MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Technical Report 1493 June, 1994

SodaBot: A Software Agent Environment and Construction System

Michael H. Coen
mhcoen@ai.mit.edu

This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

This material is based upon work supported by the National Science Foundation under National Science Foundation Young Investigator Award Grant No. IRI–9357761. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014–91–J–4038.

SodaBot: A Software Agent Environment and Construction System

by

Michael H. Coen

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis presents *SodaBot*, a general-purpose software agent user-environment and construction system. Its primary component is the *basic software agent* — a computational framework for building agents which is essentially an *agent operating system.* We also present a new language for programming the basic software agent whose primitives are designed around human-level descriptions of agent activity. Via this programming language, *users can easily implement a wide-range of typical software agent applications*, e.g. personal on-line assistants and meeting scheduling agents. The SodaBot system has been implemented and tested, and its description comprises the bulk of this thesis.

Thesis Supervisor: Lynn Andrea Stein
Title: Class of 1957 Assistant Professor of Computer Science

Acknowledgments

Yikes! It's erev Shabbos and I have to run and turn my thesis in! So, who has time to think about everyone to thank? Well, here's a list of those who come to mind in the 3 minutes I have to write this!

Professor Lynn Stein is a real mench, i.e. a good person. Where does she get the energy to provide so much support to so many students?! Needless to say, I'm quite happy to have her as my advisor.

This thesis was supposed to be about philosophical issues in knowledge representation, but Bart Selman and Henry Kautz got me interested in software agents last summer at AT&T Bell Labs. Will I ever get back to reading Wittgenstein? Chris Ramming provided thoughtful comments to earlier drafts of this thesis. Steven Ketchpel provided much comraderie while we worked at AT&T.

Robyn Kozierok tested much of the SodaBot system and provided voluminous bug reports.

My family always provides much love and support. I dedicate this thesis to my grandmother, Dora Estrin.

My friends have all helped in their own ways, some by getting me to work and some by getting me to stop. In particular, I thank (in lexicographic order): Andy, Debbie, Jeff, Sarah, Stacy, Upendra, Ye, and Yuri.

Contents

List of Figures

Chapter 1

Introduction

This thesis is about creating *software agents.* We argue that software agents should be written using a vocabulary not provided by traditional programming languages — *it should be possible to create agents solely by specifying their abstract behavior.*

Motivated by this position, we introduce *SodaBot*, a general-purpose software agent user-environment and construction system. Its primary component is the *basic software agent* — a computational framework for building agents which is essentially an *agent operating system.* We also present a new language for programming the basic software agent whose primitives are designed around human-level descriptions of agent activity. Via this programming language, *users can easily implement a wide-range of typical software agent applications*, e.g. personal on-line assistants and meeting scheduling agents. The SodaBot system has been implemented and tested, and its description comprises the bulk of this thesis.

This introduction is divided into the following sections:

(1.1) What is a software agent in the first place?

(1.2) What research problems motivated this work?

(1.3) What is the SodaBot system and how does it solve these problems?

(1.4) A reader's guide to the remainder of the thesis

1.1 Software Agent?

The beginning is always a good place to start, so what exactly is a software agent? The answer depends on whom you ask and can vary quite widely. (See chapter 5 for some typical responses.) For the purposes of this thesis, however, *we will focus exclusively on creating two specific types of software agents*: (1) *personal assistants* and (2) *application agents*.[1] First, we examine some typical high-level characteristics these two groups have in common, and then we look at several representative agents from each.

Agents in the Abstract

Agents are autonomous and temporally continuous.[2] Agents can act in behalf of particular people, i.e. they can take actions which appropriately represent the interests of others; therefore, agents must also be robust and capable of securely handling private information. Agents tend to be highly interactive — they spend much of their time communicating with other agents and human beings. Agents are active participants in their computational universe, i.e., they react to and cause changes in overall system state.

Agents in the Concrete

Agents are practical and helpful. We are particularly interested in the construction of software agents that automate simple on-line, repetitive and time-consuming tasks. Although we are interested in using software agents as a "testbed" for other areas in core AI ([Etzioni, 1993]), we must keep in mind that "one of the most challenging aspects of agent design is to define specific tasks that are both feasible using current technology and are truly useful..."([Kautz *et al.*, 1994])

[1] Although, much of our discussion applies to other types of agents as well.

[2] In this particular sense, they are similar to Unix daemon processes, but agents are generally associated with particular people or high-level applications.

We focus here on building the following two types of software agents:

1. *personal on-line assistants*: These agents generally belong to particular people and act like simple electronic secretaries. They do things such as:
   a) automatically respond to requests to schedule meetings by consulting their owner's private schedule;
   b) keep track of their owner's whereabouts and provide this information on request;
   c) contact their owner appropriately based on her location, e.g. via displaying a window on her workstation, sending a fax, or even making a phone call (SodaBot can't telephone yet, but see [Kautz *et al.*, 1994].)
   d) filter and sort incoming e-mail and faxes based on their owner's preferences (which may be provided explicitly or someday learned from observed behavior.).

2. *application agents*: These agents coordinate the transfer and processing of information among people and other agents. Application agents include:
   a) *Time schedulers* which schedule group or individual meetings among a set of people by negotiating among their personal agents to maximize some "convenience" measure.[Maes and Kozierok, 1993, Dent *et al.*, 1992, Kautz *et al.*, 1994].
   b) *Text processing systems* which allow complex processing of documents involving many people at different sites.
   c) *Receptionist agents* which accept requests and determine their appropriate destinations by interacting with other agents (and perhaps people as well).[3]

It is important to remember that software agents are simply computer programs, like expert systems, text editors, etc. Sometimes, especially given the anthropomor-

[3]This is work in progress with Randy Davis and Howie Shrobe.

phic autonomy of agents, it is easy to lose track of this and disconnect expectations from reality. However, one must keep in mind that what other programs cannot yet do, e.g. converse in English, agents cannot yet do either. Nevertheless, throughout this thesis, we will treat software agents as a unique class of computational entities — *agents are paradigmatically distinct from other types of computer software.* This outlook will direct how we intend for agents to be used and what types of tasks we expect agents to perform.

1.2 The Problem

Much of the work done in the area of software agents can be placed into one of two categories: (1) highly theoretical treatment of agents' intentions and capabilities ([Shoham, 1993, Doyle *et al.*, 1991, Etzioni *et al.*, 1992a]); (2) applied construction of specific agents ([Etzioni and Segal, 1992, Maes and Kozierok, 1990, Vere and Bickmore, 1990, Dent *et al.*, 1992, Kautz *et al.*, 1993]). However, determining for what (and if) software agents are actually useful requires building many of them, and the agent construction process poses difficult technical challenges.

It is generally straightforward to specify an agent's abstract behavior, e.g. "I want the agent to ask some user a question; it should remember her response in case the same question comes up again. Then, it should process the response by calling my preexisting C-language application on it and communicate the result to some other agent." However, traditional programming languages offer no primitive-level support for the typical kinds of high-level "online activities" in which agents engage, e.g. graphically obtaining structured, typed information from a user or communicating reliably with other agents.

Building agents generally involves a multi-layered approach. First, it requires a great deal of specific "system-hacking," e.g., of esoteric system software, networking protocols, windowing systems. (See [Kautz *et al.*, 1994] for a discussion of the difficulties involved in handling the most basic of agent functions, e.g., reading e-mail reliably.) Second, agent construction frequently involves mid-level computational issues, e.g. having agents handle several events simultaneously, provide reliable transactions, or handle errors automatically. Finally, agents (usually) *do* something with their computational foundation. For example, researchers in artificial intelligence may want to implement agent-based schedulers or knowledge representation systems.

Each of these layers can require a substantial amount of independent implementation and debugging time. So much so, in fact, that very few software agents have yet been built. And although the need to simplify agent construction is obvious, there are as yet few systems designed to assist the agent creator.

Additionally, it can be difficult to distribute new agents, i.e., to introduce them to the world and let other people use them. Agents tend to be site-specific in intricate ways and disconnecting them from their local dependencies can be technically involved. Furthermore, an agent which has been "disconnected" from its birth-site can also be quite challenging to install.

Finally, in particular with software that handles sensitive information and may even represent the user in interactions with other people, a person needs an enormous amount of confidence that the software will operate correctly. For example, who is likely to ftp and install a random program capable of autonomously sending e-mail in their name?

Thus, we see three problems with building software agents today:

1. They are technically challenging to write in traditional programming languages and operating systems.

2. They are difficult to distribute because they may have site-specific dependencies; for the same reason, they can be difficult to install.

3. People may be uncomfortable with the amount of responsibility given to an unknown (and possibly buggy) agent.

1.3 Towards a solution

This thesis presents SodaBot, a general-purpose software agent user-environment and construction system. In this section, we introduce the SodaBot system and outline how it addresses the three aforementioned problems.

The four primary components of SodaBot discussed below are:

(1.3.1) The basic software agent

(1.3.1) The graphical user interface

(1.3.2) The SodaBot agent programming language

(1.3.3) Automatic distribution of application agents

1.3.1 The SodaBot Software Agent Paradigm

In SodaBot, each user (or owner) is given a personal *basic software agent* (BSA) which typically runs in the background on her home workstation.[4] The BSA is an agent operating system — think of it as the "Unix of the software agent world." By this, we mean that it is a generic (in the sense of universal) computational framework for implementing and running specific agent applications. The BSA is programmed in the SodaBot *agent programming language* (SodaBotL).[5] As a quick sanity check, see if the following (rough) analogy makes sense:

SodaBotL is to *SodaBot* the way *C++* is to *Unix.*

A BSA runs SodaBotL programs provided both by its owner and by other people. It implements a time-sharing scheduling algorithm, so only one BSA needs to be running to simultaneously execute several agent applications for a particular user — see figure 1-1. The BSA runs an agent until it needs to wait for something, e.g.

[4]The BSA can alternately be configured in several other ways. See section 2.1

[5]Pronounced "Soda-Bottle."

user-input or communication from another agent. At this point, the BSA can put this agent to sleep and schedule another one in its place.

