this system. All the agents except **a7** have managed to satisfy themselves. **a7**'s task is to keep the rectangle slots of **bar** consistent. Since **bar/xpos** has changed, **a7** is no longer consistent.

Note that **a4** is satisfied and has succeeded in panel 1, indicating not only that the value of **bar/bottom** is set at 200 (at the level of the baseline) but also that, since **a4** has run, it will be

attempting to hold the slot at that value throughout the computation. This has consequences for the next panel.

In the next panel, **a7** is activated and succeeds in becoming satisfied. By opening up the box representing **a7**, we can see that it was the worker agent **a29** that brought this about by altering the **ysiz** slot of **bar**.

This omits much of the internal agent activity involved in accomplishing the end state, such as an attempt to satisfy **a7** by setting **bar/bottom** instead of **bar/ysiz**. This attempt failed because it came into conflict with the previously successful (and more determined) agent **a4**.

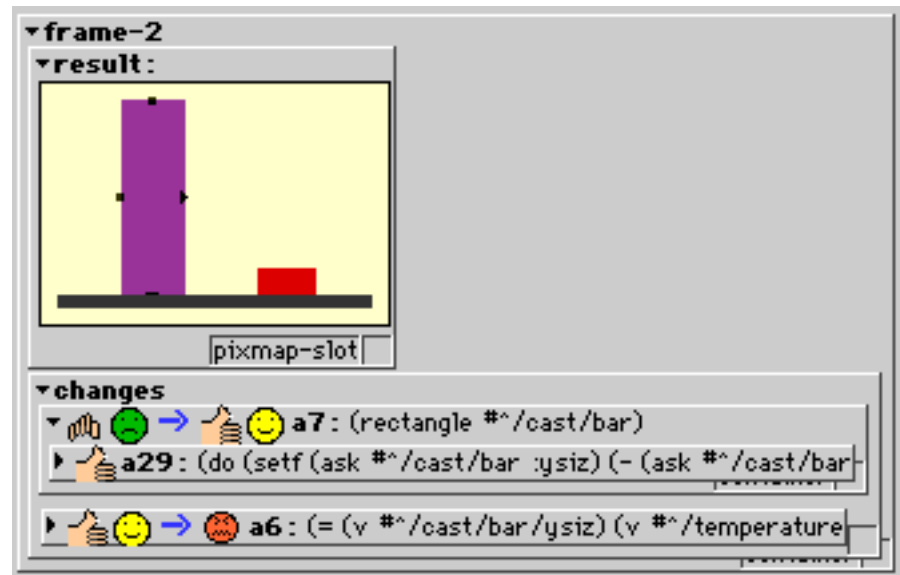


Figure 5.16: Panel 2.

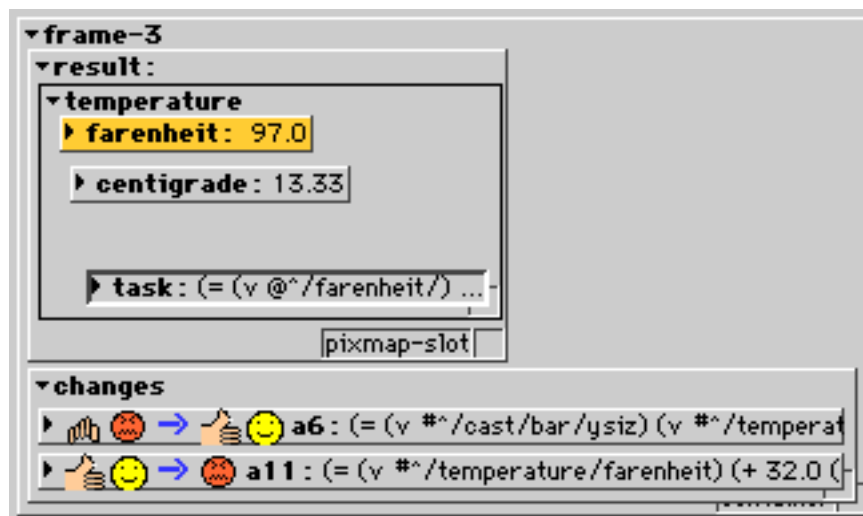


Figure 5.17: Panel 3.

a29's action has satisfied **a7** but has also caused **a6**, who relates the vertical size of **bar** to the value of the **Fahrenheit** slot, to become unsatisfied.

In **panel-3** a similar pattern is repeated. **a6** satisfied itself by changing the value of **Fahrenheit**, causing **a11**, which relates the values of **Fahrenheit** and **centigrade**, to become unsatisfied.

Since **Fahrenheit** is not a graphic slot, the system cannot illustrate the action of this panel with a snapshot of the actors involved. Instead, it selects the appropriate boxes and their surrounding context.

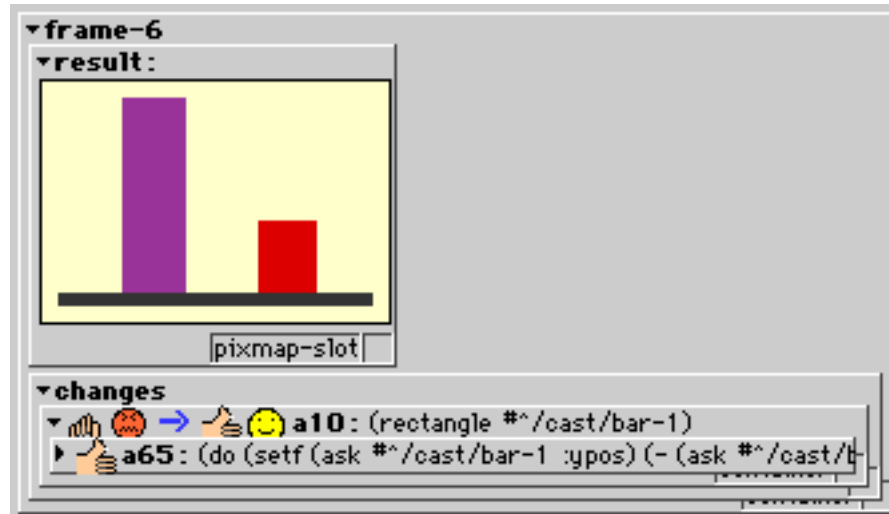


Figure 5.18: The final panel of the storyboard.

This pattern of propagating goal conflicts and resolutions continues in successive panels. In **panel-4**, **a11** sets **centigrade** to its correct value. In **panel-5**, the wave of change propagates back into the graphic part of the world, causing a change in **bar-1**'s vertical size. Finally, in **panel-6**, **bar-1**'s vertical position is adjusted without violating any further constraints, so the agents finally become quiescent, and the story is complete.

5.4 Additional Examples of Agent Systems

Some additional examples of agent based systems are presented here. The first two are a game and a simulation that use the simpler forms of agents, while the rest are built using Dynamic Agents.

5.4.1 A Video Game

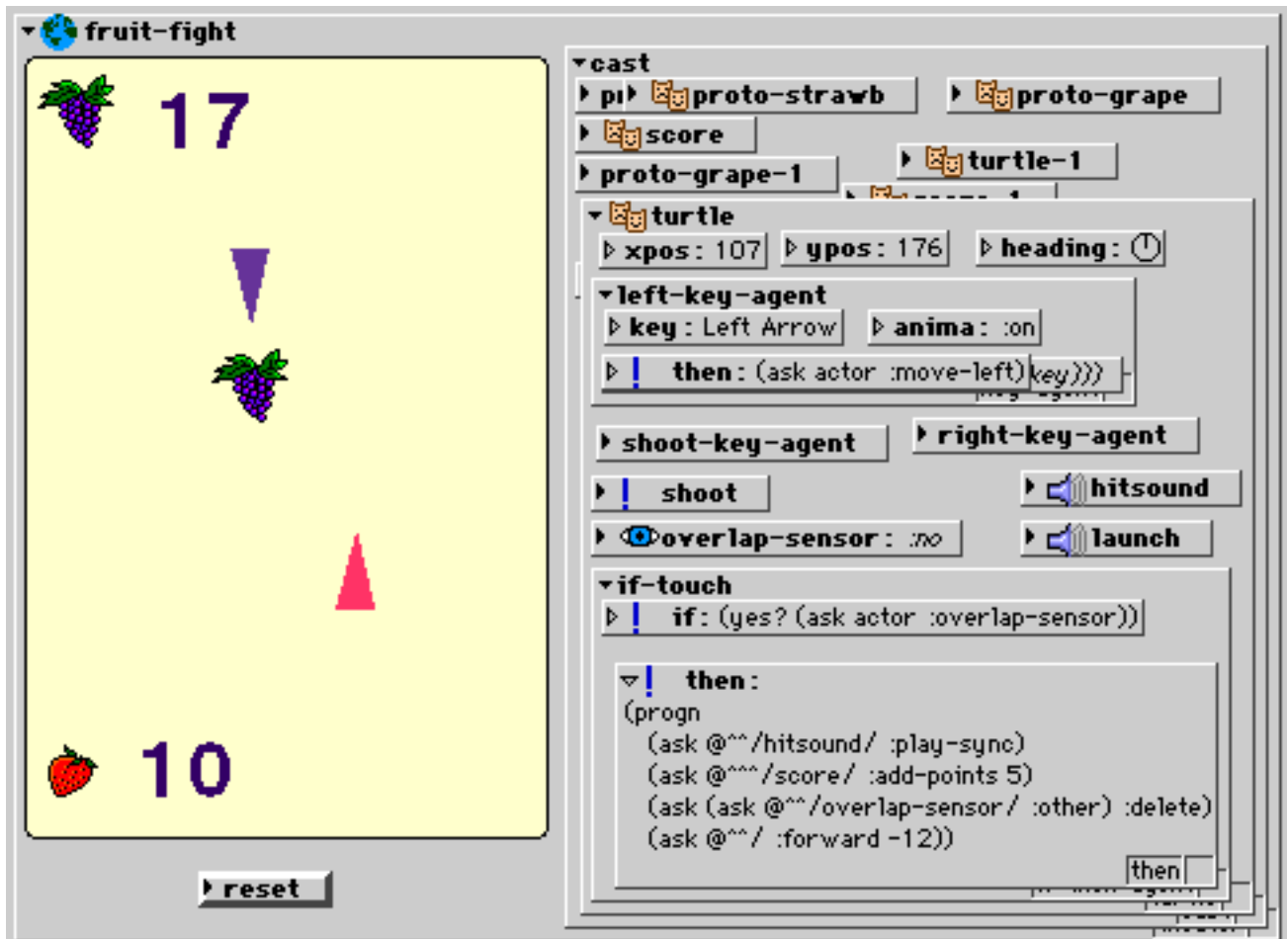


Figure 5.19: A video game.

This simple video game was constructed by a novice user (with assistance) from sets of pre-built components. The components in this case come from a library designed for videogames which includes behaviors such as motion, shooting and collision detection, agents that implement keyboard commands, and the score actors. In this game, two players attempt to drive each other backwards by shooting fruit at each other. Each player controls a turtle from the keyboard. Here one turtle is open to reveal its agents, which are mostly if-then agents, a variant of goal agents (see section 5.1.2).

The **left-key-agent** is built out of a general purpose key agent, so the constructor needs to fill in only the key and the action, which is performed repeatedly whenever the key is held down. In this case the left-arrow key results in the turtle moving left. Another agent, **if-touch**, detects when a player is hit by an opponent and moves backwards.

5.4.2 A Physical Simulation

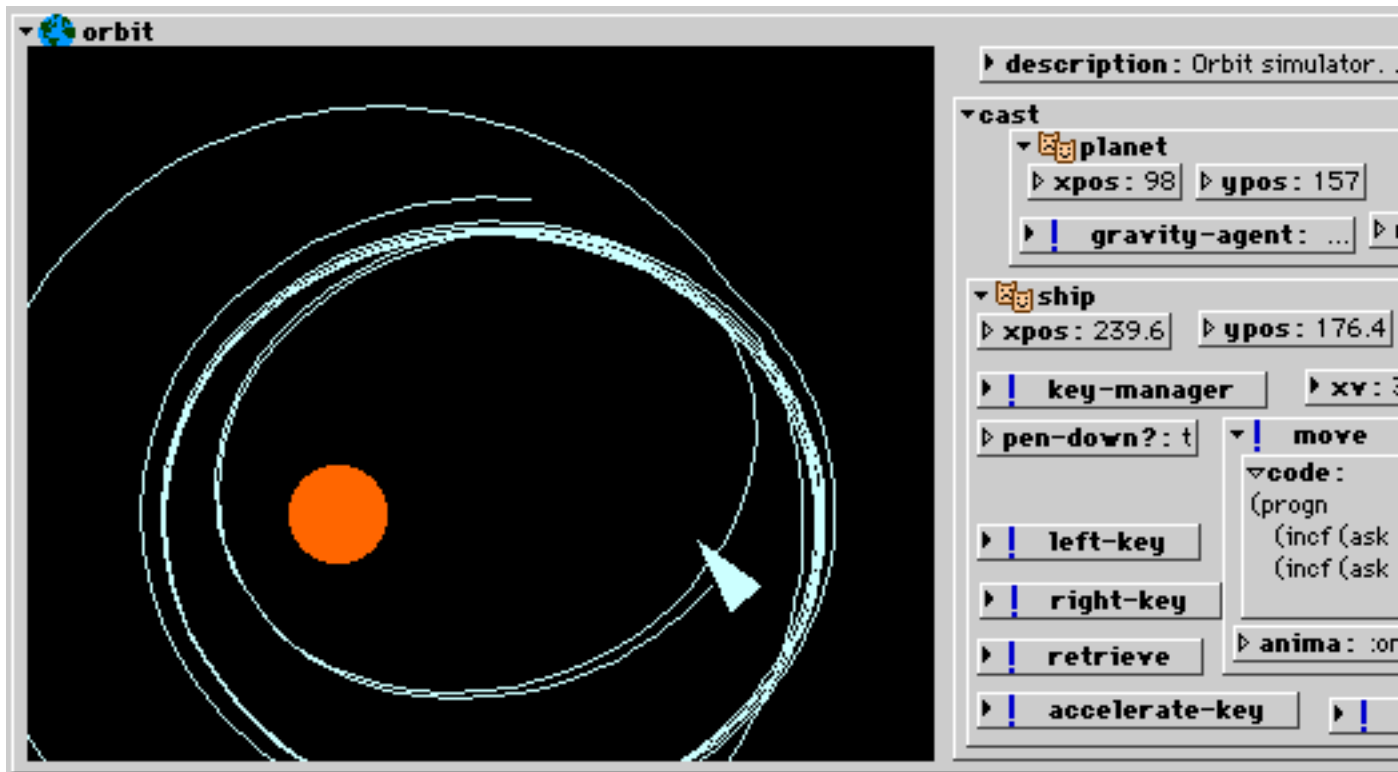


Figure 5.20: An orbital simulator.