The user and her BSA interact through the SodaBot graphical user interface (GUI). Figure 1-2 shows the GUI of a BSA running on a Sun Sparc workstation. In this setup, the GUI occupies one of several virtual screens available to the user. (However, it can open windows on whichever virtual screen the user has active.) The central window in the display contains the BSA's main interface; it allows the user to monitor and control the BSA's activity — including reviewing and limiting its access to system resources. The top of the screen displays the SodaBot *agent editor* which allows the user to create, compile, and install agent applications. The editor can also be used to inspect incoming agent applications provided by other people. Finally, the bottom of the screen contains several windows opened by various software agents running on the BSA.

The GUI was designed to provide the user with a sense of control over her BSA; she can enable or disable various capabilities — e.g. having it modifying her e-mail file — depending on her confidence in it (or lack thereof). [Kautz *et al.*, 1994] makes clear that people are uncomfortable delegating absolute authority to software agents. By giving the user a large degree of control over agent activity, we hope to help assuage fears that the BSA will do something inappropriate or destructive.

The BSA has a novel architecture which allows it to perform a variety of complex agent-oriented tasks such as: reliably handle e-mail; graphically and textually interact with users; handle multiple concurrent events; interact in site-specific ways with its computational environment (e.g. run other system software, speak with a fax machine, etc.).[6] Typically, an individual software agent application must perform a number of these activities. However, robustly implementing such tasks typically requires a large programming effort and much esoteric system knowledge. Therefore,

[6]Several other features of the BSA will only make sense after we have introduced more of the system and are therefore discussed below.

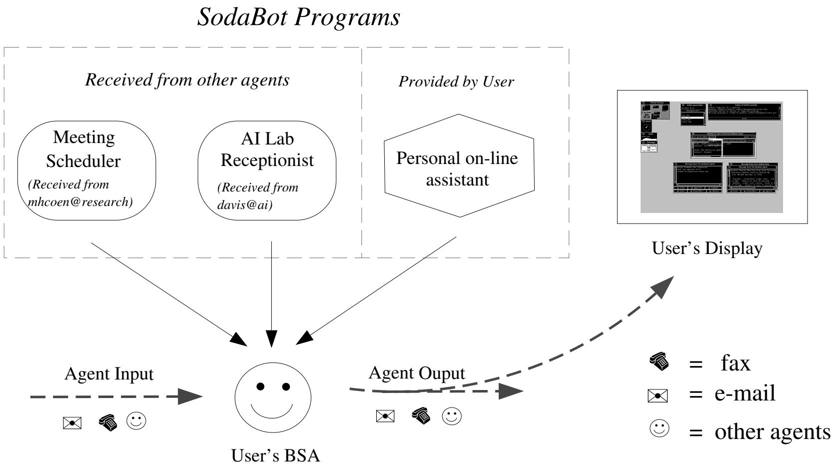

Figure 1-1: The Basic Software Agent

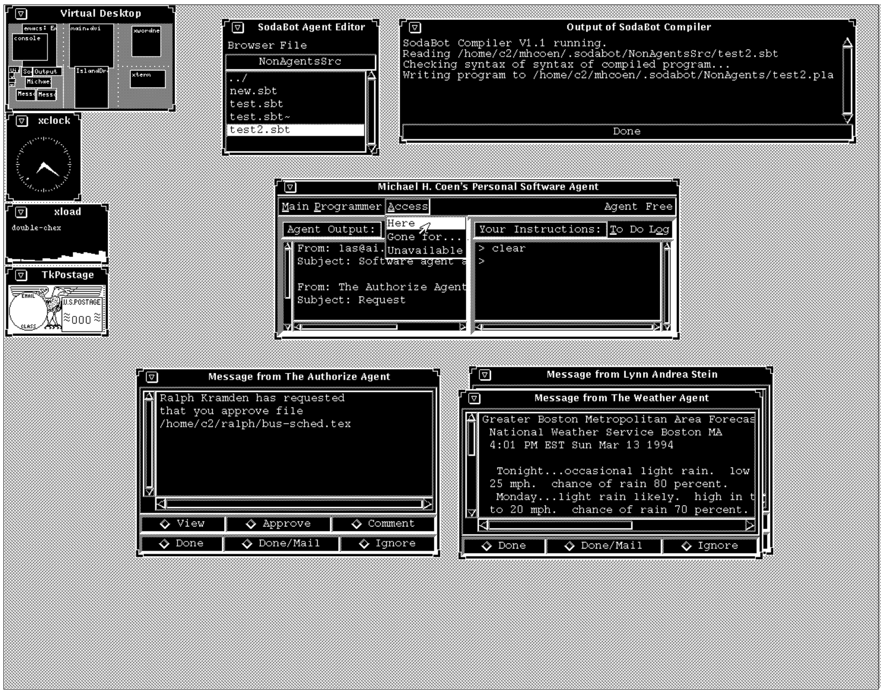

Figure 1-2: The BSA running on a Unix workstation

the BSA comes with them "built-in," and the SodaBot agent programming language offers high-level primitives through which they can be accessed.

What does the BSA buy us? It essentially *disconnects* agent programs from the specific computational environment in which they run. They no longer need to be "hard-coded" with specific parameters for particular activities, e.g. they don't require special knowledge of a host's mailer set-up. The problem of configuring many software agents is reduced to the problem of configuring a single agent, the BSA. Thus, for example, SodaBot agent programs written at MIT could be run without modification on BSAs at AT&T Bell Labs.[7] Furthermore, when a Macintosh or PC-based BSA is created, agent programs running on Unix-based BSAs can be run directly on Mac and PC-based BSAs.[8] Also, the BSA takes advantage of empirical knowledge we have gained installing various software agents on several Unix configurations, and it attempts to automate its own installation.

The BSA paradigm also assists in the development of user confidence in agent applications, because the BSA provides the only interface to critical system components. For example, if the BSA knows how to provide event time-outs or how to communicate correctly over TCP/IP, application agents running on the BSA do also.[9]

Finally, there is a simple computational efficiency gain by having only a single BSA image in memory rather than several independent agent programs. The atomicity of agent execution enforced by the time-sharing model can also simplify interaction among several separate agent applications.

[7] In fact, BSAs automatically distribute sections of new agent programs. Therefore, this property is essential for providing guarantees of correct behavior.

[8] This assumes, of course, that they are not Unix dependent because a non-portable SodaBotL *System* call. However, even in this case, an agent can be divided into various sections, some of which run in Mac and PC-based environments and others which run exclusively in Unix environments. See section 3.1

[9] Of course, simply knowing how to use something doesn't provide any guarantee of the appropriateness of a particular use. We discuss this topic in greater depth in section 3.4.

1.3.2 The Software Agent Programming Language

The SodaBot agent programming language (SodaBotL) offers high-level primitives and control-structures designed around human-level descriptions of agent activity. SodaBotL abstracts out the low-level details of agent implementation. In a typical Unix environment, for example, SodaBotL frees agent creators from the bother of dealing with system calls, mail servers, sockets, and X-windows. It is therefore much easier to have an agent:

1. Ask a question:

   Ask {prompt "Time"} "When are you returning?";

2. Display a message:

   Display {prompt "Read it?"; Choices(yes, no)} "Mail from $sender: $subject";

3. Contact another agent:

   Contact Agent <Receptionist; querier> {users: $inferred}
   "Do you know who in the AI Lab is responsible for $topic?"

4. Handle time:

   Wait until Tuesday before $date: {
   Display "Reminder, you have an appointment with $person on $date";}

The SodaBot language design focuses specifically on building two kinds of software agents: (1) *personal assistants* – Each user has a private SodaBot basic agent which she can customize to act as a simple electronic secretary. A user can program her agent in SodaBot to do things such as: filter incoming e-mail; notify her about particular events (e.g. someone has returned to his office); or automatically handle incoming requests. Users and other agents can also contact someone's personal assistant much in the same spirit one would a human secretary.[10] (2) *application agents* – Users can create agents that specifically provide various services. Application agents are also

[10] We have also been considering the development of personal assistants for mobile robots which would provide on-line remote access to them.

Agent{Pollster}:
Get Response {prompt "And what's your opinion?";
timeout in 10 minutes}
$message_body;
Reply with $response;

Figure 1-3: A simple agent for gathering opinions.

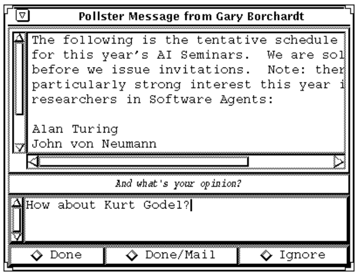

Figure 1-4: The *Pollster* window.

called *SodaBots*. For example, figure 1-3 has a simple *Pollster* agent. This agent allows us to quickly solicit opinions from a group of people by directing a *Pollster* message to their personal agents.

1.3.3 Automatic Agent Distribution

A SodaBot *environment* is composed of a society of basic software agents which are connected via the Internet (and/or a local area network). Within an environment, *new application agents are automatically distributed — as SodaBotL programs — among its constituent BSAs.* For example, suppose Robyn creates a group meeting scheduler. When she schedules the first meeting, the basic software agents owned by the people in her group will automatically request their required sections of this application agent from her BSA. Note that SodaBot does not enforce a client/server model of agent interaction. Sections of an agent's program are distributed as the need arises, and the "server" role in an agent might dynamically rotate among the basic agents or might not exist at all.

Additionally, SodaBotL allows for context-dependent interpretation of its primitives based on their run-time environment. Primitives are requests to perform an action that do not actually specify how the action should be done.[11] For example, a request in an agent's program to display a message can be satisfied by one (or more) of the following, depending on the owner's location: (1) put up a window on the user's display; (2) e-mail the message to the user; or (3) fax it to her, etc.

Finally, because SodaBotL agent programs can be much shorter than their counterparts written in other systems, e.g. C on a Unix platform, they are that much easier to debug and inspect for security threats. For example, while it may not be possible for a user to peruse a random agent program she ftp'ed from somewhere on the Internet, it is quite feasible that she can inspect in detail a SodaBotL agent received by her BSA. (Security issues are discussed in section 3.4)

1.4 Reader's Guide

The remainder of this thesis discusses the SodaBot system and how it simplifies the construction of software agents. We do not propose that SodaBot is a universal agent construction tool or that its level of support is sufficient for all or even many applications. However, central to the design of SodaBot is that all of its main components are separate, replaceable modules. If some capability is not provided or if some feature is inappropriate for a particular site, that part can be added or replaced while treating the rest of the system as a black-box abstraction. Also, particular SodaBot modules can be incorporated into other programs. For example, an application which needs

[11] [Kautz *et al.*, 1994] describes this as "intension" vs. "extension," (i.e. "connotation" vs. "denotation"). It is not clear this terminology (drawn from linguistics) accurately describes the intended phenomenon. At least with respect to SodaBot, context-dependency is simply with respect to the medium for conveying information; the information explicitly must be made available, but the BSA may select precisely how.

to receive and filter e-mail can use just these parts of the SodaBot system. Finally, SodaBot can be used to get fast, working prototypes of software agents, even if it is intended that the final application be completely written, say, in Lisp or C.