Physical forces can be modeled by simple agents. This example shows a simulation of a spaceship orbiting a planet. It was assembled from a kit of parts that provide basic physical simulation capabilities, such as velocity and acceleration. A simple agent, **move**, provides the ship with motion by constantly incrementing its position by two velocity variables. The planet exerts its force on the ship through another simple agent, **gravity**. Because the gravity agent is part of the planet, cloning the planet creates another gravity source which will have an immediate effect on the ship. The user can control the ship's bearing and acceleration (with sound effects) from the keyboard, which is monitored by other agents (i.e. **left-key**) as in the video game example.

This example illustrates how simple agents can be used to simulate physical systems, while giving the user direct tangible access to the variables (slots) and forces (agents) involved. Simulations like this admit to a variety of different modes of use: as a game in which the object is to put the spaceship in a stable orbit around the planet without colliding with it or shooting off the screen; as a vehicle for experimentation (say, by observing the effects of different masses, gravitational constants, or laws of gravity); or as both an example and kit of parts for building further simulations.

5.4.3 A More Complex Graphic Constraint Problem

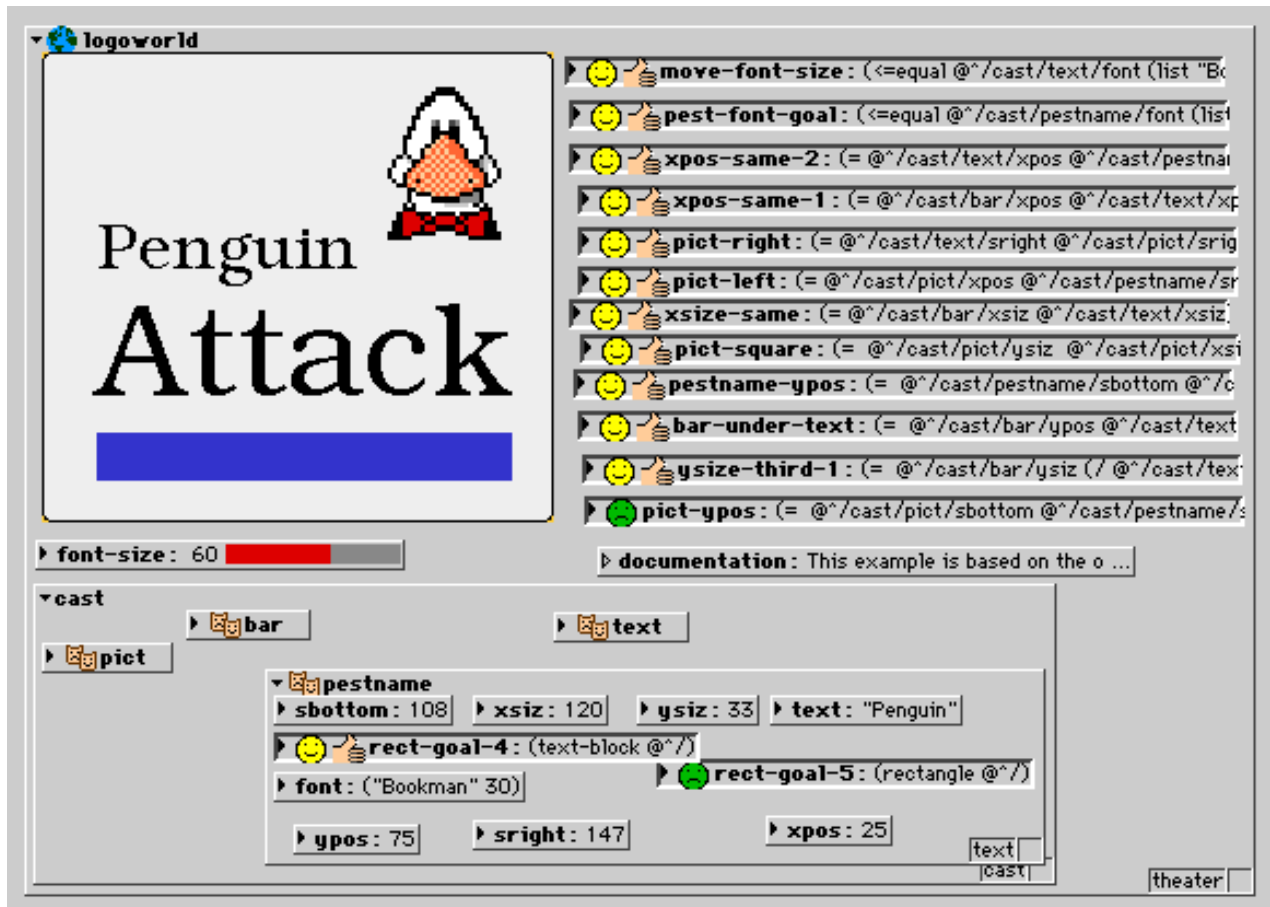


Figure 5.21: A complex constraint example.

This example shows a more complex constraint problem, involving the kinds of tasks found in realistic graphic design problems. The scenario is based on one presented in (MacNeil 1989), which involves designing a series of labels for a line of pesticides. Here we see an example of a label for repelling penguins. This system involves about 22 top-level tasks and typically generates around 100 agents. The top tasks specify relations among the position and size of the four graphic elements in the design.

For example, these user tasks:

```

▶ 🤗👍 pict-square: (= @^/cast/pict/ysiz @^/cast/pict/xsiz)
▶ 🤗👍 pict-left: (= @^/cast/pict/xpos @^/cast/pestname/sright)

```

constrain the graphic (**pict**) to be square, and to have its left side abutting the world "Penguin" (**pestname**). Other tasks include **rectangle** constraints for all the objects, and **text-block** constraints that ensure that the text actors are exactly large enough to contain their text

(these ensure that if the user should alter the text, the objects will adjust themselves accordingly). Still others specify the left and right edge alignments, hold the bar's vertical height to be 1/3 the height of the main text, and ensure that the two words maintain the relative scale of their font size.

This example illustrates a well-known problem with graphic constraint systems. A typical constrained diagram involves far more individual constraints than can be easily specified by hand. Dynamic Agents ameliorates this problem a bit by allowing abstract constraints, such as **within**, so that fewer individual constraints have to be specified. Still, the task can be daunting. One possible solution is to have agents that can create constraint agents by automatically inferring them from user examples, as in (Kurlander and Feiner 1991).

5.4.4 A Creature in Conflict

This is a fairly simple behavioral system, which illustrates arbitration between competing behaviors. **Task-1** specifies that the turtle should attempt to get to the **goal**, while **task** specifies that the turtle is to avoid overlaps with objects. These two goals are contradictory (because getting to the goal requires overlapping with it), but of even greater concern is the barrier **obstacle** which lies in between the turtle and its goal.

This example indicates the use of determination in animal behavior worlds. When the two tasks specify conflicting actions, the winner is the agent with the higher determination. In this case, determination can be controlled by the user through annotations to the user task objects. If the **get-to** task has higher determination, the turtle will bravely penetrate the unpleasant obstacle on its way to the goal, while if the **avoid-overlaps** task is stronger, as in the figure, it will end up going around it. This illustrates how determination may be set by the user as annotations to tasks.

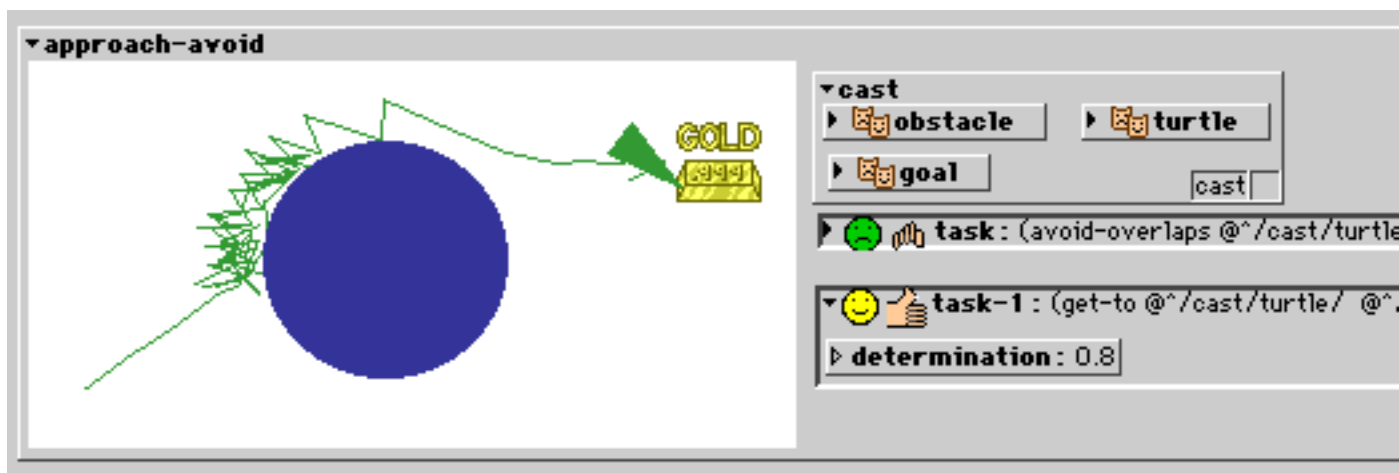


Figure 5.22: A conflicted creature.

The **get-to** task and template are described in section 5.2.5.3. **Avoid-overlaps** is only satisfied when there is no overlap between its **?creature** argument and any other actor. Its

action is a script that backs off from the overlap, then turns randomly and moves forward. The effect of this is to move in a rather groping fashion around the edge of the obstacle.

```
(deftemplate (avoid-overlaps ?creature)
  :satisfied (no? (ask ?creature :overlap-sensor))
  :action `(repeat
    (script
      (do (ask ?creature :forward -15))
      (do (ask ?creature :turn-right (arand 0 180)))
      (do (ask ?creature :forward 15))))))
```

5.4.5 An Ant

This is a slightly more complicated behavioral simulation, based on the realistic ant simulation of Agar (Travers 1988). It illustrates some other methods for organizing multiple goals. In this case, the goals are controlled by special tasks that sequence them. **script** and **try** tasks specify a series of subtasks to be performed in order. This is a different style than in the previous example, where both agents were active concurrently and conflict was resolved at the slot level using determination.

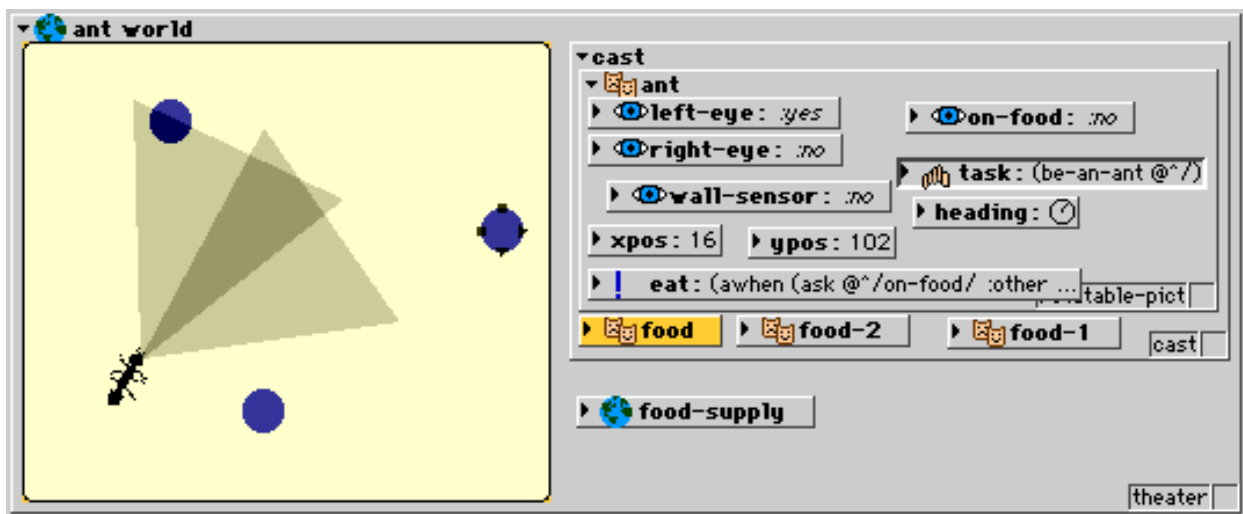


Figure 5.23: An ant..

The top-level **be-an-ant** task arranges three subgoals in a strict priority. The latter ones will not be processed until the earlier ones are satisfied. The use of **and-script** rather than **script** means that if the early tasks in the sequence become unsatisfied (for instance, if **find-food** moves and thereby makes **avoid-walls** become unsatisfied) then control will revert to the earlier goal (see section 5.2.4.1). A plain **script** goal could cause the ant to get stuck in its **find-food** subtask and wander off the screen.

```
(deftemplate (be-an-ant ?a)
  :action '(repeat
    (and-script
```

```
(eat ?a)
(avoid-walls ?a)
(find-food ?a)))
```

This somewhat oddly-structured **eat** agent is satisfied when the ant is not on top of any food, and will eat any such food until it is gone. The actual eating is accomplished by a method named **eat**.

```
(deftemplate (eat ?a)
  :satisfied (no? (ask ?a :on-food))
  :action '(do (ask ?a :eat)))
```

An **avoid-walls** agent will be satisfied when the ant is not touching any walls, and backs away from walls when necessary.

```
(deftemplate (avoid-walls ?a)
  :satisfied (no? (ask ?a :wall-sensor))
  :action '(do (ask ?a :forward -10)
              (ask ?a :turn-right (arand 180 30))))
```

The following **find-food** agent is satisfied when the ant *is* touching food¹⁹. Note the duality between **script** and **try**. **Try** is useful for precondition-based subtasks that are likely to fail, **script** (and **and-script**) works well with goal-based tasks. Note that the goal-tasks like **avoid-walls** could easily be expressed in precondition/action form rather than as a goal. Unfortunately it is hard to mix these two types of task underneath one manager.

```
(deftemplate (find-food ?a)
  :satisfied (yes? (ask ?a :on-food))
  :action '(repeat
            (try
             (find-food-forward ?a)
             (find-food-left ?a)
             (find-food-right ?a)
             (find-food-random ?a))))

(deftemplate (find-food-forward ?a)
  :precondition '(and (yes? (ask ?a :right-eye))
                     (yes? (ask ?a :left-eye)))
  :action '(do (ask ?a :forward 10)))

(deftemplate (find-food-left ?a)
  :precondition '(yes? (ask ?a :left-eye))
  :action '(do (ask ?a :turn-left 10)))

(deftemplate (find-food-right ?a)
  :precondition '(yes? (ask ?a :right-eye))
  :action '(do (ask ?a :turn-right 10)))

(deftemplate (find-food-random ?a)
```

¹⁹Note that this means that the **and-script** agent that manages the **be-an-ant** goal can never itself be satisfied, because it contains two contradictory terms. This is permissible, and it means that the **repeat** around the **and-script** is not strictly necessary.

```

:action '(do
  (ask ?a :turn-left (arand 0 90))
  (ask ?a :forward 20))

```

5.4.6 A Recursive Function

While the DA system deliberately de-emphasizes computation in the literal sense, it is still capable of performing the recursive computations that are the specialty of procedural and functional languages. Because DA tasks cannot return values, making this work requires a bit of trickery, which is to use LiveWorld's hierarchy to store intermediate values in temporary boxes.



Figure 5.24: Computing factorial with agents.

```

(deftemplate (factorial ?x ?x-fact)
  :name fact-end-test
  :precondition '(= ?x 1)
  :action '(<== ?x-fact 1))

(deftemplate (factorial ?x ?x-fact)
  :name fact-recurse
  :precondition '(> ?x 1)
  :action (let ((temp (getb ?x-fact :iresult)))
    `(all (factorial (- ?x 1) (v ,temp))
      (<== ?x-fact (* ?x (v ,temp))))))

```

Two templates define agents that implement the end-test and recursion steps of the standard recursive factorial definition. The recursion step involves creating an intermediate box, as an annotation of the final result, and posting two subtasks: one to compute the intermediate result that will be stored in the new box; and another to relate this intermediate result to the final result. The user task specifies that anytime the value of **n** changes, **fact-n** will be set to the factorial of the new value, with intermediate result boxes being created as necessary. Note that the tasks here involve one-way equality (<==) rather than the equality relations found in the constraint examples. This means that changing **fact-n** will just result in a failed and unsatisfied task, rather than an attempt to compute the (generally undefined) inverse of factorial.

5.4.7 A Scripted Animation

When young users first encounter a system that allows them to create moving objects, their first impulse is often to create a story based on simple movements: “Sam the elephant leaves his house, goes to the park, then goes to school, then goes back home”. Unfortunately systems that are purely reactive make this simple task difficult. In DA, the **script** construct allows this task to be done naturally.

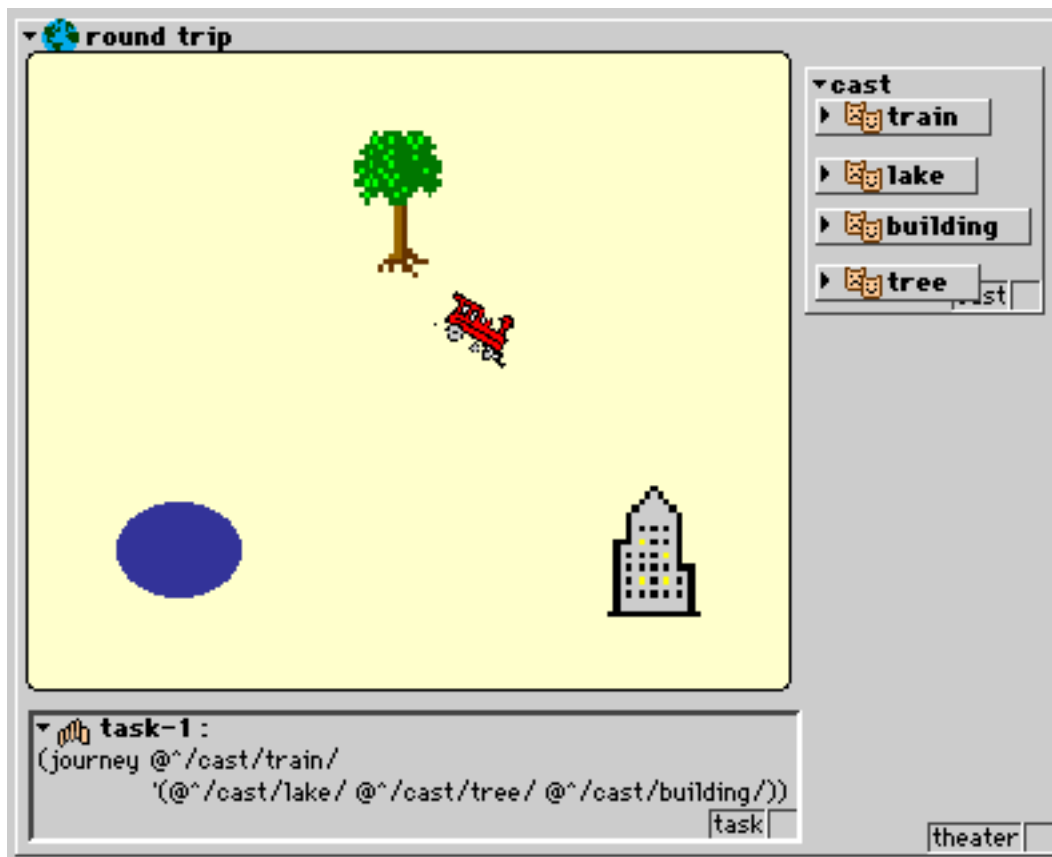


Figure 5.25: A scripted journey.

```
(deftemplate (journey ?creature ?itinerary)  
  :action `(script ,(mapcar #'(lambda (place)  
    `(get-to ?creature ,place))  
    ?itinerary)))
```

This template for **journey** tasks uses a Lisp macro to allow the task to take a variable-length itinerary, which is converted into a **script** task with the appropriate number of arguments. Because it uses **get-to** tasks (see above), it works even when the destinations are in motion—the train will simply re-orient as necessary. The effect is that the train will seek its goal, even if it is moving, but as soon as it touches the goal, the train will go on to seek the next goal in its script.

5.4.8 BUILDER in the Blocks World

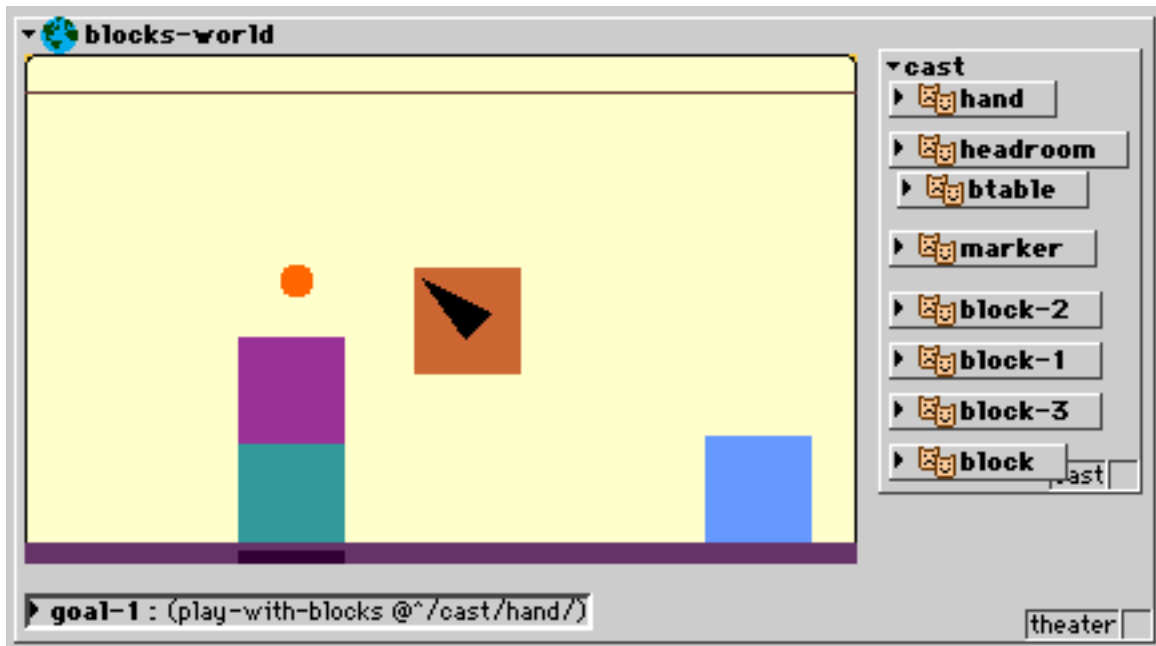


Figure 5.26: Stacking blocks.

The sole user task in this world is **play-with-blocks**, which has two subordinates, **build** and **wreck**. The **all** construct specifies that they should both be activated at once.

```
(deftemplate (play-with-blocks ?a)
  :action `(all (build ?a) (wreck ?a)))
```

The **add** task is the basic block-building script, which serially finds a block, gets it, finds the top of the tower, and puts the block there. **Build** simply repeats the **add** operation until a block intersects the headroom object.

```
(deftemplate (build ?a)
  :satisfied (yes? (ask #*/cast/headroom :block-sensor))
  :action '(repeat (add ?a)))

(deftemplate (add ?a)
  :action `(script (find-free-block ?a)
                  (get-obj ?a)
                  (find-tower-top ?a)
                  (put-obj ?a)))
```

The **find-free-block** and **find-tower-top** agents call up behaviors written in the lower-level language. These functions use the marker as a kind of sensor, based on the idea of a visual routine (Ullman 1984). For instance, **find-free-block** moves the marker to the left hand side of the world, just above the table, and moves it to the right until it encounters a block. If so, it moves up until it detects either empty space (in which case it returns the value of the block it was just on) or another block (in which case it continues its leftward search). These operations

could have been written in DA, but have been implemented as methods in the underlying language for speed.

```
(deftemplate (find-free-block ?a)
  :action '(do (ask (ask ?a :marker) :findfree)))

(deftemplate (find-tower-top ?a)
  :action '(do (ask (ask ?a :marker) :findtop)))
```

Get-obj and **put-obj** make use of the marker object to find their argument (the object to get or the place to put it). So **get-obj** first makes the hand **get-to** to the marker, then activates a **grasp** task, which calls an underlying method that effectively grabs an underlying block object (grabbing is in fact implemented by a separate simple agent, which continually moves the grabbed block to the location of the hand as it moves).

```
(deftemplate (get-obj ?a)
  :action `(script (get-to ?a (ask ?a :marker))
                  (grasp ?a)))

(deftemplate (put-obj ?a)
  :action `(script (get-to ?a (ask ?a :marker))
                  (release ?a)))

(deftemplate (grasp ?a)
  :satisfied (ask ?a :grasped-object)
  :action '(do (ask ?a :grasp)))

(deftemplate (release ?a)
  :satisfied (null (ask ?a :grasped-object))
  :action '(do (ask ?a :release)))
```

Wreck works by finding the tower top, moving the hand to it, and systematically scattering the blocks. Since **wreck** is activated concurrently with **build**, we need to give it a lower determination so that **build** has a chance to work. Once **build** is satisfied (which happens when the tower is tall enough), then it will no longer attempt to run and **wreck** will have a chance. In fact, since **wreck** is only inhibited when there is a slot that both **build** and **wreck** are attempting to change, **wreck** might occasionally seize control of the hand while **build** is not moving it (for instance, when it is moving the marker). This implementation of unconscious destructive urges is considered a feature. If stricter control is necessary, **script** can be used in **play-with-blocks** to ensure that **wreck** is dormant until **build** is done.

```
(deftemplate (wreck ?a)
  :determination .9
  :action ...)
```

5.5 Discussion

The design of the dynamic agent system and its predecessors attempts to weave together a number of different ideas and influences:

- the deliberate use of anthropomorphic metaphors;

- the power and generality of procedural programming languages;
- the responsiveness of behavioral control networks;
- the declarative power of constraint systems.