To summarize, SodaBot provides the following capabilities:

1. simple, fast construction of application agents and personal assistants
2. support for complex human/agent and agent/agent communication
3. automated distribution of new agents
4. a wide-range of default behaviors for "typical" situations

Chapter 2 introduces the SodaBot software agent paradigm, i.e., essentially, what we mean when we talk about an agent. We introduce the automated distribution of application agents in chapter 3. The SodaBot agent programming language is presented in chapter 4: first, by way of example, and then through a more formal specification. How SodaBot relates to other current software agent research is examined in chapter 5. Finally, we evaluate the system in chapter 6.

1.4.1 SodaBot agents written in SodaBotL for a BSA?

The reader may be quite pleased that this document contains no first-order predicate calculus. Even so, there is the risk that some of the new terms defined here may cause confusion. To dispel any perplexity, we offer the following guide:

- *SodaBot* is a software agent user-environment and construction system.
- *A SodaBot* is a software agent implemented in SodaBot.
- *SodaBotL* is the name of the programming language for building software agents in SodaBot.
- The *Basic Software Agent* (BSA) is the foundation users build upon to create *personal assistants* and *application agents.* It is a basic agent operating system.

- A *personal assistant* is a simple electronic secretary. It runs on a user's BSA.
- An *application agent* is an agent which performs a specific task. It runs among the BSA's that comprise a SodaBot *environment*.
- A SodaBot *environment* is composed of a group of basic software agents which communicate with each other (via E-mail, TCP/IP, etc.)
- An *owner* or *user* is the human being directly associated with a particular BSA.

Chapter 2

The Basic Software Agent Paradigm

This chapter details the computational foundation of the SodaBot system — the basic software agent. We first examine various configurations for running the BSA. We then detail the behavior and architecture of the BSA along with issues that arose during its implementation. Finally, we discuss the benefits provided by the BSA to software agent creators. Although this chapter primarily addresses Unix-specifics of the SodaBot system, presumably parallels exist in the PC world to much of this discussion.

This chapter is divided into the following sections:

(2.1) BSA Installation

(2.2) BSA Behavior and Architecture

(2.3) What's It Good For?

2.1 BSA Installation

SodaBot can be configured to run in one of several ways:

1. The BSA can run as a constant background job on some workstation — usually (although not necessarily) the user's home workstation. This way, it is always ready to respond to incoming requests, and if permitted, it can take actions without the user's explicit approval. This configuration does not tie the BSA to a particular display if the site permits "xhosting" to other machines; in this case, the user can simply notify her BSA of her current location. This is the default and simplest way to configure SodaBot.

2. The BSA can be run when the user logins and terminated when the user logs out. This option is appropriate if the user is uncomfortable leaving SodaBot running in her absence or if option (1) causes technical complications, e.g., "xhosting" is not permitted and she moves around frequently. This configuration disallows certain interactions, e.g., TCP/IP connections from other agents, when the user is not logged in.

3. The BSA can be started dynamically only when there is: incoming e-mail, an incoming fax, a TCP/IP connection from another agent, etc. Unlike with options (1) and (2), configuring SodaBot for this behavior can be quite difficult and setting up the system requires "user-wizardry." (It generally requires root access as well.)

For option (1), the user does nothing more than start her BSA by typing "SodaBot" at the Unix prompt. The first time it is run, SodaBot creates a directory hierarchy in "~/.sodabot/" that holds the user's agent files and personal configuration information, such as the name of her home display. The BSA also tries to determine site and organization-specific information such as the location of mailer files, system libraries, and the name of institution.[1] It may have to ask the user several questions during installation, but in the current implementation this process is not generally

[1]Institution name can be guessed from the IP address although the current table of known addresses is very small — just the MIT AI Lab and AT&T Bell Labs.

interactive. Option (2) merely requires that the user start option (1) and "quit" the BSA whenever she logs out. The BSA's starting and stopping can also be performed by her .login and .logout files respectively. The remainder of this thesis assumes the BSA has been installed with either the first or second option. While we have done much experimentation with option (3), the vast array of extant Unix configurations today makes it simply too difficult to automate installation, and the skills required for custom installation are beyond the capabilities of most users.[2] More technical details regarding agent configuration for options (1) and (2) are discussed in section 2.2.2.

2.2 The Basic Software Agent

When it is not running an application agent, the BSA spends most of its time sleeping. However, when idle, it periodically wakes up (e.g. every 5 seconds) and checks for (1) incoming e-mail; (2) user activity in the GUI; (3) a record "waking up" in one of the system databases; (4) completion of a *system* command invoked by the agent or its owner; or (5) contact from another agent.[3] In order to insure responsiveness during user interaction, the GUI runs as a separate process (which responds to various X-windows events).

Figure 2-1 shows the general SodaBot system architecture. The BSA is connected to available system resources, and all application agents access these resources through the BSA. Thus, for example, the BSA is solely responsible for displaying windows on the user's display and processing her e-mail.

Because application agents do not directly access system resources, the BSA is free

[2]If the user wants to try option (3), there is a seperate installation program which can automatically generate the (generally) necessary setuid-to-root "wrappers." However, the user will have to connect these wrappers to the system files manually. Also because a system's servers rarely honor setuid-to-root flags from remote clients, the installation program must be run directly on each server and the wrappers must be stored on one of the server's local disks.

[3]The BSA's polling behavior when idle (i.e. waking up every few seconds) is no more CPU intensive than that of other popular Unix applications, such as *xbiff*, and it would seem to be a good deal less of a burden on system resources than the ubiquitous *xload*. The length of the sleep-wake cycle is adjustable by the user.

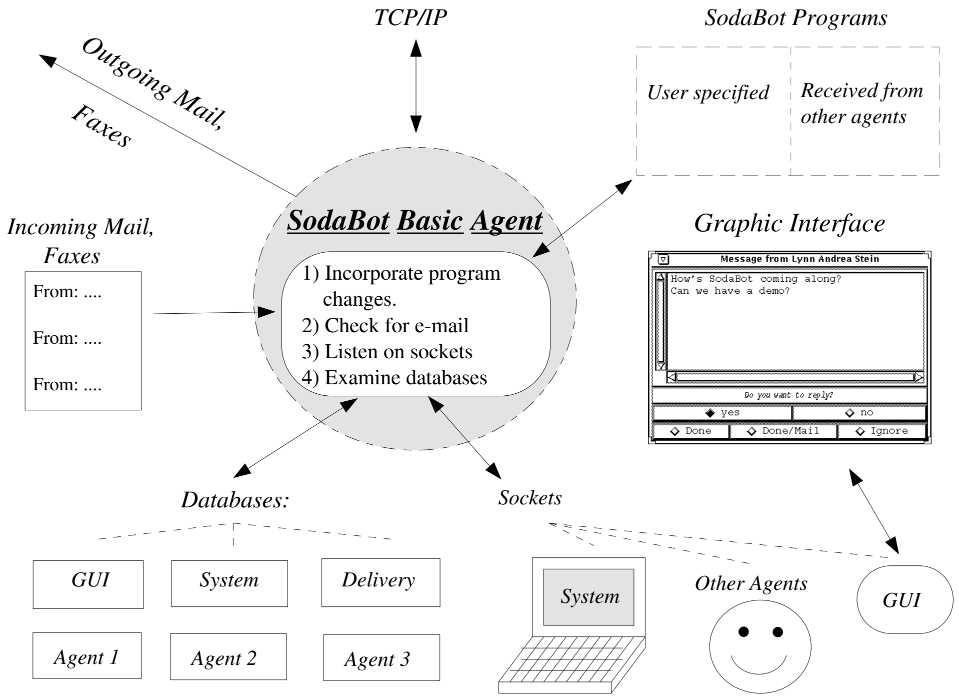

Figure 2-1: The SodaBot basic software agent architecture

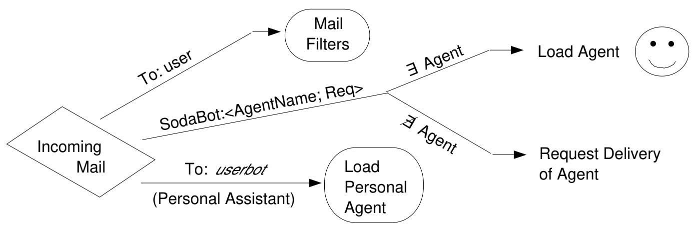

Figure 2-2: Mail handling in the BSA

at run-time to reinterpret their access requests. For example, suppose an application agent wants to display a message to the user, and the BSA knows that she is not at her workstation but is reachable by fax. It can alternatively fax the message to her in addition to (or instead of) displaying the message on her screen. This is dealt with in more detail in section 1.3.1 in the chapter covering SodaBotL.

Work on a system TCP/IP connection for agent-to-agent communication is just beginning. It will help alleviate the bottleneck caused by relatively slow e-mail delivery and it will allow new or cautious users to configure the BSA so that application agents can run without accessing their e-mail. When faced with network "fire-walls" or with temporary network failures, however, the BSA will be able to resort to using e-mail for inter-agent communication.[4] Also, note that how messages are sent between application agents is actually invisible to those agents; the SodaBotL *Contact Agent* primitive means send the message without specifying the transfer medium.

The connection to the system mailer is the most complicated system resource link in SodaBot. (See section 2.2.2) In the standard configuration, the BSA and its owner share the same mailbox. Incoming e-mail can be intended for (see figure 2-2):

1. The BSA's owner — In this case, it loads and runs her specified mail filters over each new message. (Figure 2-3 contains a sample filter.)