The developmental history of the agent systems began with the idea of loosely anthropomorphic agents. I built several behavioral control systems using them, and then attempted to integrate the agent idea with a more general-purpose language. This involved thinking about agents driven by goals rather than by situations, and led to the realization that agents could be a good basis for constraint solving systems.

5.5.1 DA as a Procedural Language

DA provides equivalents to most of the basic control and modularity constructs found in procedural languages: sequencing, conditionals, iteration, abstraction, and recursion. These appear in different forms than they do in procedural languages, however. Iteration, for example, is to some extent implicit in the concept of an agent. Therefore rather than having an explicit iteration construct, DA provides a construct, **repeat**, which does not generate a loop in the way imperative loop constructs do but instead influences the behavior of the iterative agent interpreter.

It is worth clarifying the conceptual relationship between DA agents and procedures. Agents are clearly similar to procedures (or more precisely, procedure invocations): both perform tasks, both are capable of being anthropomorphized; agents can invoke other agents and thus create a hierarchy similar to a tree of procedure invocations. However, there are important differences. Agents can be running concurrently, whereas only one thread of a procedure tree can be active at any one time. Agents are persistent, and are driven iteratively and thus capable of autonomous activity. An agent's supervisor can activate or deactivate it, but while active it is in charge of its own operation. When an agent has a goal, this means that it is in effect constantly monitoring the goal and taking action to achieve it when necessary.

One notable feature of procedural languages that is missing from DA is the notion of the value of an expression. In DA, tasks do not have values, and all computation is strictly by means of side-effects. The choice to omit values was made to emphasize the notion of an agent as something that performs an *action*, rather than as something that computes a value. In part this was a reaction (perhaps an over-reaction) to the emphasis on value computation found in functional or mostly-functional languages, as well as some attempts to model agents in a functional style (see section 3.4.2.4) which did not, to my mind, capture agency adequately. Another reason for de-emphasizing the return of values was the accessibility of Lisp substrate for doing actual computation when necessary.

Programs that normally work by returning values can be transformed into equivalent programs that work via side-effects, by creating temporary boxes to hold the intermediate values (see the factorial example above). It might be possible to integrate value-returning back into the language. It did not fit into earlier agent systems, in which agents were totally

independent, but since DA preserves a calling hierarchy the idea that a called agent could return a value would be more natural.

5.5.2 DA as a Behavioral Control System

As a behavioral control system, DA offers capabilities roughly equivalent to those of its predecessor Agar, Brook's subsumption architecture (Brooks 1986), Pengi (Agre and Chapman 1987) or teleo-reactive systems (Nilsson 1994). Agents can implement a set of responses to conditions in the world that implement adaptive behavior. Unlike these systems (except for Agar), DA uses a hierarchy of agents for control. This makes writing agents simpler since sets of behaviors can be treated as a group (for instance, see how **get-to** is used as a "prepackaged" behavior in several of the examples above), and it increases the salability of the system. Another advantage of DA in comparison to other behavioral control systems is built in structures for implementing sequences.

DA's method of handling inter-agent conflict is somewhat different than these other systems, most of which require hardwired conflict links or other methods of arbitration, if they treat conflict at all. DA detects conflicts at the last possible moment, that is, only if agents actually attempt to simultaneously take incompatible actions (see section 5.2.6). When a conflict is detected, arbitration is performed based on the determination of the agents. The determination of agents is usually fixed, although since it is specified by a regular slot it is possible to alter its value by other agents (say, a hunger-simulating agent that would slowly increase the determination of a food-finding agent).

More sophisticated methods of handling determination are possible within the DA framework. One possibility would be to use a system similar to Maes' Action Selection algorithm (AS), which spreads activation energy (more or less equivalent to determination) between behavior modules based both on current goals and the state of the world (Maes 1990). This algorithm provides a flexible way for modules to string themselves into sequences of action in an emergent fashion. There are some differences in style and goals between this work and DA that make this complicated.

First of all, AS is explicitly non-hierarchical. There are no managers or "bureaucratic modules"; instead all modules exist in an unlayered network, connected by successor, predecessor, and conflict links. To generate these links, AS modules must specify in detail both their preconditions and the expected results of their actions, expressed as lists of conditions. DA is somewhat more flexible, allowing agents to be controlled by either preconditions or goals (expected results). DA does no backwards chaining through the precondition, relying instead on supervisory agents to order goals appropriately. Only one AS module can run at any one time, so all are essentially in competition with one another to run, which contrasts with DA's emphasis on concurrency.

Whereas AS deliberately avoids pre-specified sequences of behavior, preferring them to emerge from the interaction of the available behavior modules, the world, and the goals of the creature, being able to specify sequences explicitly is a design goal of DA. This reflects a deeper difference in the purpose of the two schemes. AS is deliberately done in an emergent or bottom-

up style. This is certainly a worthwhile scientific goal. However, emergent systems are notoriously difficult to understand, since their knowledge and activity is concealed as a collection of opaque numerical values. This makes them less suitable as a vehicle for encouraging constructionist novice programming, which is top-down (in some sense) by nature.

Of course DA must make use of numerical values too, as a last-resort mechanism for resolving conflicts. In any distributed control system, such values act as a *lingua franca* for making comparisons between agents. However, the semantics of such values are difficult to define and their opacity makes systems that use them hard to understand. One alternative is to use symbolic statements of preference: that is, explicitly state that behavior *a* has priority over behavior *b*, rather than hiding this fact behind numbers. This technique has been used in the Pengi and Sonja systems (Agre and Chapman 1987) (Chapman 1991). Symbolic Preferences are easier to understand, but must be specified explicitly for every new behavior. Another advantage they have is that they are easily specified by the user using a conflict display like the one described in section 5.1.1.3.

5.5.3 DA as a Constraint System

DA has been shown to be capable of performing some of the functions of a constraint system. Tasks can be declarations of relationship, and templates can be written that generate agents that enforce these relationships. Since templates can take arbitrary action, it is difficult to characterize exactly how powerful the constraint system is. The ability to match task patterns and generate new tasks is similar to the equational-rewriting techniques of Bertrand (Leler 1988). This ability can be used to implement the simple constraint-solving technique of local propagation of known states, as seen in the temperature conversion example and elsewhere (see section 5.2.7). It can also be used to implement special-purpose methods for solving constraints that cannot be performed by local propagation. For instance, a constraint of the form $(= a (* b b))$ cannot be solved for *b* by local propagation, but DA allows templates to be added that know how to compute square roots in this particular case. Inequalities are another instance where DA's agent-based approach works well. While they cannot be solved in general, special-purpose methods can be created to handle them in ways appropriate to the domain.

The main problem with DA is that its control system is limited and often hard to understand. Here is an example that illustrates some of these limits. The world below contains two **rectangle** tasks as well as the **within** task that is visible (see section 5.2.5.2). This latter task specifies that the solid rectangle is to be contained within the hollow rectangle. The effect is to allow the user to drag either object around, and have the other one adjust its position or size accordingly.

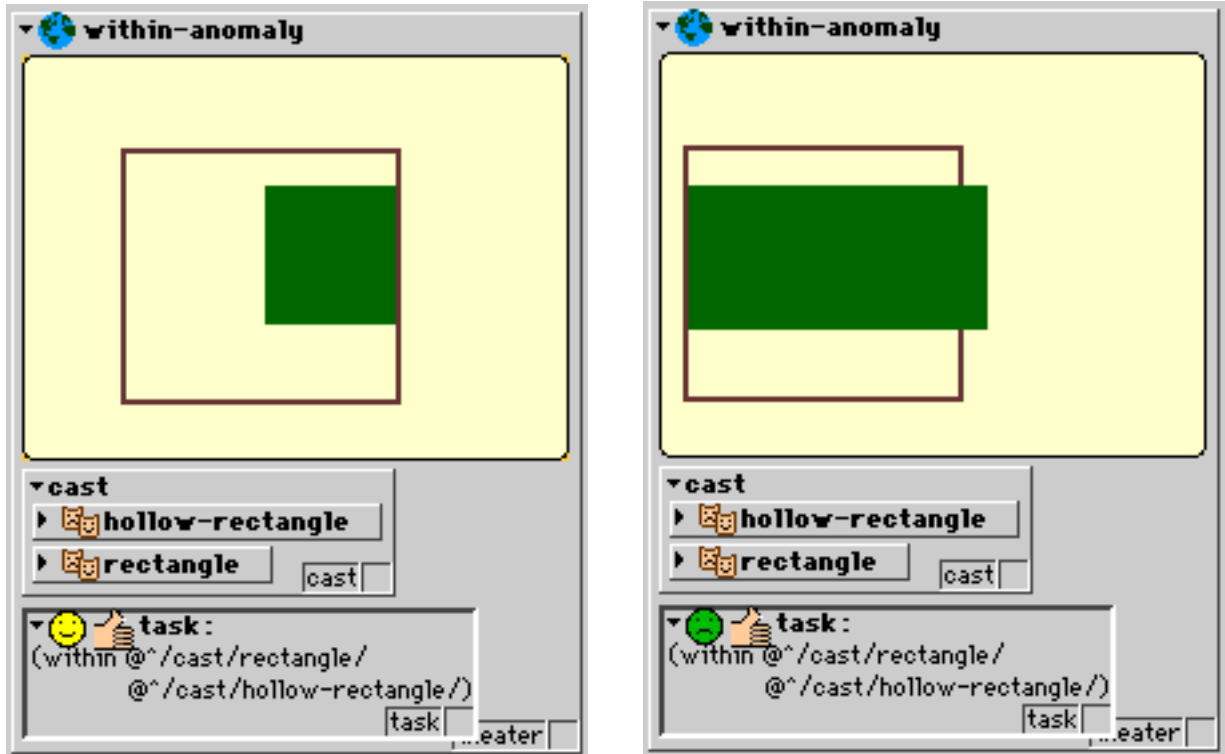


Figure 5.27: The **within** world, in working and anomalous states.

However, if the user should increase the size of the inner solid rectangle, the system fails to come to a stable state (that is, a state in which all top-level tasks are satisfied). Instead, both objects begin traveling to the left until they are off the stage!

The reason for this behavior is a limitation in the way conflicts are detected and handled. A worker agent's action will be suppressed (that is, fail) only if there is a slot conflict involved—that is, if it attempts to alter a slot that another more determined agent has laid claim to. As it turns out, in the situation above there is a conflict but it is a *goal* conflict: agents interfere with each other's goals, but alter mutually exclusive sets of slots (see section 5.2.6). Therefore, rather than picking a better way of resolving the conflict, the agents just alternate their actions ad infinitum.

There are several possible solutions to this problem: one is to support undoing an action. If an agent's action could be withdrawn if it was found to cause a goal conflict with a stronger agent, then it could be undone. Another, perhaps more general technique is to employ another type of manager agent that is separate from the main hierarchy of agents and monitors it for looping behavior, and can intervene if necessary (this idea derives from the Society of Mind concept of a *B-Brain* (Minsky 1987)). In fact, these ideas were explored in various early versions of DA, but rejected as too difficult to fit into the agent model. The current model is a compromise between power and simplicity.

5.5.3.1 Related Work

There is a large body of work on constraint-based problem solving, and a smaller one on interactive graphic systems based on constraints, including the classic systems Sketchpad (Sutherland 1963) and ThingLab (Borning 1979) as well as more recent efforts such as Juno-2 (Heydon and Nelson 1994). Garnet (Myers, Giuse et al. 1994), while not providing true constraints (it has so-called “one-way constraints”, which are more like spreadsheet cells or LiveWorld’s self-computing slots) is of interest because it also explores prototype-based object systems in the context of interaction. ThingLab, in particular, is oriented towards creating dynamic simulations as well as static drawings.

(Ishizaki 1996) applied agents to problems of graphic design. The emphasis of this work was different, however. Rather than solving static constraints, agents were employed to create dynamic designs, or to endow graphic elements with responsive behavior. For instance, in a Dynamic News Display, an agent would be in charge of creating and placing a headline when a news story arrived, based on its contents and the current display. The system (called Model of Dynamic Design) had three levels of control, agents, strategies, and actions, which correspond roughly to top-level agents, manager agents, and worker agents in DA. However, there is no concept of conflict resolution or constraint solving in the model.

5.5.4 DA and Anthropomorphism

DA agents are intended to be understood through animate metaphors. Recall that for our purposes animacy was defined by three key properties:

- autonomy (the ability to initiate action);
- purposefulness (being directed towards a goal);
- reactivity (the ability to respond to changes in the environment).

These qualities are woven into the way DA agents work. An agent is continually clocked (subject to being activated by its superior), which allows it to take action whenever it needs to. It should be emphasized that the clocking process in DA and the other agents systems exists in some sense at a level beneath the animate metaphor, in the sense that in a person, a heartbeat is something that is necessary for action but is not normally considered a factor in the process leading to action. The clocking process (based on animas in the simpler agent systems) does, in fact, serve to give the agents that are clocked the equivalent of both an energy source and a metronomic pulse that can be harnessed to drive action when necessary. This clocking allows an agent to appear autonomous by initiating action at any time. It is equally responsible for allowing agents to be responsive to changing conditions (indeed, autonomy and responsiveness are really two sides of the same coin (see section 3.4.1). Purposefulness is realized by allowing agents to be controlled by goal-like tasks.