[4]Communication via e-mail can also be quite useful for debugging agent applications because it is readable by a person, unlike TCP/IP data.

```
Mail filter:
Received mail {from: /las/; subject: /funding/}:
Display{prompt "Read now?"; choices(yes, no, check)}
    "Important mail from Lynn!: $subject";
if      (choice(yes))      {Display $body;}
elsif   (choice(check))    {System "finger $address";
                            Display $result;}
```

Figure 2-3: A SodaBot mail filter

2. A particular application agent — Mail can be directed to application agents by including a special *SodaBot* header in the message, such as

   To: mhcoen@ai.mit.edu

   SodaBot: <Pollster>

   in which case it is directed to the named agent. The specification, particularly in headers generated by other agents, can also include a particular "section" of the agent and a version number, e.g. "SodaBot: <Scheduler; request_meeting; v1.0>"

   Agents can also be contacted via e-mail aliases. For example, the address "ScheduleBot@ai.mit.edu" can resolve to "mhcoen@ai.mit.edu." In this case, the BSA is given a list of *(alias, agent)* pairs which it uses to resolve the contents of the "To:" header into an application agent.

3. The owner's personal assistant — By default, the personal assistant is named "user_name*bot*@address," e.g. "mhcoenbot@ai.mit.edu."

2.2.1 The Basic Software Agent Internals

The BSA can be viewed as the kernel of a time-sharing agent operating system. SodaBotL programs are compiled into the BSA's native operating language which is directly interpreted by the BSA when it runs an agent. The compiled agent programs are divided into multiple sections which can be stopped and restarted by the BSA's agent scheduler.[5]

The agent scheduler allows the single-threaded BSA to respond to variety of concurrent system activity. It is not reasonable for the BSA to spend large amounts of time waiting for an some event, e.g. user input or completion of an invoked application (e.g. a constraint-propagation package for an agent-based planner), at the expense of others. For example, the mail filter in figure 2-3 displays a message to the user and then *it needs to wait* for the user's response before evaluating the subsequent if-statements. SodaBot handles this required delay by putting the application agent or mail-filter to sleep until the user provides the requested input (or the message times out). Sleeping agents are stored in one of three *sleeping agent* databases where they wait for particular events to occur or for time limits to pass.[6]

When a BSA receives input from the user interface, it checks the *GUI* database for a record waiting for this input. This record would contain: (1) the name of the sleeping agent; (2) the position in this agent at which to resume processing; and (3) the data the agent was running on, e.g. an e-mail message and the agent program's local state. If the appropriate record is found, its corresponding agent is then reloaded and continues running until completion or until the next expression which causes it to wait.

The BSA also has a database for dealing with the SodaBot "system" command, which can be used to access local software, e.g., the "finger" command, LaTeX, ghostscript, etc. The system command can take an arbitrary amount of time to finish, so the BSA creates a separate process to run the specified command and puts the agent to sleep in the *system* database. The BSA will continue with some other

[5] SodaBot is written in Perl, Extended TCL/TK, and C. This part of the system is implemented in Perl, a language which provides little support for this type of non-local program flow. In order to permit it, agent programs get divided into many individual procedures, each of which must be called in order to execute the agent. Thus, we can interrupt an agent's execution by pausing between subroutine invocations.

[6] Mail filters are actually run by the internal *mail filter* agent which can also be placed in one of the *sleeping agent* databases.

activity. When it hears that the command has completed (over a Unix-style socket), the BSA will reload the sleeping agent with the command's output stored in the *$result* variable.

The third database is for incoming messages which have been directed to an unknown application agent. When e-mail is sent to an agent that the BSA does not know about, the BSA requests that the specified agent be supplied either by the original sender or by a central agent depository. Until that agent arrives (as a SodaBotL program), the mail is held in the *delivery* database. Automatic distribution of agents is discussed in chapter 3.

Agents can specify maximum lengths of time they are willing to spend sleeping in a database. After this timeout expires, the agent is restarted at some specific error-recovery or expired-timeout point where it can take remedial or default action, such as re-sending a request or notifying its owner of the problem. Certain error-recovery protocols are built-in to SodaBot, such as recovery from failure to receive requested agent programs.

2.2.2 Agent Configuration

The user can configure the BSA based on both her preferences for its behavior (see section) and the amount of confidence she has in its correct operation. For example, once the BSA has the ability to communicate directly via TCP/IP, it will be capable of running without accessing the user's e-mail if she so desires; additionally, the BSA's ability to run local software can be greatly curtailed or eliminated. It seems quite essential, particularly because SodaBot is a new and experimental system, that we provide people with a minimally risky way of using it. Thus, the BSA's connections to specific system resources can be temporally or permanently disabled. As the user becomes more comfortable with the system, she can selectively re-enable features.

Users can also determine precisely how the BSA, if permitted, processes their e-mail. This is actually a rather complex issue, and there are many technical details that are not sufficiently interesting to document here. For example, there are (too) many ways to connect a BSA to the system's mailer. The simplest way is for the BSA and its owner to share the same mailbox file. In this case, we would like the BSA to quickly remove its messages from this file so the sharing is transparent to the user. However, in Unix, there are certain file locking problems on distributed file systems which could cause a program that writes to a mail spooler file to clobber incoming mail.[7] In response to this problem, SodaBot's default configuration does not write user-messages back to the spooler file after it removes mail intended for the BSA. Thus, the user needs to obtain her new messages from somewhere other than the system-mailer's spool file; therefore, she must explicitly provide a location for this new spool file, and inform her mail-reader of its existence.[8] It is also possible for the BSA to have its own e-mail address (as done in [Kautz *et al.*, 1994]), but on some systems this can require root access to set up.

2.3 The Benefit

The basic software agent provides preliminary solutions to the three problems outlined in section 1.2 with software agent construction. More importantly, it is a foundation for other components of the SodaBot system, e.g. SodaBotL, to more completely address these issues.

1. Software agents can be technically challenging to write in traditional programming languages and operating systems:

    - We discussed earlier the three coding-layers typically involved in software agent construction: (1) low level, e.g. networking; (2) mid level, e.g. error handling; and (3) high level, e.g. knowledge representation. The BSA

[7]The problem here is quite technical — it involves the distinction between lockf(3) and fcntl(2v) — and occurs only very rarely, but I have verified its existence with local GNU mail-wizards.

[8]If she prefers, the user can have her BSA write back to the mail-spooler file. The risks are no greater than if she did so herself using one of the more popular mail reading programs.

helps free the agent creator from the first two of these efforts so that she can concentrate on the third. It pushes system-specific aspects of agent creation beyond the abstraction barrier.

2. Software agents are difficult to distribute because they may have site-specific dependencies; for the same reason, they can be difficult to install.

   - The BSA disconnects application agents from the specific computational environment in which they run. Agent programs no longer need to be "hard-coded" with site-specific information.
   - This disconnection makes it easy to move agents around. They don't need to be disconnected from their "birth-site" or hooked-up to anything when being installed.
   - The problem of configuring many software agents is reduced to the problem of configuring a single agent, i.e., the BSA.
   - The BSA has knowledge of several standard Unix configurations and tries to automate its installation.

3. People may be uncomfortable with the amount of responsibility given to an unknown (and possibly buggy) agent.

   - The BSA allows the user to gradually establish confidence in its behavior and to selectively disable and enable access to specific system resources.
   - The agent-system disconnection allows the BSA to reinterpret at run-time requests for system resources from application agents. Users can therefore customize the behavior of application agents without actually modifying them.
   - The BSA provides a stable agent-framework over which the user can exert ultimate control.

Chapter 3

Distributed Agents in SodaBot

In SodaBot, an application agent generally doesn't run as a single program on a particular basic software agent. Rather, various sections of an agent are automatically distributed to and run on the BSAs which comprise a SodaBot *environment*. The agent's activity is manifested by the coordinated interaction of these program sections.

This chapter discusses how SodaBot agent programs are structured and how they are distributed. We are concerned here only with how SodaBotL programs are organized and how they move around the network, not with what they actually do (or "mean") once they arrive at their destination. It might seem a little odd to discuss how SodaBotL programs travel before saying what it is they actually do. However, because understanding agent distribution is fundamental to writing a SodaBot agent, we present it before detailing SodaBotL's semantics (in Chapter 4).

We also discuss several related issues, including SodaBot environment topology and security concerns inherent in any automated distribution/installation of software. Security against malicious adversaries was at best a peripheral consideration during the design of the current implementation of SodaBot. However, it is a very interesting research topic and one we hope to explore in the future.

3.1 The Anatomy of an Agent

A SodaBot *environment* is formed by a group of basic software agents which cooperatively implement an application agent. These agents may all be running on the same physical machine or they may be distributed across a network, each running (in theory) on a different operating system and platform. An individual BSA is most likely a member of several different environments, each expressing the unique connectivity required for a particular application agent.

SodaBotL programs are composed of (possibly overlapping) *groups*, where each group represents a different distribution of sections of the agent's program. *Groups specify what role the agent requesting software is going to play in the execution of this agent.* For example, the simplest group labellings might be "client" and "server;" one BSA might then contact another, "I need section X of agent Y," where section X is in the "client" group. Then all sections of agent Y in the "client" group would be sent out to the requesting agent; presumably, in its role as a "client," it will need access to the other sections of agent Y which also fall into this group. However, SodaBot doesn't enforce this model; agents can have more than two groups and individual BSAs can fall in more than one group. It may also not be computable a priori which BSAs will end up in which groups when the agent is running. As an agent gets added to more groups, it may have to issue additional requests to obtain the sections of the agent associated with them.