The anthropomorphic icons used in agent boxes serve to indicate, in a limited way, the state of the agents vis-a-vis their goals and actions. Two different displays were created that make use of these icons. The agent graph shows all the agents in the system in their hierarchical structure, and updates them as the system runs through its cycles. It can be used to watch the

agents states change in real time. The other form of display is the storyboard, which is generated after the fact and selectively presents the dynamics of the agent system in a static narrative form.

5.5.4.1 Creation of Agents

One aspect of the way the agent system works seems to violate the anthropomorphic metaphor, or at least strain it. This is the manner in which new agents come into being. When a template specifies that new tasks are to be created, new agents are also created to embody and supervise the tasks. However, this means that agents are constantly coming into being (and are discarded rather unceremoniously as garbage when they have outlived their usefulness). This is a rather callow way to treat anthropomorphic beings! Of course, the ease of creating agents is also one of the chief features of DA, so this problem is fundamental in some sense.

One way to fix this is simply to change the metaphor a bit, so that agents are *recruited* instead of created. In this variant of the metaphor, tasks are created, but agents are drawn from a pool of pre-existing agents to deal with the tasks as they are needed. When the task is completed, the agent (and any subordinates it might have recruited) are returned to the pool.

While this may make agents appear more like real living entities, it complicates the implementation and its presentation. Now there is a new entity, the pool, and there will be the problem of how many agents it should be staffed with, what happens when it runs out, and so forth.

In both variants of the metaphor, an agent is a general-purpose worker who can perform any task. The knowledge of how to perform these tasks come from templates (which might be metaphorically presented as making up an “employee’s manual”, were we to extend the corporate metaphor a bit more). Another possible modification to the metaphor would be to conflate agents and templates. In this variant, an agent would be specialized for particular tasks, and emerge from idleness to deal with them as necessary. The problem with this is that there may be a need for multiple agents of the same type, so there is still the problem of agent creation, which now can’t even be handled by having a pool of general purpose agents (see the next section for further discussion). Perhaps agents could train other “new hire” agents in their expertise.

All this shows that the details of the mapping between the animate and computational domains is not at all straightforward. In this case the ease with which software entities come into being, which is a principle source of the power of any programming system, is in direct tension with anthropomorphism.

5.5.4.2 Agents, Tasks, and Templates

The fact that there are three different but closely related entities in DA—tasks, agents, and templates—seems somewhat unfortunate. Although they all serve necessary and separate functions, it is confusing and can strain the anthropomorphic metaphor.

While developing DA, I resisted the separation of agents and templates for a long time. Most of the earlier prototypes did not have this distinction. In my earlier images of how the system

should work, based on Society of Mind, agents are both program specifications and concrete instantiated mechanisms. Rather than having procedural instantiations and bound variables, their “arguments” are supplied again by fixed hardware buses called pronomes (Minsky 1987, p226). This picture may be more biologically realistic (in the sense that functionality is closely linked to a piece of hardware), but a programming language that works this way is severely handicapped by the lack of basic tools such as recursion. It is conceivable to add these capabilities to a Society of Mind-based model with the addition of special stack memory that incorporates K-lines for remembering the state of agents, but this is awkward. Some early implementations of agents did work this way. However, my devotion to realism gave way to a greater need for simplicity. Working without the separation between program and invocation felt like going back in time to older languages like FORTRAN that did not support recursion. To address these problems, I developed the idea of templates as entities separate from agents.

It might be possible to hide this split from the user. For instance, it might be possible to combine the functionality of agents and templates, call the results “agents”, and allow them to make copies of themselves, thus preserving the ability to support multiple instantiations. It is also probably possible to de-emphasize tasks as a separate entity, since tasks are mostly matched one-to-one with agents. There still needs to be some way to talk about the specification of an agent’s job (the task) as separate from the agent itself, which includes a good deal of other information besides the task.

5.5.4.3 Are Agents Too Low-level?

Each agent is very simple, just embodying an expression from the rather basic task language and executing a simple interpreter to achieve them. This simplicity makes for an understandable system, yet it strains somewhat against the anthropomorphic metaphor. Can such simple things be thought of in animate terms? In particular, some constructs force us to posit more “people” in the agent hierarchy than you might expect (i.e. the **repeat** construct forms an agent whose only job is to tell its one subordinate to continue doing whatever it is already trying to do²⁰). The graphic version of Fahrenheit/centigrade conversion, for instance, can use up to 70 agents to accomplish its task.

One early version of DA attempted to deal with this problem by compressing tasks into an agent that was slightly more complex than the final version. This agent could have both a goal and an action; that is, it could combine what would be two agents in the final version. Because the pattern of task-generation often alternates between goals and actions, this had the effect of generating about half as many agents as in the final version. It also helped strengthen the notion of an agent as something that tied together declarative and procedural information. However, the process of packing tasks two-to-an-agent was complex and confusing, and so I decided to use the simpler form of agent described here.

This problem would only be worse in a system that was uniformly agent-based (see section 6.3.4). If all computation was done with agents, rather than offloading some tasks to Lisp, there

²⁰This may in fact accurately model the role of many real-world managers.

would be at least an order of magnitude more agents to deal with. The solution, then, might be not to reduce the total number of agents but to use some form of hierarchical display to normally hide the low-level ones from the view of the user. This is already done to some extent in the current agent display and storyboard system, and is in keeping with the general interface style of LiveWorld.

5.5.4.4 Variants of Anthropomorphic Mapping

Part of LiveWorld's general design aesthetic is to not hide anything from the user. Detail must often be suppressed to save screen resources and the user's attentional resources, but it should always be ultimately accessible. Another design goal is simplicity and regularity. These goals may sometimes be in conflict with the goal of using anthropomorphic metaphors. It may help convey the anthropomorphic feel of the system to suppress some of the internals of the system. For instance, the **god** agent strains the metaphor of the system at several points. It exists primarily for the sake of *internal* regularity, making it possible to treat the entire group of agents as a single structure. But from the user's point of view, it is at variance with the metaphor of distributed agents. Why are we gathering the diverse set of top-level tasks under a single director? Perhaps it would be better to hide **god** from the user, preserving the impression of independent agents for top-level tasks.

Another instance in which the anthropomorphic mapping is not simple is in the area of emotional icons. We would like to use the emotional state anger to indicate that two nodes are in conflict. But this can't be done with a straightforward mapping between the agent states and the displayed emotional states, because in the usual case, at any given time one of the nodes involved in a conflict is going to be satisfied (its goal is currently met) and one is not. Satisfaction and being-in-conflict are thus somewhat orthogonal properties of nodes, and if they were translated straightforwardly into a screen representation, there would have to be an angry face *and* a smiling face displayed (in fact an early version of the system worked in just this way). However, this would strain the anthropomorphic metaphor, which dictates that an agent present a single face to the world. The solution was to just show the angry face, which encoded the more salient of the node properties. Another solution would be to have richer icons that could represent angry-and-satisfied and angry-and-unsatisfied in a single expression, or even better, generate faces parametrically rather than using fixed icons. This opens up the possibility for incorporating all relevant information about an agent (satisfaction, determination, success, conflict status) in a single highly expressive facial representation generated on the fly (Thórisson 1996). Humans can read a lot into a face—whether the computer can write out its state in such a way that it can be so read remains to be seen.

5.6 Summary

The agent metaphor suggests that programs be built out of components that are designed to be understood in anthropomorphic terms. In order to achieve this, we attempt to endow our agents with purpose, autonomy, and the ability to react to events in their world. This chapter presents a series of agent systems and shows that the metaphor can be used to unite several disparate styles of computation:

- the behavior-module-based, action-selection approach to the generation of creature behavior (Tinbergen, Brooks, Maes);
- the dynamics, generative power, and ability to deal with sequential operations found in procedural programming languages (Lisp, Logo);
- the ability to integrate declarative and procedural information found in early constraint systems (Sketchpad, ThingLab).

DA is the most powerful agent system and integrates these three strands most fully, but it is somewhat hard to understand. The simpler agent systems lack the full power of a programming language, but are simple to understand and lend themselves to the drag-and-drop, mix-and-match construction paradigm of LiveWorld.

You can't have everything. Where would you put it?
– Steven Wright

6.1 Summary and Contributions

This dissertation was motivated by a desire to find new ways of thinking about action, programming, and the control of animate systems. To this end, it has explored a particular approach to programming based on the central idea of animate agents. There are three main contributions:

- An analysis of how metaphors in general, and animate metaphors in particular, are used in the construction and comprehension of computation and programming languages;
- A series of agent-based programming systems that take the insights of the above analysis into account;
- An environment, LiveWorld, that is designed to support the construction of agent-based systems.

The analysis of metaphor and animacy (chapters 2 and 3) is an attempt to grapple with the informal conceptual underpinnings of a domain usually presented in formal terms. The notion of metaphorical models was chosen as the vehicle for this exploration because of its success in revealing the structure of everyday domains. While metaphor has been studied in the realm of computer interfaces, there has been very little prior study of the metaphors underlying programming languages. This section of the dissertation thus took a broad approach to the subject. Some effort was necessary to recover the ability to detect the presence of metaphors that have become transparent. A number of established programming paradigms were subjected to an analysis based on their underlying metaphors.

Particular attention was paid to the role of animate metaphors in these paradigms, and in computation generally. It was found that animate metaphors are part of the foundation of computation, and play a role in most (but not all) programming paradigms. However, the form of animacy used to construct computation is of a peculiarly limited sort that relies on images of humans behaving in a rote or mechanical fashion. Some of the key properties that otherwise define the animate realm, such as autonomy and purposefulness, are missing from this standard form of computational animism.

Agent-based programming is then introduced as an organizing metaphor for computation that includes these properties. A variety of computational techniques for realizing agents were examined. Different aspects of agent-like functionality can be found in the computational world in the form of processes, rules, objects, goals, procedures, and behavior modules. The task of designing an agent-based programming system is to use these elements to generate a programming system that is both powerful and can be seen in animate terms.

Chapter 5 presented a number of such systems. Simple Agents is a minimalist version of an agent, offering only concurrency and a form of conflict detection and resolution. Goal Agents extends this model by introducing a more structured version of an agent that includes a goal. It was shown that this can provide the necessary foundation for anthropomorphic interfaces and for organizing systems of agents. Dynamic Agents is a more powerful system that introduces dynamic creation of new agents, hierarchical control, sequencing, and a variety of control constructs. This system approaches the generality of a full-powered programming language using the agent metaphor. The utility of the system for behavioral simulation, constraint problems, and a variety of other uses is explored.

The LiveWorld environment (chapter 4) was conceived originally as a platform or tool for the exploration into agent-based programming languages, but contains some original contributions in its own right. In particular, it combines Boxer's style of hierarchical concrete presentation in the interface with an underlying object system based on recursive annotation and inheritance through prototypes. LiveWorld extends the underlying object system, Framer, in order to make it operate in a dynamic interactive environment. LiveWorld contains many features to support agent-based programming, including sensors that allow objects to react to one another, and animas that allow autonomous processes to be easily introduced into a constructed world.

6.2 Related Work

The individual chapters in the body of the thesis discuss work that is related to the separate themes of this thesis. Here, we discuss a few systems that are related to more than one theme: that is, some other environments that are designed for novices, strive for a lively feel, and contain programming capabilities that are in some sense agent-like.

6.2.1. KidSim

KidSim (Cypher and Smith 1995) is a programming environment designed to allow children to create graphic simulations, using a rule-based approach to programming. KidSim has a colorful interface which permits direct manipulation of a variety of objects (i.e. agents, costumes, and rules). In KidSim, an agent is equivalent to what LiveWorld calls actors or creatures (and what KidSim calls a rule, we would call an agent). Each agent has a set of rules and a set of properties (slots) that are accessible through a graphic editor. While you cannot create entirely new kinds of objects as in LiveWorld, agents once created can be used as a basis for creating more agents of the same type. KidSim originally had the ability to support a deep inheritance hierarchy as does LiveWorld, but this feature was eliminated because it was judged to be too confusing.

KidSim avoids text-based programming languages, instead using a form of programming by demonstration to generate graphic rewrite rules, which can then be modified in an editor. The world of KidSim is based on a grid of cells which can be occupied by various graphic objects. The rewrite rules thus operate on patterns of cell occupancy. Each agent has a set of rules which can be arranged in order of priority.

KidSim is based on a simple idea which makes it easy to learn, yet permits creating a wide range of simulations. However, the rewrite rule approach makes some things difficult. For instance, if an object is to both move and deal with obstacles in its path, it must be given separate rule sets for each of the four possible directions of movement. More fundamentally, pure rule-based systems have difficulties dealing with sequential action and control in general. They also do not lend themselves to building powerful abstractions in the way that procedural languages do, and thus are limited in their ability to deal with complexity.

6.2.2. Agentsheets

Agentsheets (Repenning 1993) is not an end-user environment itself, but a tool for creating a variety of domain-specific graphical construction systems. Each such system includes a set of parts (agents) which the end-user can arrange on gridded field (the agentsheet). The end-user cannot construct the agents themselves; instead these are designed and built by a separate class of expert designers. Agents in this system are autonomous objects that contain sensors, effectors, behavior, state, and a graphical depiction. Agents can move from cell to cell, although not all of the visual languages supported by the system make use of this feature. They can also sense and communicate with their neighbors. Their behavior is specified by a Lisp-embedded language called AgenTalk. (In fact there are other ways to program agents, such as specifying graphical rewrite rules as in KidSim, but most of the system's flexibility comes from the AgenTalk level).

This simple scheme supports a wide variety of metaphors. In flow-based languages, agents are static but change their state based on their neighbor, and so can simulate electrical circuits or fluid flow. In anthropomorphic metaphors, the agents represent creatures that move around in the world and interact with one another, and can support behavioral simulations or certain types of videogames. Limited programming can be done at the end-user level with languages that provide agents with control-flow metaphors (one example is a visual language for designing the flow of automatic phone answering services).

Agentsheets has been shown to be a very flexible system, but its full range is only accessible to experts. There is a sharp division between the end user, who can only manipulate a set of pre-constructed objects, and the expert designer who can build new kinds of objects and new languages. This is in contrast to LiveWorld's goal of making as much of the inner workings of the system accessible to the end-user as possible.

6.2.3. ToonTalk

ToonTalk is the most unusual of the environments described here, and perhaps the most powerful. Its interface is entirely visual and highly animated, and its underlying computational model is a version of Concurrent Constraint Programming (CCP) (Saraswat 1993), an advanced programming paradigm descended from logic programming. The system provides an elaborate mapping from the constructs of CCP to animated objects. For example, methods are shown as robots; their programs exist in thought bubbles; a communication is a bird flying out of its nest and back; terminating an agent is represented with an exploding bomb. Just as Logo was

designed as a child-friendly version of Lisp, ToonTalk is a version of CCP in which mathematical formalisms are replaced by animated characters.

ToonTalk is extremely lively, sometimes to the point of causing confusion. The underlying language is inherently concurrent, and so is the environment. Graphic objects or sprites form the basis of user worlds. Games and simulations on the order of Pong can be readily constructed. The programming style is a form of programming by example. The user performs a sequence of concrete actions using a set of tools, which can be recorded into a robot (method). The user can then make the resulting programs more abstract by explicitly removing constants from them.

Program execution is animated, and since there is no textual code other than the graphic actors, Kahn is justified in claiming that the programs themselves are in the form of animations, not merely illustrated by them. Whether this is really easier to understand than a textual description is debatable, however. While the idiom of animated video characters might be more immediately accessible to a child, it is hard to follow a complex set of moving objects. Static descriptions of programs and their activity (such as storyboards; see section 5.3.5) have the advantage of standing still so you can read them!

Another problem is that some of these metaphors seemed strained—for instance, agents and objects are represented as houses, presumably because they contain an assortment of other objects, while an entire computation system is a city. However, when this metaphor is extended via recursion it leads to expectation violations such as objects that have within them whole cities. In other cases, the metaphor is apt but not extensible: for instance, a scale is used to indicate a comparison between numbers, but this metaphor does not readily extend to other predicates.

CCP comes from an essentially declarative logic programming tradition, which ordinarily would make it an awkward tool for programming dynamic systems. Nevertheless ToonTalk manages to represent action and perception using extensions to the basic model in the form of sensors and “remote control variables” which cause side effects. Because of these, ToonTalk is quite capable of building animate systems such as videogames despite its declarative heritage.

6.3 Directions for further research

6.3.1 What Can Novices Do with Agents?

LiveWorld has had only casual testing with novice programmers. Several students succeeded in building video-game worlds using simple agents and other objects drawn from a library of video-game components (see sections 4.2.10 and 5.4.1). Obviously, more evaluation with users is needed: of LiveWorld, its implementations of agents, and the basic ideas of agent-based programming.

Dynamic Agents, at least in its present form, is probably too complex for novices. While the basic concepts of the language are not that much more complicated than those found in a

procedural language, and it should not be beyond the ability of a novice to create tasks and templates, it can be very difficult to understand the resultant complex concurrent activity of the agent system. A scaled-back version of DA that eliminated concurrency and conflict detection, but retained the goal-directed control structure, might be more tractable although less powerful. There are many possible languages that lie on the continuum between a basic procedural language and DA, allowing for a tradeoff between power and comprehensibility. At the low-powered end of the spectrum, there are concepts from agent-based programming that could be usefully transferred to purely procedural languages, such as turning invocations into graphic objects and displaying them anthropomorphically.

The simpler agent systems might be more promising places to start evaluating the utility of agent-based programming for novices. The ideas behind these systems are sufficiently simple and intuitive (at least to recent generations that have grown up with video games) that they should be usable by young children. Work with children on programming environments with similar capabilities is encouraging: (Marion 1992) describes elementary-school students building animal behavior simulations in PlayGround; (Burgoin 1990) describes a group of somewhat older children building robots that are controlled by a rule-like language; and the systems described in section 6.2 also have shown that children can create behavioral simulations using agent-like constructs. None of these systems use explicit goals or anthropomorphize their agents, so it remains to be seen whether these features make agents easier to understand or not. It would also be interesting to study the nature of children's anthropomorphization of computers and program parts, and whether these tendencies are different in agent-based systems.

The understandability of any programming system will be based both on the underlying concepts and on how well those concepts are presented to the user through the interface. Thus, the answers to this question and the next will be linked together.

6.3.2 Can Agent Activity Be Made More Understandable to the User?

Like any complex system, agent systems can be hard to understand. However, agent systems also suggest new techniques for presenting program activity. The use of narrative techniques to explain (and eventually to produce) agent activity is one of the more interesting areas opened up by agent-based programming. The storyboards that DA can generate only begin to explore this area. Presenting the activity of complex collections of hundreds of agents will require more knowledge about characters and narrative.

The representation of agents in anthropomorphic terms can be extended. There are additional emotional states that can be used to reflect computational states (such as surprise to indicate an expectation violation (Kozierok 1993)). The magnitude of emotions such as happiness or anger can be conveyed by parameterizing facial displays (see section 5.5.4.4). Perhaps more importantly, there is a need to individualize agents so that they can be thought of as characters. Libraries of animated characters could be selected as representations that match the characteristics of the underlying agents. This idea was explored in (Travers and Davis

1993), but only for a two-agent system. Extending this idea to more complex systems of agents is an interesting challenge.

The mappings from agent activity to narrative forms can also be extended. The stories told now are all of a fairly simple type, and the technique used to tell them is simplistic. We can imagine other kinds of stories when agent systems become more sophisticated, for instance, stories about conflict being resolved into cooperation, or developmental stories in which a character learns from experience. As for the presentational forms, the medium of the storyboard has many rich possibilities, such as parallel time tracks and highlighting details, that have yet to be explored.

6.3.3 Can Agent Systems Be Made More Powerful?

The Dynamic Agents system is somewhat limited in terms of the intelligence it can apply to performing its tasks. While it has some support for exploring multiple methods of achieving a goal, it cannot perform searches for solutions as do planners. Since an agent may perform arbitrary actions, it is difficult to represent the projected results of a potential action, which is planning requires. More fundamentally, putting actions under the control of a centralized planner violates the agent metaphor. Decentralized methods that simulate some of the effects of planning (such as Maes' action selection algorithm; see section 5.5.2) might be an acceptable substitute.

Other methods for improving DA performance that are more consonant with the agent metaphor are also possible. For instance, there could be agents that are outside of the standard agent hierarchy and act to improve its performance (see section 5.5.3). Manager agents like this could also be used for more sophisticated forms of conflict resolution than the contest-of-strength technique currently employed by LiveWorld. Rather than have two conflicting agents fight it out, they could refer their conflict to an outside mediator who could implement negotiation of compromises or make use of non-local information to help resolve the dispute.

Agents currently do not learn from experience, and thus have to solve each constraint problem all over again each time any element of the system changes. Smarter agents could remember which methods worked and under which circumstances, and make the correct choices without having to search through failures. Mediator agents too could have memories. Some of these techniques were explored, for extremely simple cases, in (Travers and Davis 1993). Case-based techniques were used for learning, so that situations and solutions would be remembered in a case library and matched with new situations as they arose. However, the challenge is to implement such facilities in a way that fits into the overall metaphor and makes the behavior of the system more understandable rather than more opaque. There is a tension between making the system smarter and making it more comprehensible.

6.3.4 Can Agents be Organized in Different Ways?

The Dynamic Agents system is based on a hierarchical model of agent organization, and employs metaphors based on human managerial hierarchies. This form of organization is

common in computational systems, but is problematic to some because of its metaphorical implications. Bureaucratic hierarchies can be unpleasant and inefficient in real life, so why duplicate them in the computer?

Hierarchy arises in computational systems for the same reason it does in society: to allow a measure of centralized control to be exerted over a domain that is too big or complex to be controlled by a single individual or agent. Hierarchy allows parts of the domain to be controlled by subordinate agents while keeping ultimate control centralized. This control is not obtained without a cost, however. In social systems, the costs include the ethical and political consequences that attend the subordination of individuals to the power of others. This sort of cost is presumably not a direct issue when designing a software system around social and anthropomorphic metaphors, except insofar as it might promote the idea of hierarchical subordination in general.

But there are other problems with hierarchical organization that can affect both human and artificial societies. These stem from the fact that such systems lead to long and sometimes unreliable paths of communication between the worlds they manipulate and the agents that make decisions. Information about the world must percolate upwards through a chain of command while decisions filter downwards. This process can be slow and error-prone. Hierarchical organization disempowers agents who are most in touch with the world and isolates the agents with power from the world they seek to control. Allowing subordinates a greater degree of independence can alleviate some of these problems, but at the risk (from the manager's perspective) of losing control and causing incoherent system behavior. As a result, hierarchical organizations are constantly seeking a balance between top-down control and autonomy.

The DA system was also designed to seek a balance between control and autonomy. In contrast to procedural languages, which use a strict top-down command regime, DA employs a more flexible style of control. While they are subject to hierarchical management structure, dynamic agents execute autonomously and concurrently, in the manner similar to that employed in hierarchical behavioral control systems (see 5.2). Managers actually manage, rather than command: their role is to activate and coordinate the activities of their subordinates, rather than to directly control them. The control path between perception, decision-making and action is shortened, and most commonly only involves a single level of agent.

Certainly there are other ways to organize systems of agents that avoid hierarchical models and metaphors. Perhaps an agent system could be organized around the idea of group negotiation towards a consensus, after the model of Quaker meetings. Such systems suffer from scaling problems, however, and presuppose a more sophisticated agent that is capable of representing and negotiating among alternative courses of action. Another mode of organization which avoids hierarchy is a market-based (or "agoric") system in which agents hire others to perform tasks for them (Miller and Drexler 1988). However, in such systems the relationship between agents is looser, and it is correspondingly harder for a manager to manage the activity of a "hired" agent when (for example) the subtask no longer needs to be accomplished because circumstances have changed (this sort of constraint is presumably the reason real-world economic activity makes use of both hierarchical firms and non-hierarchical markets).

All non-hierarchical schemes have serious problems with comprehensibility. A large system of agents is a formidable thing to understand, and organizing agents along hierarchical lines is the most powerful way to make such a collection into something that can be understood. The true advantage of hierarchy, from the standpoint of comprehensibility, is that it is the scheme that allows the details of agent activity to be abstracted away. This is one reason why hierarchical organization is so highly valued in the design of engineered systems.

The question of control in agent-based systems is fraught with political overtones, and forces designers and users to think about issues of control, autonomy, power, and organization in the human world as well as the computational. There are many possible ways to organize the activity of agents. From the standpoint of constructivist education, what would be most desirable is a system that allowed users to create and explore a variety of organizational forms.

6.3.5 Can the Environment Be Agents All the Way Down?

One legitimate criticism of the system as a whole is that it has a split personality: the basic LiveWorld level and the agent languages are to some extent separate, and organized around different metaphors. LiveWorld uses an unusual form of the OOP paradigm, highlighted by an interface which emphasizes the tangibility of objects. The DA system uses the agent metaphor which emphasizes action, goal-seeking, and autonomous behavior. This is partially an artifact of the system's developmental history—the object system had to be in place before the research into agents could take place. But in the future, wouldn't it be better to use a single paradigm for the system? Can an environment be made that is “agents all the way down”?

There are several answers to this question. For one thing, it is not clear that a single metaphor can really work at all levels. Agents need to take their action in a world of objects, so in some sense the operation of the agent-based level depends upon an object level already being in place. Making everything an agent, the way some OOP systems make everything an object, might require diluting the concept of agent to the point where it is no longer useful.

Still, there could be more integration between the levels. One interesting prospect is to base the agent system on a more suitable low-level general computational mechanism than Lisp and the LiveWorld object system. One candidate is the Actors programming formalism (Hewitt 1976), which already supports object-orientation, concurrency, and message-passing. Actor languages in theory could replace Lisp as the base language in which agents are programmed. Actors are already quite similar to dynamic agents in some ways. An actor is like a procedure invocation, but can also persist over time and have state. Unlike procedures, but like dynamic agents, actors do not implicitly return values. If an invoker of an actor wants a response, it must create another actor as a continuation to handle the value, and pass the continuation along as an argument.

But actors are still essentially driven from external messages rather than running as autonomously independent processes (Agha, Mason et al. 1993). They are not goal-oriented and have no built-in models for conflict resolution. Thus if Actors is to serve as a basis for an agent system like DA, it will be necessary to build a layer of agent-based protocols on top of the

basic Actor model. Another problem is that the actor model has not, as yet, resulted in many practical programming tools. Actor programming can be difficult, unnatural, and verbose, and actor languages do not have the sophisticated inheritance mechanisms found in other object-oriented systems. Nonetheless the Actor model looks like a promising way to put agent-based programming on a more elegant and rigorous footing.