Figure 3-1 outlines the generic SodaBot application agent structure. An application agent is divided into various sections (numbering corresponds to that in the figure):

1. Global declarations — These are declarations which get distributed to every BSA running the agent.

2. The main agent — This section specifies what happens when the agent gets invoked. Notice there is no group expression specified here. Presumably, a

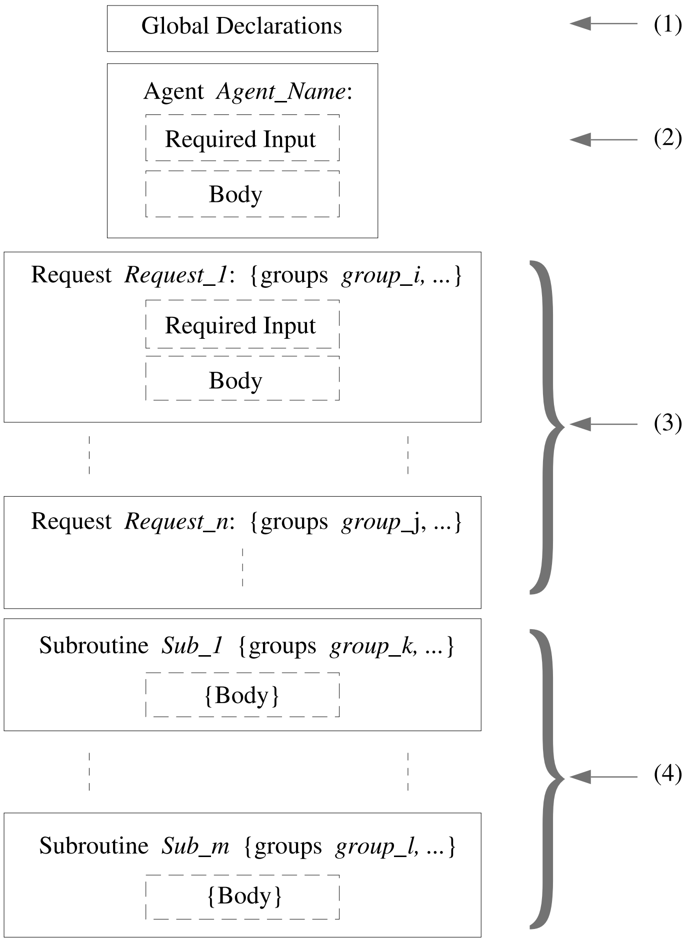

Figure 3-1: The generic SodaBot agent anatomy

BSA that requests this part of the agent needs all the other sections as well so that it can in turn distribute them. So, a BSA that requests the agent without specifying a group is sent the entire agent. The *Required input* specifies the format of the input which must be supplied to the agent. The *body* is a list of SodaBotL expressions.

In the current implementation of SodaBot, agents can only be invoked via e-mail.[1] For example, one might send the following structured message to the hypothetical *Authorize* agent (which will be presented more fully in section 4.1):

To: authorize@ai.mit.edu ------ *Person1: las* *Person1: brooks* *pathname: ~mhcoen/tr.ps*	*as input to*	**Agent** authorize: **Required input** { person1: *username person2: *username pathname: *pathname}

Note that the structured input is in the body of the message, not its header.

---

3. Request *Request_i* — Requests are atomic sections of agents which get distributed to and run on members of its environment; *essentially, an application agent's requests are simply what different BSAs can ask each other to do while running the agent; group membership simply determines which requests a BSA is allowed to issue.* Note that each request must specify to which group(s) it belongs. One BSA can issue a request to another BSA by sending e-mail to its owner such as:

   To: las@ai.mit.edu
   SodaBot: <authorize; certify>
   SodaBot-Parser: <l:1;s:0;e:35. l:2;s:18;e:13.>

   Michael H. Coen (mhcoen@ai.mit.edu) requests that you authorize
   the submission in ~mhcoen/tr.ps

[1] We hope to soon also make them accessible via the graphic user interface. In a straightforward way, we can generate a graphic input-window based on the *Required input* specification. Most likely it will look very much like the *form* input type in Mosaic.

The "SodaBot-Parser:" header is used by the receiving BSA to determine where SodaBotL variables have been substituted into the message body by the agent that sent it. For example, the first argument in this case (*$requester*) is on line 1 of the message, starting at position 0, and runs for 35 characters. Requests can also specify *Required input* which is used if the message was sent by a person rather than a BSA.

4. Subroutines *Sub_i* — Subroutines are called only by requests, not other subroutines. If they list no group membership, only the main agent has access to them. Otherwise, they are distributed with their respective groups the same way requests are.

3.2 A Sample Distribution

We now consider the simplest type of distribution — that between peers — with the *Pollster* agent introduced in section 1.3.2. Suppose Patrick creates (or updates) this agent and then sends e-mail directed to Gerry's *Pollster*.[2] (See figure 3-2.) Upon receipt of this message, Gerry's BSA checks whether it has a *Pollster* agent. In order that we have something to say here, let's assume that it doesn't.[3] Then, Gerry's BSA places the incoming *Pollster* message in its *delivery* database, which consists of messages awaiting the arrival of application agent software. It then requests that Patrick's BSA supply the *Pollster* agent program. When it arrives, Gerry's BSA compiles the Pollster SodaBotL program and then starts the *Pollster* on the messages in the *delivery* database that were awaiting its arrival.

Every BSA has two built-in agents called the *RequestAgent* and the *DeliveryAgent*.

[2]Software updates are distributed like new software and are indicated by providing a higher than current version number.

[3]Another issue that arises with agent distribution is developing a common namespace. Suppose Gerry's BSA did have a Pollster agent, but a different one than intended by Patrick. Currently, the only way to handle this problem is to specify not only the agent name, but also its flavor and version, such as "<*Pollster*; Patrick; v1>".

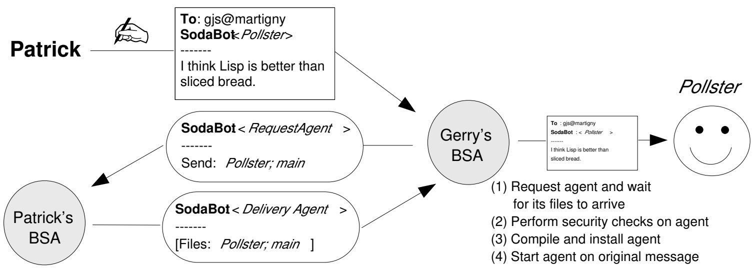

Figure 3-2: Distribution of the *Pollster* agent

When a *RequestAgent* receives a request to provide some section of an application agent, it tars, compresses, and mails the SodaBotL program corresponding to that section to the requester's *DeliveryAgent*. In turn, when a *DeliveryAgent* receives a previously requested agent, it unpacks the incoming program and uses the SodaBot compiler to install the agent after taking whatever security precautions (discussed below) that it has been configured for.

In this example, the *Pollster* agent is an atomic whole and requesting it is an all-or-nothing affair; it is not broken down into groups. However, "peer distribution" doesn't require this. What is important is that when Gerry's BSA asks for the Pollster agent, Patrick's BSA is ready to supply it. When the groups an agent contacts are a subset of its own, then we call the resulting transfer of software a "peer distribution," i.e. your peer can supply you with the software it is requesting that you run.

3.3 Network Topology

What happens when an agent's environment is more complex? Unlike like with the simple peer distribution above, sections of agent may contact other groups in the agent of which they are not members; therefore, how do they supply any requests for these groups' software?

Possible solutions include:

- If an agent doesn't have the software, presumably, one of its "ancestors" does. So, it should relay the request to its "parents" and pass on their responses.

  This can require $O((n/(n-g+1))^2)$ separate communications overall, where n is the number of BSA's involved and g is the number of groups in the agent. It can incur long communication delays, particularly if done by e-mail.

- If an agent contacts another whose software it doesn't have, it should include the the name of either: (1) its "parents"; or (2) the name it was given to contact for software (if one was provided). Thus, requests can be sent to them directly, and the agent is removed from any ensuing communication regarding obtaining the software.

  This can require $O(ng)$ separate communications overall; assuming unit cost for processing requests, this is perhaps the minimum we can expect.[4] Note that the communication patterns that invoke the upper bound here and in the case above are not necessarily bizarre or unlikely. However, *these are only start-up costs* — invoked only the first time the agent is run or when it is updated — so they may be quite tolerable.

- We set up a central agent depository which keeps all current agents in a library, available upon request.

  This is the solution we currently prefer for SodaBot. However, it is conceivable that some applications may not be freely distributable, due to secrecy, export limitations, software/patent licenses, etc. Thus, we might need to provide guarantees of limited access. We have not looked into this issue, but it would seem a secure protocol — perhaps via a kerberos ticketing scheme — might be developed to do this.

[4] The caveat is that a single BSA may receive a flood of requests for agent programs from all of its descendants.

- Do away with group distinctions and send out agents in their entirety. Then, everybody has everything they need.

  Note that application agents may contact completely different agents — e.g. the *Scheduler* can contact the *RoomCoordinator* — not merely other parts of the same agent. Should we require that every BSA possess every application agent? Also, the distribution limitation issues discussed above apply here as well — effectively ruling out this option.

3.4 What's the Risk?

There are genuine security concerns for this (and any other) method of automatic software distribution. In the introduction, we noted that no one is likely to ftp and install random software that could send e-mail in her name. All the more so, shouldn't a user be concerned about random software which can simply appear on her workstation without her involvement?

Currently, SodaBot can be configured with several simple options including:

- Don't compile received agents which use the *system* command.
- Ask for owner permission before compiling such agents. However, if the user has to read a program, what advantage does SodaBot provide? Simply that the programs are much shorter and easier to understand. We would argue that malicious behavior is therefore more easily detected.
- Insure that application agents only access files and execute commands in particular user-specified directories.

A user is free to limit the system access given to her BSA, thereby limiting the access available to all application agents. She can also require that her BSA (and therefore application agents as well) obtain explicit permission before taking certain actions, e.g. sending e-mail.

We hope to add more sophisticated security measures such as encrypting the distribution of agents to guarantee authenticity and integrity.

3.5 The Benefit

SodaBot's automatic distribution of application agents addresses the second and third of the problems outlined earlier with software agent construction:

(2) Software agents are difficult to distribute because they may have site-specific dependencies; for the same reason, they can be difficult to install.

- Agents in SodaBot distribute themselves over the Internet and/or local area networks. This distribution can even be across platforms and operating systems. Simply put, acquisition of new agents does not require that the user do anything.

(3) People may be uncomfortable with the amount of responsibility given to an unknown (and possibly buggy) agent.

- SodaBot simultaneously helps alleviate and further exacerbates this problem. The BSA is a trusted, much tested framework, and its behavior is under the ultimate authority of its owner. Regardless of this, however, automated software distribution would seem to get many people very nervous. Hopefully, additional efforts to protect against malicious adversaries will provide sufficient reassurance.

Chapter 4

Writing SodaBot Agents

This chapter presents SodaBotL — the SodaBot agent programming language. We described in chapter 3 how an application agent gets distributed in sections to the BSAs comprising its environment. Now, it's time to discuss what these programs actually mean — both to the people who write them and the BSAs that receive them.