6.3.6 Agents in Shared Worlds

Animate systems naturally lend themselves to implementation in distributed and multi-user environments. In such environments, users at various locations would construct active objects that would be loosed into common areas to interact with one another. Text-based systems of this kind (called MUDs or Multi-User Dungeons) have become widespread in recent years, and show promise as environments for learning to program. The social nature of such artificial worlds provides an audience, context, and motivation for constructionist learning (Bruckman 1993).

However, the text-based nature of MUDs limits the kinds of interactions that can take place, and the programming languages that are available are often limited in what they can express. Some MUDs are inhabited by sophisticated artificial personalities (Foner 1993), but these generally are developed outside of the MUD itself, and simulate a user connected to it from the outside. Users cannot generally construct such complex autonomous actors from within the environment.

The ideas of agent-based programming and the LiveWorld environment should fit naturally into shared worlds and might improve their utility. Programs in shared environments have to respond to a variety of events and changing circumstances, a task well-suited to the agent-based paradigm. LiveWorld's design encourages thinking of programs and behaviors as objects that can be manipulated inside of a graphic environment. This objectification of program components should be particularly useful in a shared social environment, allowing programs to become social objects that can be readily discussed and shared.

6.4 Last Word

We are entering an era in which a multitude of new forms of media will be invented and deployed. Whether these new forms become tools for thought or mere distractions depends upon how easy it is for non-specialists to use them to express their ideas. Programmability has been neglected in the rush to develop new media, but what is new and interesting about the new digital forms, namely their interactivity and dynamism, require some grasp of the art of programming in order to be fully utilized.

Programming should thus be thought of as a basic intellectual skill, comparable to writing. Like writing, programming has to be learned, sometimes laboriously; and while it can certainly be made simpler and more accessible than it has been, it cannot and should not be reduced to triviality or eliminated. In my own experience, genuine learning takes place when the learner is motivated to learn, when the topic is interesting for its own sake or is a means to a genuinely

desired end. Programming can be either, but too often is presented in a dystonic manner unconnected with the learner's actual interests. And too often the tools that programming environments provide are too weak to allow a beginner to advance to the point where their skills can match their vision.

My project has been an effort to set programming in new contexts and put a new face on it. Rather than framing computing and programming as a means of manipulating abstract and disembodied data, LiveWorld frames it as a task of building concrete autonomous graphic worlds, and supplies the tools necessary for building such worlds. Rather than thinking of programs in formal or mechanical terms, agent-based programming sees them in animate terms, with purposes, reactions, and conflicts of their own. The hope is that thinking about these systems can engage the learner's thinking about purpose, reaction, and conflict in the wider world, a world that is inhabited by and largely determined by the purposeful mechanisms of life.

Bibliography

Abelson, H. and G. J. Sussman (1985). *Structure and Interpretation of Computer Programs*. Cambridge MA: MIT Press.

Ackermann, E. (1991). The "Agency" Model of Transactions: Toward an Understanding of Children's Theory of Control. In *Constructionism*, edited by I. Harel and S. Papert. Norwood, NJ: Ablex.

Agha, G. A. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge: MIT Press.

Agha, G. A., I. A. Mason, et al. (1993). "A Foundation for Actor Computation." *Journal of Functional Programming* 1(1).

Agre, P. E. (1992). "Formalization as a Social Project." *Quarterly Newsletter of the Laboratory of Comparative Human Cognition* 14(1), 25-27.

Agre, P. E. (1995). "Computational Research on Interaction and Agency." *Artificial Intelligence* 72((1-2)), 1-52.

Agre, P. E. (1996). *Computation and Human Experience*. Cambridge University Press.

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

Apple Computer (1987). *Apple Human Interface Guidelines: The Apple Desktop Interface*. Reading, Massachusetts: Addison-Wesley.

Apple Computer (1991). Knowledge Navigator. Videotape.

Apple Computer (1992). *Dylan: an object-oriented dynamic language*.

Baron-Cohen, S. (1995). *Mindblindness: An Essay on Autism and Theory of Mind*. Cambridge, Massachusetts: MIT Press.

Bateson, M. C. (1972). *Our Own Metaphor*. New York: Knopf.

Black, M. (1962). *Models and Metaphors*. Ithaca, NY: Cornell University Press.

Boden, M. A. (1978). *Purposive Explanation in Psychology*. Harvester Press.

Bond, A. H. and L. Gasser, Eds. (1988). *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann.

Borges, J. L. (1962). *Labyrinths: Selected Stories & Other Writings*. New Directions.

- Borning, A. (1979). Thinglab: A Constraint-Oriented Simulation Laboratory. Xerox PARC SSL-79-3, July 1979.
- Borning, A. (1986). "Defining Constraints Graphically", Proceedings of CHI'86, Boston.
- Boulay, B. d. (1989). Difficulties of learning to program. In *Studying the Novice Programmer*, edited by E. Soloway and J. C. Spohrer. Hillsdale, NJ: Lawrence Erlbaum.
- Boyd, R. (1993). Metaphor and theory change: What is "metaphor" a metaphor for? In *Metaphor and Thought*, edited by A. Ortony. Cambridge: Cambridge University Press.
- Braitenberg, V. (1984). *Vehicles: Experiments in Synthetic Psychology*. Cambridge: MIT Press.
- Brooks, R. A. (1986). Achieving Artificial Intelligence through Building Robots. Massachusetts Institute of Technology Artificial Intelligence Laboratory AI Memo 899.
- Brooks, R. A. (1986). "A Robust Layered Control System for a Mobile Robot." *IEEE Journal of Robotics and Automation* 2(1), 14-23.
- Brooks, R. A. (1991). "Intelligence Without Representation." *Artificial Intelligence* 47, 139-159.
- Bruckman, A. (1993). Context for Programming. MIT Thesis Proposal, August 6, 1993.
- Bruner, J. (1986). *Actual Minds, Possible Worlds*. Cambridge, Massachusetts: Harvard University Press.
- Bruner, J. S. (1966). On Cognitive Growth: I. In *Studies in Cognitive Growth*, edited by J. S. Bruner, R. R. Oliver and P. M. Greenfield. New York: Wiley.
- Burgoin, M. O. (1990). Using LEGO Robots to Explore Dynamics, MS thesis, MIT.
- Carey, S. (1985). *Conceptual Change in Childhood*. Cambridge, Massachusetts: MIT Press.
- Chang, B.-W. and D. Ungar (1993). "Animation: From Cartoons to the User Interface", Proceedings of UIST '93, Atlanta, Georgia.
- Chapman, D. (1991). *Vision, Instruction, and Action*. Cambridge, Massachusetts: MIT Press.
- Chapman, D. and P. E. Agre (1986). "Abstract Reasoning as Emergent from Concrete Activity", Proceedings of 1986 Workshop on Reasoning About Actions & Plans, Los Altos, California.
- Charles River Analytics (1994). Open Sesame. Software.
- Connell, J. (1989). A Colony Architecture for an Artificial Creature. MIT Artificial Intelligence Laboratory Technical Report 1151.
- Cypher, A., Ed. (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, MA, MIT Press.
- Cypher, A. and D. C. Smith (1995). "KidSim: End User Programming of Simulations", Proceedings of CHI'95, Denver.
- Dahl, O. J., B. Myhrhaug, et al. (1970). Simula Common Base Language. Norwegian Computing Center Technical Report S-22, October 1970.
- Davis, R. and R. G. Smith (1983). "Negotiation as a Metaphor for Distributed Problem Solving." *Artificial Intelligence* 20(1), 63-109.

- Dennett, D. C. (1987). *The Intentional Stance*. Cambridge, MA: MIT Press.
- Dennett, D. C. (1991). *Consciousness Explained*. Boston: Little Brown.
- DiGiano, C. J., R. M. Baecker, et al. (1993). "LogoMedia: A Sound-Enhanced Programming Environment for Monitoring Programming Behaviour", Proceedings of InterCHI '93, Amsterdam.
- Dijkstra, E. W. (1989). "On the Cruelty of Really Teaching Computing Science." *CACM* 32(12), 1398-1404.
- diSessa, A. A. (1986). Models of Computation. In *User Centered System Design*, edited by D. A. Norman and S. W. Draper. Hillsdale NJ: Lawrence Erlbaum Associates.
- diSessa, A. A. and H. Abelson (1986). "Boxer: A Reconstructible Computational Medium." *CACM* 29(9), 859-868.
- Dreyfus, H. (1979). *What Computers Can't Do: The Limits of Artificial Intelligence*. New York: Harper and Row.
- Edel, M. (1986). "The Tinkertoy Graphical Programming Environment", Proceedings of IEEE 1986 CompSoc.
- Etzioni, O. and D. Weld (1994). "A Softbot-based interface to the Internet." *CACM* 37(7), 72-76.
- Fenton, J. and K. Beck (1989). "Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages", Proceedings of OOPSLA '89.
- Ferber, J. and P. Carle (1990). Actors and Agents as Reflective Concurrent Objects: A MERING IV Perspective. LAFORIA, Université Paris Rapport LaForia 18/90, 1990.
- Finzer, W. and L. Gould (1984). "Programming by Rehearsal." *Byte* 9(6), 187-210.
- Fisher, D. (1970). Control Structure for Programming Languages, PhD thesis, CMU.
- Foner, L. N. (1993). What's an Agent, Anyway?: A Sociological Case Study. MIT Media Lab Agent Memo 93-01, May 1993.
- Forgy, C. L. (1982). "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem." *Artificial Intelligence* 19(1).
- Freeman-Benson, B. N., J. Maloney, et al. (1990). "An Incremental Constraint Solver." *CACM* 33(1), 54-63.
- Gamma, E., R. Helm, et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Gelman, R. and E. Spelke (1981). The development of thoughts about animate and inanimate objects: implications for research on social cognition. In *Social Cognitive Development*, edited by J. H. Flavell and L. Ross. Cambridge: Cambridge University Press.
- General Magic (1995). The Telescript Language Reference. General Magic, October, 1995.
- Gentner, D. (1989). The Mechanisms of Analogical Learning. In *Similarity and analogical reasoning*, edited by S. Visbadiy and A. Ortony. : Cambridge University Press.

- Gentner, D. and M. Jeziorski (1993). From metaphor to analogy in science. In *Metaphor and Thought*, edited by A. Ortony. Cambridge: Cambridge University Press.
- Gibbs, R. W. (1993). Process and products in making sense of tropes. In *Metaphor and Thought*, edited by A. Ortony. Cambridge: Cambridge University Press.
- Goffman, E. (1974). *Frame Analysis: An Essay on the Organization of Experience*. Boston: Northeastern University Press.
- Goldberg, A. and A. Kay (1976). Smalltalk-72 Instruction Manual. Xerox PARC Technical Report.
- Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley.
- Greiner, R. (1980). RLL-1: a Representation Language Language. Stanford Heuristic Programming Project HPP-80-9.
- Gross, P. R. and N. Levitt (1994). *Higher Superstition: the academic left and its quarrels with science*. Baltimore: Johns Hopkins University Press.
- Haase, K. (1992). Framer, MIT Media Lab. Software.
- Harvey, B. (1985). *Computer Science Logo Style: Intermediate Programming*. Cambridge, MA: MIT Press.
- Havermiste, C. (1988). The metaphysics of interface. Center for Computational Theology, University of Erewhon CCT Memo 23, May 23, 1988.
- Heims, S. J. (1991). *The Cybernetics Group*. Cambridge, MA: MIT Press.
- Henderson, P. (1980). Is it reasonable to implement a complete programming system in a purely functional style? University of Newcastle upon Tyne Computing Laboratory PMM/94, 11 December 1980.
- Hetenryck, P. V. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge MA: MIT Press.
- Hewitt, C. (1976). Viewing Control Structures as Patterns of Passing Messages. MIT AI Lab AI Memo 410, December 1976.
- Hewitt, C. (1986). "Offices are Open Systems." *ACM Transactions on Office Information Systems* 4(3), 271-287.
- Heydon, A. and G. Nelson (1994). The Juno-2 Constraint-Based Drawing Editor. Digital Systems Research Center SRC Research Report 131a, December 1994.
- Hinde, R. A. (1978). *Animal Behavior: A synthesis of ethology and comparative psychology*. New York: McGraw-Hill.
- Hudak, P., S. P. Jones, et al. (1991). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1). Yale University Department of Computer Science Technical Report YALEU/DCS/RR777, August 1991.
- Hutchins, E. L., J. D. Hollan, et al. (1986). Direct Manipulation Interfaces. In *User Centered System Design*, edited by D. A. Norman and S. W. Draper. Hillsdale NJ: Lawrence Erlbaum Associates.