By way of example, we first outline how to create a SodaBot application agent that involves interactions among several people. We then present a SodaBotL programming reference guide and discuss the benefits provided by SodaBotL to software agent creators. Note, Appendix B contains a SodaBotL BNF grammar.

4.1 Agents Describe Dialogs

Chapter 3 described how SodaBot application agents run across a network of interconnected BSAs. Generally, no single BSA runs an entire agent program; rather, each BSA falls into one or more groups and runs only those particular sections of an agent that correspond to its group membership.

Our model for writing agents is that agent execution corresponds to a series of dialogs among the BSAs that comprise the agent's environment. Thus, while running a particular agent, a BSA's group membership determines who it's allowed to speak with and what it's allowed to say in a series of conversations with people and other BSAs. We use "conversation" here somewhat freely; we consider the BSA's owner making a selection in the GUI, an incoming fax, or mail from another agent to be among the things that the BSA can listen to. In turn the BSA can process what it hears and then communicate it via the GUI, fax machine, TCP/IP, etc.

So, each *request* section of an agent's SodaBotL program expresses the agent's reaction to "hearing" a particular request while running on a BSA in its environment. In the framework of a conversation then, each request section of the program must specify:

1. What to listen for — This is the section's permissible input. It can consist of a required format for incoming e-mail and/or an input specification from the user through the GUI.
2. What to do with it — This specifies how to process what the agent "hears," including searching the input, textually manipulating it, saving it to a file, or feeding it to some external program.
3. How to continue the conversation — If it is not finished, the agent needs to "pass the buck," by issuing one or more requests to BSAs in its environment and thereby create continuing threads.

Thus, an agent writer outlines the shape of a dialog, i.e. what inputs are allowed in the dialog, how they should be transformed, and how to communicate them. Figure 4-1 presents a simple *authorize* agent for sanctioning document publication.[1] This agent was designed according to MIT AI Lab's procedure for authorizing the publication of memos and technical reports. Our experience designing other *ad hoc*

[1] The current running SodaBot compiler uses a slightly different, less readable syntax than the one presented here. We are working towards making the syntax given here the standard for SodaBotL and had initially hoped it would be ready by the publication date of this thesis. However, the new compiler is not finished. We are presenting only the new syntax in the body of this document. See Appendix A for details relevant to the current implementation.

```
$auth_dest = "publications@ai.mit.edu";

Agent authorize:
  Required input {
    person1: *username
    person2: *username
    pathname: *pathname}

  $requester = $from;
  Contact Agent <authorize; certify> {users: $person1, $person2}:
    "$requester requests that you authorize the submission in $pathname";

Request certify:{group signer}
  Display {prompt "Do you want to see the file now?"; choices(yes, no, view)}:
    "You have been requested to authorize $pathname by $requester";
  if choice(yes) {
    Load $pathname $contents;
    Display {prompt "Do you want to authorize this?"; choices(yes, no)}:
      $contents;}
    if choice(yes) {&grant_authorization}
  elsif choice(view) {
    System "ghostscript $pathname";
    Display {prompt "Do you want to?"; choices(yes, no)}:
      "You have been requested to authorize $pathname by $requester";
    if choice(yes) {&grant_authorization}}

Request grant:{group publications}
  Display:
    "Authorization received from $from for $requester for file $pathname";
  $record_file = "/home/com/publications/$requester.$pathname";
  Save {append} $record_file $message;

Subroutine grant_authorization {group signer}{
  Contact Agent <authorize; grant> {users: $auth_dest}:
    "I grant authorization to $requester for file $pathname.";}
```

Figure 4-1: A SodaBot *authorize* agent for approving document publication

task-specific software agents leads us to estimate that an agent of this complexity would optimistically require several weeks of implementation time and roughly 50 pages of code. This SodaBot agent took 10 minutes to write and is approximately one page in length.

The SodaBot version of the document publishing process works as follows:

1. The document author sends mail to the *authorize* agent at an aliased "publications" e-mail address. This mail contains the name of two "authorized signers" and the pathname of the viewable document.
2. The *authorize* agent contacts the authorized signers by sending mail to the *certify* section of their *authorize* agents.
3. The authorized signers are given the opportunity to examine the document and certify it.
4. Authorizations are sent back to the publications officer and recorded.

In this example, the BSAs of the authorized signers run the *signer* section of the agent, and the BSA of the publications officer runs the *publications* section. Recall that the "group" specifications in the SodaBotL program determine which other sections of the program get distributed when a BSA requests some specific portion. For example, when the BSA of an authorized signer first requests the *certify* section of the *authorize* agent, it is also sent all other members of the *signer* group, in this case, the *grant_authorization* subroutine.

It is essential that messages mailed by agents be readable by people who do not use SodaBot.[2] Therefore, agents must send human-readable text. Note that the *certify* section refers directly to variables, such as *$requester*, which have been substituted into the text message it receives without any explicit parsing of the message body or requirements for structured format. This is achieved by including an unobtrusive extra header in the message which contains minimal sufficient information for determining the demarcations of the text substituted for the variables referenced by the receiving agent. (Page 35 illustrates such a header.) The BSA automatically extracts this text and stores it in the appropriate local variable(s).[3]

4.2 SodaBotL Reference Manual

This section is an abbreviated guide to programming in SodaBotL. Readers who know C or Perl may find it useful to keep in mind that SodaBot's syntax is loosely related to each of theirs.

We discuss in this section the following topics:

(4.2.1) Where the BSA looks for things

[2] Of course we exclude here messages intended for SodaBot's internal use, such as a request for agent distribution.

[3] SodaBot currently does not interact well with people who do not have SodaBot BSA's. We are currently adding automated construction of a structured text form-generator based on the *required input* specification. This will be coupled with automated error handling for user text-based input which differs from an agent's specified input format that will result in appropriate explanatory error messages.

(4.2.2) Variables in the BSA

(4.2.3) Conditionals

(4.2.4) How to write a mail filter

(4.2.5) How to write an application agent

(4.2.6) SodaBotL primitives

4.2.1 SodaBot File Hierarchy

When a user first runs SodaBot, it creates a directory hierarchy in “~/.sodabot/.” The top-level subdirectories in this hierarchy are:

1. Agents — Contains compiled SodaBot application agents.
2. AgentsSrc — Contains the SodaBotL sources to all agents received by the BSA and to agents written by its owner. Each application agents is kept in a separate subdirectory.
3. DBase — Stores the sleeping, system, and delivery databases.
4. Lib — Internal SodaBot library files
5. Log — Stores the BSA's extensive, human readable logs of its activity.
6. NonAgents — Contains compiled SodaBot mail filters.
7. NonAgentsSrc — Contains the SodaBotL sources to the mail filters.
8. SodaBot — Internal SodaBot directory, stores information about owner.

Users can create new agents and mail filters through the SodaBot *agent editor*, or they can write and compile SodaBotL files directly. The agent editor is invoked via the “Programmer” menu in the main GUI window. (See figure 1-2.) The SodaBotL compiler looks for its input in the appropriate “Src” directories and places compiled output in the appropriate object file directories. A sample interaction might be:

mhcoen@double-chex>**compile test.sbt**

SodaBot Compiler V1.1 running.

Reading /home/c2/mhcoen/.sodabot/NonAgentsSrc/test.sbt
Checking syntax of syntax of compiled program...
Writing program to /home/c2/mhcoen/.sodabot/NonAgents/test.pla

4.2.2 Variables

All variables in SodaBotL are prefaced with a *$*. Variables are typed according to their current context, so no explicit type declaration is necessary; for example, "10" can be either a number or a string, depending on how it is used. Variable assignment and reference work as you would expect. Here are some sample expressions:

$name = $username;
$address = "545 Technology Square
Cambridge, Ma 02139\n";
Mail to $host:
"My snail-mail address is:\n$name\n$address";

Strings can be multiline and can contain \" to quote things, e.g.

$quoted_string = "\"Inside Quotes!\"";

The BSA automatically defines a number of variables. These include variables corresponding to all headers present in the current e-mail message as defined by [Horton, 1983]. (For example, *$from* and *$subject*. See the BNF "Field" production on page 73 for a list of all valid headers.) Also, the following are always kept current:

Variable name	Description of value
$home	user's home directory
$user[_]name	user's full name
$user_login, $me	user's login name
$message	the complete text of the current message
$body	the body of the message
$name	sender's full name if specified
$address	sender's e-mail address
$reply-to	if not specified, value of $from is used

4.2.3 Conditionals

Conditionals in SodaBotL perform regular expression matching and numerical comparisons:

Expression	True if
($a eq $b)	$a and $b are equivalent
($a eq "moo")	$a equals the string "moo"
($a neq $b)	$a and $b are not equivalent
($a =~ /$b/)	$a contains $b
($a =~ /moo/)	$a contains "moo"
($a =~ /\d+/)	$a contains a number, etc.
($a !~ /$b/)	$a doesn't contain $b, etc.
($a == $b)	$a and $b are numerically equal
($a <= $b)	$a ≤ $b, etc.

Regular expression are contained in "/.../" do not require internal quotations. Good references for building regular expressions are [Wall and Schwartz, 1990, p24-29] and [Dougherty, 1990].

You can use && for AND and || for OR and ! for NOT. Conditionals can be nested in the standard way, e.g

if ((($from eq "las") || ($from eq "gjs")) && ($subject =~ /6\.001/))

4.2.4 Writing a Mail Filter

The BSA can execute a series of SodaBotL expressions upon the arrival of specific incoming e-mail. The user specifies which e-mail triggers the BSA by providing a number of regular expressions that must match the e-mail's headers and/or body. For each of these triggers, the user also specifies the BSA's appropriate reaction.

A *mail filter* is a list of these (trigger, response) pairs. User's can create multiple mail filters which are examined by the BSA when there is incoming e-mail. The BSA can also batch new messages, waiting until a certain minimum number (specifiable by the user) arrive before running the mail filters. Currently, SodaBot does not give the user particularly good control over incoming message volume, i.e., how many windows the BSA is allowed to pop-up on the user's screen. While the announcement of new messages can be delayed, there is no mechanism for prioritizing a group of incoming requests and perhaps eliminating those of lesser importance or allowing the user to select among them. We hope to add these capabilities shortly.

SodaBotL mail filter primitives:

Mail filter: (No arguments)

Declares that the following expressions constitute a mail filter. All subsequent SodaBotL expressions up to the next *Agent* or *Mail filter* statement comprise the body of the mail filter.

Received Mail [{*header1: /reg-exp1/; header2: /reg-exp2;* ... } *expressions*]

If mail arrives where each header matches the corresponding regular expression, execute the given SodaBotL expressions. A *received mail* statement with an empty header-regular expression list gets triggered on every message.

Sample expressions:

Mail filter:

Received mail {from: /fax notifier/; to: /$me/; subject: /arrival/}:

Display "An incoming fax has arrived.";

Received mail {from: /$me/;}:

Save {append} "~/Mail/outgoing" $message;

4.2.5 Writing an Agent

SodaBot application agents are divided into four sections: (See figure 3-1.)

1. Global declarations

2. The main agent
3. Agent requests
4. Agent subroutines

This section presents the syntactic specification of each of these sections. A model for approaching application agent design was discussed in section 4.1.

Global declarations

Global declarations are variable assignments which can be referenced by all agents requests and subroutines. They are included in all distributions of the agent.

Agent Name:

[*Required input* {*String_1: *type_1*
*String_2: *type_2* ...}]

[*SodaBotL expressions*]

The *Agent* declaration begins the definition of the named agent. All subsequent SodaBotL expressions up to the next *Agent* or *Mail filter* statement comprise the body of the entire agent.

The *Required input* specifies the format of the input which must be supplied to the agent. The SodaBotL expressions are run by the agent when it is invoked.

Required input {*String_1: *type_1*
*String_2: *type_2* ...}]

The *Required input* specification is a template for describing the format of the structured e-mail message which invokes the agent. Each line in the specification must match a corresponding line in the message such that the ordering is preserved. The type expression **type_i* can be any of the types shown on page 73 or an arbitrary regular expression.

For example, we again note the correspondence between the e-mail from page 35 and the *authorize* agent in section 4.1:

To: authorize@ai.mit.edu	*as input to*	**Agent** authorize:
------		**Required input** {
Person1: las		person1: *username
Person1: brooks		person2: *username
pathname: ~mhcoen/tr.ps		pathname: *pathname}

Request Name: {*group[s] group1, ...*}

[*Required input* {*String_1: *type_1*

*String_2: *type_2 ...*}]

[*SodaBotL expressions*]

A *Request* declaration begins the definition of the named request. All subsequent expressions up to the next *Request* or *Subroutine* or until the end of the agent's program comprise the body of the request. Requests can specify *Required input* to allow people to invoke them directly. However, assuming it was invoked by a BSA, a request's body can directly reference variables substituted into the e-mail when it was sent. For example, note the correspondence between:

Agent A:	*contacting*	*(in Agent A)*
$person = $username;		**Request** hello: {group main}
Contact Agent <A; hello> {user: mhcoen}		Display "You received
"Hi, I am $person.";		greetings from $person.";

Subroutine Name {*group*[*s*] *group1*, ...}{

[*SodaBotL expressions*]}

A *Subroutine* declaration defines the named subroutine. The body of the subroutine is contained within the indicated brackets. Subroutines have no explicit arguments; however, they can directly access the variables defined in the requests which invoke them. Note that subroutines are invoked only by requests, not other by subroutines. They are called by prefacing their name with an "&."

4.2.6 SodaBotL Primitives

Contact Agent <Agentname; Requestname> {*user*[*s*]*: user1, user2*, ...} *[string];*

Agents issue requests to one another via *Contact Agent*. The *users* field specifies whose BSAs receive the given request. The optional *string* specifies input for the particular request being issued. Any variables referenced in this string can be directly referenced in the receiving agent. If a receiving BSA doesn't have either the specified agent or the particular request section of that agent, it issues use *Contact Agent* to get the software from the *RequestAgent* of whoever initiated this interaction.

Currently, agents contacts are only relayed via e-mail. We are beginning work on a TCP/IP connection for the BSA as an alternate and hopefully faster inter-agent communication medium.

Sample expression:

Contact Agent <007; setname> {*user: $secret*} *"James Bond";*

Load filename variable;

Loads the contents of *filename* into the named *variable*.

Sample expressions:

Load $filename $contents;

Load "/home/c2/mhcoen/.schedule" $schedule;

Mail [*to*] *address: string*

Mail simply sends the specified string the given address.

Sample expressions:

Load "~/letter" $letter;

Mail to mhcoenai.mit.edu: $letter;

Mail to $user: "Your toast has popped up.\n";

Reply with string

Replies to current mail message with given string.

Sample expression:

Reply with "My final offer is $USDollars.\n";

Save [{*append*}] *filename string;*

Saves (or appends) the specified string into the named file.

Sample expressions:

Save "/usr/tmp/current" "$from, $subject";

Save {*append*} *"/usr/games/XChess/$book_openings" "It was a dark and stormy night.\n";*

GUI Primitives

Display [{*choices(cstring_1, cstring_2); . . . ; no delay; timeout in interval; for interval*}] *string;*

Display creates a window containing the specified *string* on the user's current display. *Choices* allows the user to select possible responses from stacked rows of "radio buttons." The BSA puts the agent to sleep until she has made her selections unless *no delay* is specified. The SodaBotL *choice* array is indexed over all specified options, e.g. cstring_1 – cstring_i; after the user makes her selections, an element in the array is true if the user selected the corresponding button in the displayed window. The *timeout* (and equivalently *for*) specification force the window created by this command to disappear after the given interval and wake the agent up. Multiple new windows appear in a staggered, overlapping layout to reduce screen clutter.

Display (and other GUI-specific primitives) have emacs-style "hooks" which can be invoked before the command is executed. These hooks are simply appropriatety named subroutines, e.g. *Display_hook*, which are intended to allow the GUI access to be redirected to another communication medium.

Sample expression:

Display {choices(yes, no); for one hour} "Mail from Lynn!
Do you want to read it now?";

Get Response [{*prompt = string; no delay; timeout in interval; for interval*}] *string;*

Get Response displays creates a window containing the specified *string* on the user's current display. This window also contains a mini-editor that allows the user to enter an arbitrary textual response to the specified *prompt*. This response is available to the SodaBotL program in the *$response* variable.

Sample expression:

Get response {prompt = "What is your answer?"}
"$quiz_question";

Query [{*type = reg-exp; prompt = string; no delay; timeout in interval; for interval*}] *string;*

Query displays creates a window containing the specified *string* on the user's current display. It allows the user to enter a one-line response which much match the given *reg-exp* in response to the given *prompt*.

Sample expression:

Query {prompt = "When you are you free?"; type = time}
$schedule;

4.3 The Benefit

SodaBotL contributes more complete solutions to two of the three problems outlined above with software agent construction.

1. Software agents can be technically challenging to write in traditional programming languages and operating systems:

   - SodaBotL offers high-level primitives and control-structures designed around human-level descriptions of agent activity. It allows users to easily create automatically-distributed software agents while ignoring low-level implementational details.

   - Because SodaBotL agent programs can be much shorter and higher-level than their counterparts in more traditional systems, they are easier to debug and maintain.

2. People may be uncomfortable with the amount of responsibility given to an unknown (and possibly buggy) agent.

   - Users can inspect SodaBotL programs for security threats more easily than would be possible with agent programs written in other programming languages, because: (1) the level of discourse is so much higher; and (2) the programs are smaller.

Chapter 5

Software Agents

The thesis has so far presented a specific view of what software agents are and how they should be used. However, software agents come in many flavors, and they differ widely in terms of specialization, usefulness, and theoretical motivation. This chapter discusses related work in the field and then outlines the common ties that link even vastly dissimilar agent implementations. We simultaneously discuss how SodaBot relates to and differs from other work in the field.

Section 5.3 presents our motivation for designing the SodaBot system based on experience described in [Kautz *et al.*, 1994]

5.1 What's an Agent?

There is simply no set of necessary or sufficient criteria for determining whether some program is indeed a software agent. The "definition" varies widely, as do approaches to building agents. However, we can look for some general agent characteristics by examining some typical (and not necessarily mutually exclusive) approaches.

We note here that no other software agent system (of any flavor) that we know of has an automated distribution mechanism like SodaBot's. In fact, personal communication with several of the researchers below indicates that this currently poses some difficultly, because few people outside of their research groups can make use of their agents.

5.1.1 Software agents are on-line pseudo-people

Software agents are ontologically grounded in their role in the agent community. Agents have beliefs, commitments, obligations, intentions, and perhaps even confusion, stubbornness, etc. Exactly what these agents do with all their commitments, obligations, intentions, etc, has not necessarily been made particularly clear, but what's supposed to be important is that we have a motivated vocabulary for describing coordinated agent interaction, e.g. *Agent1* sent *Agent2* e-mail because it felt "obligated," or perhaps *Agent1* crashed the network because it was "confused."

[Shoham, 1993] has defined an formal language for describing agents' "mental states" in terms of epistemic logic. He also presents a corresponding agent programming language called AGENT-0 ([Torrance and Viola, 1991]) which is semantically grounded in this mental state language. AGENT-0 very much resembles Prolog, but it has primitives which are well-suited for communication of obligations, beliefs, and capabilities between agents.

Whereas SodaBot is intended for assisting with practical, on-line tasks, AGENT-0 is suited for researching the interaction of coordinated cognitively-based agents, i.e. agents that think, but don't do much else. It would seem that neither system would be particularly adept at handling the job of the other. His approach does not necessarily conflict with our own. In fact, it would be very interesting to try combining aspects of both systems by providing BSAs with some type of formal intentional state.

We note that there is much other theoretical research into agent cognition, such as [Doyle *et al.*, 1991]. Again, it would be very interesting to ground this work by implementing it in a realized system.

5.1.2 Software agents are a testbed for other realms in Core AI.

Software agents are the new universal research tool for AI. Because expert systems and robots are leaving the limelight, software agents (and the people who research them) should enjoy their moment in the sun.

[Etzioni, 1993] argues that software agents are an ideal "foundation for core AI research." While we agree with this conclusion, we do not accept the arguments he uses to reach it (see [Coen, 1994]). Regardless, Etzioni *et al*'s work on Unix "softbots" ([Etzioni *et al.*, 1992a, Etzioni *et al.*, 1993, Etzioni and Segal, 1992, Etzioni *et al.*, 1994]) provides a very interesting foundation for exploring many central issues in traditional core AI, particularly in planning. There are many differences between this work and our own. Softbots are intended for much more system-administration oriented applications than are SodaBots; therefore, the softbot level of discourse is in terms of (low-level) Unix primitives. Softbot agents do not seem to interact with anything other than their owners, and thus, their capabilities do not extend to inter-agent communication.[1] Finally, the softbot system does not seem to have any provisions for assisting with distribution of softbot agents or their UWL plans.