- Ingalls, D., S. Wallace, et al. (1988). "Fabrik: A Visual Programming Environment", Proceedings of OOPSLA '88.
- Ishizaki, S. (1996). *Typographic Performance: Continuous Design Solutions as Emergent Behaviors of Active Agents*, PhD thesis, MIT.
- Jones, S. L. P. and P. Wadler (1993). "Imperative functional programming", Proceedings of ACM Symposium on Principles of Programming Languages, Charleston.
- Kaelbling, L. P. (1988). "Goals as Parallel Program Specifications", Proceedings of Proceedings AAAI-88 Seventh National Conference on Artificial Intelligence.
- Kay, A. (1981). *New Directions for Novice Programming in the 1980s*. Xerox PARC Technical Report.
- Kay, A. C. (1990). *User Interface: A Personal View*. In *The Art of Human-Computer Interface Design*, edited by B. Laurel. : Addison-Wesley.
- Keller, E. F. (1985). *Reflections on Gender and Science*. New Haven: Yale University Press.
- Kornfeld, W. A. and C. Hewitt (1981). *The Scientific Community Metaphor*. MIT AI Lab AI Memo 641.
- Kowalski, R. (1979). "Algorithms = Logic + Control." *CACM* 22(7), 424-436.
- Kozierok, R. (1993). *A learning approach to knowledge acquisition for intelligent interface agents.*, SM thesis, MIT.
- Kuhn, T. S. (1993). *Metaphor in Science*. In *Metaphor and Thought*, edited by A. Ortony. Cambridge: Cambridge University Press.
- Kurlander, D. and S. Feiner (1991). *Inferring Constraints from Multiple Snapshots*. Columbia University Department of Computer Science Technical Report CUCS 008-91, May 1991.
- Lakin, F. (1986). *Spatial Parsing for Visual Languages*. In *Visual Languages*, edited by S. K. Chang, T. Ichikawa and P. A. Ligomenides. New York: Plenum.
- Lakoff, G. (1987). *Women, Fire, and Dangerous Things*. Chicago: University of Chicago Press.
- Lakoff, G. (1993). *The contemporary theory of metaphor*. In *Metaphor and Thought*, edited by A. Ortony. Cambridge: Cambridge University Press.
- Lakoff, G. and M. Johnson (1980). *Metaphors We Live By*. Chicago: University of Chicago Press.
- Lakoff, G. and Z. Kövecz (1987). *The cognitive model of anger inherent in American English*. In *Cultural Models in Language & Thought*, edited by D. Holland and N. Quinn. Cambridge: Cambridge University Press.
- Lanier, J. (1995). "Agents of Alienation." *ACM Interactions* 2(3).
- Latour, B. (1987). *Science In Action*. Cambridge, Massachusetts: Harvard University Press.
- Laurel, B. (1990). *Interface Agents: Metaphors with Character*. In *The Art of Human-Computer Interface Design*, edited by B. Laurel. Reading, MA: Addison-Wesley.

- Leler, W. (1988). *Constraint Programming Languages: Their specification and generation*. Reading, MA: Addison-Wesley.
- Leslie, A. M. (1979). The representation of perceived causal connection, PhD thesis, Oxford.
- Leslie, A. M. and S. Keeble (1987). "Do six-month-old infants perceive causality?" *Cognition* 25, 265-288.
- Lieberman, H. (1986). "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", Proceedings of First ACM Conference on Object Oriented Programming Systems, Languages & Application, Portland.
- Lloyd, D. (1989). *Simple Minds* MIT Press.
- Lopez, G., B. Freeman-Benson, et al. (1993). Kaleidoscope: A Constraint Imperative Programming Language. Department of Computer Science and Engineering, University of Washington Technical Report 93-09-04, September, 1993.
- MacNeil, R. (1989). "TYRO: A Constraint Based Graphic Designer's Apprentice", Proceedings of IEEE Workshop on Visual Languages.
- Maes, P. (1987). "Concepts and Experiments in Computational Reflection", Proceedings of OOPSLA.
- Maes, P. (1989). "The Dynamics of Action Selection", Proceedings of Eleventh Joint Conference on Artificial Intelligence.
- Maes, P. (1990). Situated Agents Can Have Goals. In *Designing Autonomous Agents*, edited by P. Maes. Cambridge: MIT Press.
- Malone, T. W., R. E. Fikes, et al. (1988). Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments. In *The Ecology of Computation*, edited by B. A. Humberman. : Elsevier/North-Holland.
- Marchini, M. Q. and L. F. B. Melgarejo (1994). "Ágora: Groupware Metaphors in Object-Oriented Concurrent Programming", Proceedings of ECOOP 94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Bologna.
- Marion, A. (1992). Playground Paper.
- Martin, F. (1988). Children, cybernetics, and programmable turtles, MS thesis, MIT Media Laboratory.
- Mayer, R. E. (1979). "A psychology of learning BASIC." *Communications of the ACM* 22(11), 589-593.
- Mayer, R. E. (1989). How Novices Learn Computer Programming. In *Studying the Novice Programmer*, edited by E. Soloway and J. C. Spohrer. Hillsdale, NJ: Lawrence Erlbaum.
- McCloud, S. (1993). *Understanding Comics: The Invisible Art*. Northampton, MA: Tundra.
- McDermott, D. (1987). Artificial Intelligence Meets Natural Stupidity. In *Mind Design*, edited by J. Haugeland. Cambridge, Massacustts: MIT Press.
- Michotte, A. (1950). *The Perception of Causality*. New York: Basic Books.

- Miller, M. S. and K. E. Drexler (1988). Markets and Computation: Agoric Open Systems. In *The Ecology of Computation*, edited by B. Huberman. : North-Holland.
- Minsky, M. (1980). "K-lines: A Theory of Memory." *Cognitive Science* 4(2).
- Minsky, M. (1987). *Society of Mind*. New York: Simon & Schuster.
- Minsky, M. (1991). "Society of Mind: a response to four reviews." *Artificial Intelligence* 48, 371-396.
- Motherwell, L. (1988). Gender and style differences in a Logo-based environment, PhD thesis, MIT Media Laboratory.
- Myers, B. A., D. A. Giuse, et al. (1994). Making Structured Graphics and Constraints for Large-Scale Applications. Carnegie Mellon University CS CMU-CS-94-150, May 1994.
- Nardi, B. A. (1993). *A Small Matter of Programming*. Cambridge, MA: MIT Press.
- Nass, C., J. Steuer, et al. (1993). "Anthropomorphism, Agency, & Ethopoeia: Computers as Social Actors", Proceedings of INTERCHI'93, Amsterdam.
- Nelson, T. H. (1974). *Computer Lib/Dream Machines* self-published.
- Nilsson, N. J. (1994). "Teleo-Reactive Programs for Agent Control." *Journal of Artificial Intelligence Research* 1, 139-158.
- Norvig, P. (1992). *Paradigms of Artificial Intelligence: Case Studies in Common Lisp*: Morgan Kaufmann.
- Oren, T., G. Salomon, et al. (1990). Guides: Characterizing the Interface. In *The Art of Human-Computer Interface*, edited by B. Laurel. Reading, Massachusetts: Addison-Wesley.
- Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.
- Papert, S. (1991). Situating Constructionism. In *Constructionism*, edited by I. Harel and S. Papert. Norwood, NJ: Ablex.
- Papert, S. (1993). *The Children's Machine: Rethinking School in the Age of the Computer*. New York: Basic Books.
- Penrose, R. (1989). *The Emperor's New Mind*. Oxford: Oxford University Press.
- Piaget, J. (1929). *The Child's Conception of the World*. London: Routledge and Kegan Paul.
- Piaget, J. (1970). *Genetic Epistemology*. New York: Columbia University Press.
- Pimm, D. (1987). *Speaking Mathematically: Communication in Mathematics Classrooms*. London: Routledge & Kegan Paul.
- Premack, D. (1990). "The infant's theory of self-propelled objects." *Cognition* 36, 1-16.
- Reddy, D. R., L. D. Erman, et al. (1973). "A model and a system for machine recognition of speech." *IEEE Transactions on Audio and Electroacoustics* 21, 229-238.

- Reddy, M. J. (1993). The conduit metaphor: A case of frame conflict in our language about language. In *Metaphor and Thought*, edited by A. Ortony. Cambridge: Cambridge University Press.
- Repenning, A. (1993). Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments, PhD thesis, University of Colorado.
- Resnick, M. (1988). MultiLogo: A Study of Children and Concurrent Programming, MS thesis, MIT.
- Resnick, M. (1992). Beyond the Centralized Mindset: Explorations in Massively -Parallel Microworlds, PhD thesis, MIT.
- Resnick, M. and F. Martin (1991). Children and Artificial Life. In *Constructionism*, edited by I. Harel and S. Papert. Norwood NJ: Ablex.
- Saraswat, V. (1993). *Concurrent Constraint Programming*. Cambridge, MA: MIT Press.
- Sayeki, Y. (1989). Anthropomorphic Epistemology. University of Tokyo.
- Schaefer, R. (1980). Narration in the Psychoanalytic Dialogue. In *On Narrative*, edited by W. J. T. Mitchell. Chicago: University of Chicago Press.
- Schank, R. C. (1975). *Conceptual Information Processing*. Amsterdam: North-Holland.
- Schoppers, M. J. (1987). "Universal plans for reactive robots in unpredictable environments", Proceedings of Tenth International Conference of Artificial Intelligence, Milan.
- Searle, J. R. (1980). "Minds, Brains, and Programs." *The Behavioral and Brain Sciences* 3, 417-424, 450-457.
- Shneiderman, B. (1983). "Direct manipulation: A step beyond programming languages." *IEEE Computer* 16(8), 57-69.
- Shneiderman, B. (1992). *Designing the User Interface: Strategies for Effective Human - Computer Interaction*. Reading, MA: Addison-Wesley.
- Shoham, Y. (1993). "Agent Oriented Programming." *Artificial Intelligence* 60(1), 51-92.
- Siskind, J. M. and D. A. McAllester (1993). "Nondeterministic Lisp as a Substrate for Constraint Logic Programming", Proceedings of AAAI-93.
- Sloane, B., D. Levitt, et al. (1986). Hookup!, Hip Software. Software.
- Smith, B. C. and C. Hewitt (1975). A PLASMA Primer. MIT Artificial Intelligence Laboratory Working Paper 92, October 1975.
- Smith, R. (1987). "Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic", Proceedings of SIGCHI+GI'87: Human Factors in Computing Systems, Toronto.
- Solomon, C. (1986). *Computer Environments for Children: A Reflection on Theories of Learning and Education*. Cambridge, Massachusetts: MIT Press.
- Solomon, C. J. (1976). Teaching the Computer to Add: An Example of Problem-Solving in an Anthropomorphic Computer Culture. MIT Artificial Intelligence Lab AI Memo 376, December 1976.

- Stallman, R. M. (1981). EMACS: The Extensible, Customizable Display Editor. Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 519a.
- Steele, G. L. (1976). Lambda: The Ultimate Declarative. MIT AI Lab AI Memo 379, November 1976.
- Steele, G. L. (1980). The Definition and Implementation of a Computer Programming Language Based on Constraints, PhD thesis, MIT.
- Steele, G. L. (1990). *Common Lisp: The Language (2nd Edition)*. Digital Press.
- Steele, G. L. and G. J. Sussman (1976). Lambda: The Ultimate Imperative. MIT AI Lab AI Memo 353, March 10, 1976.
- Stewart, J. A. (1982). Object Motion and the Perception of Animacy, PhD thesis, University of Pennsylvania.
- Sutherland, I. (1963). Sketchpad: A Man-machine Graphical Communications System, PhD thesis, MIT.
- Sweetser, E. (1990). *From Etymology to Pragmatics: Metaphorical and cultural aspects of semantic structure*. Cambridge University Press.
- Tanaka, Y. (1993). "IntelligentPad." *Japan Computer Quarterly*(92).
- Thórisson, K. R. (1996). *ToonFace*: A System for Creating and Animating Interactive Cartoon Faces. MIT Media Laboratory Learning and Common Sense Section Technical Report 96-01.
- Tinbergen, N. (1951). *The Study of Instinct*. Oxford: Oxford University Press.
- Travers, M. (1988). Agar: An Animal Construction Kit, MS thesis, MIT Media Laboratory.
- Travers, M. (1988). Animal Construction Kits. In *Artificial Life: SFI Series in the Sciences of Complexity*, edited by C. Langton. Reading, MA: Addison-Wesley.
- Travers, M. and M. Davis (1993). "Programming with Characters", Proceedings of International Workshop on Intelligent User Interfaces, Orlando, Florida.
- Turing, A. M. (1936). "On computable numbers, with an application to the *Entscheidungsproblem*." *Proc. London Math. Soc., Ser 2* 42, 2330-265.
- Turkle, S. (1984). *The Second Self: Computers and the Human Spirit*. New York: Simon & Schuster.
- Turkle, S. (1991). Romantic reactions: paradoxical responses to the computer presence. In *Boundaries of Humanity: Humans, Animals, Machines*, edited by J. Sheehan and M. Sosna. Berkeley, CA: University of California Press.
- Turkle, S. and S. Papert (1991). Epistemological Pluralism and the Revaluation of the Concrete. In *Constructionism*, edited by I. Harel and S. Papert. Norwood NJ: Ablex.
- Turner, M. (1991). *Reading Minds: The Study of English in the Age of Cognitive Science*. Princeton, NJ: Princeton University Press.
- Tyrrell, T. (1993). Computational Mechanisms for Action Selection, PhD thesis, University of Edinburgh.

Ullman, S. (1984). "Visual Routines." *Cognition* 18, 97-159.

Ungar, D. and R. B. Smith (1987). "Self: The Power of Simplicity", Proceedings of OOPSLA '87, Orlando, Florida.

Winograd, T. (1991). Thinking Machines: Can There Be? Are We? In *Boundaries of Humanity: Humans, Animals, Machines*, edited by J. Sheehan and M. Sosna. Berkeley, CA: University of California Press.

Winston, P. (1982). Learning by Augmenting Rules and Accumulating Censors. MIT Artificial Intelligence Laboratory Report AIM-678.

Winston, P. H. (1992). *Artificial Intelligence*. Reading, MA: Addison Wesley.