The Darpa Knowledge Sharing Effort ([Neches *et al.*, 1991]) has encouraged much agent-based research into knowledge representation and communication languages. This effort has led to the design of an agent communication language (ACL) intended as a universal medium for agent discourse. Genesereth *et al.* ([Genesereth and Singh, 1994, Genesereth and Ketchpel, 1994]) present a "federation" agent architecture that employs this ACL, and [Genesereth 1994] discusses these agents obtaining arbitrary software programs from other agents by advertising their required specifications written in ACL.

[1] We don't consider even sophisticated interaction with disk drives and printers to be communication.

It is worth noting that work on ACL has yet not been completed, so agent systems which communicate in ACL do not yet exist. We also remain highly skeptical of this ACL's ontological sufficiency and soundness. Furthermore, agents would have to "know" a program existed before they could advertise for it; this type of distribution does not address how novel programs are spread among networked agents. Finally, this work makes no mention of the practical consequences its type of distribution would entail, nor does it discuss the required effort to realize the described hypothetical agents.

The work of [Vere and Bickmore, 1990] is quite unusual. Their "basic agent" has a remarkably wide core AI foundation, drawing on a broader range of research areas than any other system with which we are familiar. However, their domain is so narrow and their application so involved that it bears little resemblance to any current work in software agents.

5.1.3 Software agents are intelligent on-line assistants

Software agents are artificial secretaries which are the electronic counterpart of their real-world namesakes. This is not to say that anyone looks forward to the prompt delivery of simulated coffee each morning! Rather, these personal assistants are designed for tasks such as: filtering e-mail, scanning NetNews, providing appointment reminders, etc.[2] Given the complexity of on-line environments and huge volume of information flowing across the Internet, this type of agent looks quite attractive.

Interface agents ([Maes, 1994, Sheth, 1994]) are a special class of on-line assistants which are designed to simply user-interaction with particular pre-existing applications. These agents are designed to learn and predict users' behaviors and preferences. SodaBots have little in common with interface agents, because interface agents are each highly elaborate, custom-crafted programs designed for very specific

[2] One rather extreme and slightly dismaying example of an on-line assistant might be the H.A.L. 9000 in 2001: A Space Odyssey.

applications. Also, SodaBot does not have built-in capabilities for learning user behavior and preferences. Providing these would be an interesting direction for future development of the SodaBot system.

For example, Sheth provides an interactive information retrieval system for UseNet NetNews articles. It is designed to autonomously select those articles whose content would interest the user according to some complex metric involving the user's previous selections. Sheth's work is similar in spirit to much of the activity regarding *knowbots*™ ([Kahn and Cerf, 1988, Waldrop, 1990]), which has generally fallen outside of the realm of mainstream AI research, although there are some exceptions, e.g. [Knoblock and Arens, 1994]. Knowbots (knowledge robots) are intended roughly as librarians for enormous digital data-libraries. They are not general purpose but are highly elaborate, specific creations. Thus, these too have little if anything in common with SodaBots.

5.1.4 Software agents are negotiators

Groups of software agents can make decisions or form coalitions. If a group of people with complex time-constraints need to arrange a meeting, software agents can do it for them without requiring that a person bother with the intricate constraint balancing inherent in meeting-scheduling (and perhaps without anyone's feelings getting hurt).

In fact, meeting scheduling is the most popular software agent negotiation application. [Kozierok, 1993, Maes and Kozierok, 1993] schedules group meetings, [Kautz *et al.*, 1994] schedules meetings between individuals, and [Dent *et al.*, 1992] does both (and more). The backbone of all of these systems could be implemented in SodaBot. However, the actual scheduling processes would require external applications. For example, Kautz *et al.*'s VisitorBot requires use of CPlex — a sophisticated integer programming package — which could be accessed through the SodaBot *system* command.[3]

[3]CPlex actually requires a very expensive machine-specific license. However, we note (without

There has also been much theoretical work on abstract agent negotiation protocols, as in [Zlotkin and Roesnschein, 1994, Rosenschein, 1993]. As we pointed out earlier with reference to the work of Doyle, it would be very interesting to ground this theoretical work in negotiation by implementing it in a realized system.

5.1.5 Other Points of View

[Stein, 1994] has suggested that "agency" (i.e., the property of being an agent) is determined by an observer's intentional stance; what a person views as an agent *is* an agent. While this may well be a tenable philosophical position, it is not clear what benefit it provides. Rather, in term of directing research efforts, particularly with the growing popularity of "agents," it might be preferable to narrow the scope of the term. Even in research communities this designation is perhaps being abused. Work such as [Lansky, 1994] was once called an *expert system.* It was quite surprising to hear such a classic example of that paradigm being presented at the AAAI 1994 Spring Symposium on Software Agents.

The final system we discuss is Telescript ([Wayner, 1994]). Although few details of this proprietary system have been disclosed, enough information has been released to permit a tentative comparison. Telescript is a very sophisticated computational environment in which machine-independent programs move freely around a network. Telescript programs are interpreted, and interpreters exist for all standard platforms. Essentially, in terms of portability, it is the algorithmic equivalent of "postscript."

High Telescript, the system's programming language, is reportedly very similar to Smalltalk and Modula-3. Thus, it does not provide the right level of abstraction for writing agent applications. Furthermore, Telescript programs have fixed meanings, i.e. primitives are not interpreted with respect to their context. However, it would seem that Telescript might be an ideal system for reimplementing SodaBot.

advocating) that by setting up a BSA on the machine on which CPlex has been installed, it is trivial to allow anyone, anywhere, to access CPlex via a simple SodaBot application agent.

Regardless, we look forward to seeing what comes out of this very promising endeavor. Telescript seems like a step in the right direction.

5.2 Agency Defined

Clearly, there is an enormous variety in what people deem a software agent, and it is somewhat difficult to tell whether this is good or bad for the field. Nonetheless, having many enthusiastic researchers working on their various "agents" is probably to everyone's benefit, so we refrain from complaining too loudly.

However, we favor the following as a set of minimum criteria for establishing a program's "agency:"

1. Software agents engage in dialogs; we don't issue commands to agents, rather we have conversations with them. The communication patterns among agents can be quite complicated.

2. Software agents are autonomous and intelligent; they respond to complex stimuli with sophisticated (and appropriate) behaviors.

3. Software agents must be robust. Because they are autonomous and presumably doing something important, agents must be able to respond to unexpected changes in their computation world.

4. Software agents are generally not *time invariant* — they have memory and change what they do over time. Agents can employ formal machine learning techniques, or they can more casually collect data while they operate. Personal assistants can learn patterns in their owners' behavior, and more generally, agents can spontaneously react to particular events in their computational world.

5. Software agents are typically distributed across a network, so their behavior can

have both local and global effects. Abstraction barriers can become confused if an agent is responsible for too many non-local events.

5.3 SodaBot's Motivation

The SodaBot system was heavily influenced by my participation in developing the *VisitorBot* ([Kautz *et al.*, 1994]) in the AT&T Bell Laboratories' AI Principles Research Group.[4]

5.3.1 The VisitorBot

The VisitorBot is a software agent that schedules meetings with a visiting researcher (who is presumably also giving a talk). The VisitorBot distributes the talk's abstract and accepts requests for meetings with the speaker. It then distributes schedule-outlines which are filled out by those interested in reserving a time slot. Finally, after receiving submitted time constraints from users, the agent generates (and distributes) a schedule of meetings with the visitor.

The history behind the development of this agent is revealing. The version described in [Kautz *et al.*, 1994] was begun at AT&T Bell Labs while I was a summer student there. However, due to numerous technical difficulties, this agent was not yet completed by the time I returned to MIT at the end of the summer.[5] Therefore, I ported the agent to the MIT AI Lab in order to finish working on it. Interfacing the agent to the AI Lab's mailer involved nontrivial effort, and after completing it here, installing and debugging the agent at Bell Labs remotely from MIT required a ridiculous amount of time. (This was primarily due to Bell Lab's network "firewall.")

While writing the VisitorBot (among other agents)[6], it became clear that get-

[4] I worked at Bell Labs from the the middle of May through the first week of September during 1993.

[5] Steven Ketchpel actually implemented an earlier, complete version of the VisitorBot on top of a simple mail-reading agent developed by Henry Kautz.

[6] I wrote several other software agents at Bell Labs over the summer. Most notable is the LaTeXBot

ting an agent to run at a particular location required a large amount of site-specific information. Additionally, an agent which centrally controlled all aspects of user-interaction was prone to failure in a networked environment. (For example, it is not possible to open X-Windows across a firewall.) Essentially, there were no clean abstraction barriers for writing agent software.

The development of personal agents in [Kautz *et al.*, 1994] was a first step towards establishing some minimum level of distinction between local and non-local agent activity. For example, the VisitorBot could tell a user's personal agent to open a window on the user's display rather than doing so itself. However, both the VisitorBot and its involved personal agents are very much *ad hoc*, non-generalizable creations. The personal agent in [Kautz *et al.*, 1994] is hard-coded and custom tailored to the VisitorBot, i.e. a hypothetical *PaperReviewBot* would require that users obtain a different personal agent to interact with it. According to this approach, every time a new agent is written, each user must install the appropriate personal agent to permit interaction with it.

SodaBot was my reaction to the effort required for writing and installing the VisitorBot. Although we found that it was generally very easy to state succinctly the desired agent behavior in English, it was quite another thing to formulate this in Perl and C. This distinction differentiates between SodaBot and efforts in the field of automated programming. Loosely speaking, the shortest specification of a program is generally the program itself; however, given the highly specialized domain in which software agents function (at least in SodaBot), it is usually quite easy to give a short high-level specification of an agent's desired behavior. SodaBot takes advantage of this by allowing an agent creator to provide merely this high-level specification. The system essentially handles all the effort involved in actually realizing the specified agent.

---

which allowed me to edit my SM thesis proposal at Bell Labs and process it remotely at MIT. It notified me of any errors encountered during text processing at MIT, displayed the final results on my Bell Labs' workstation, etc.

We did not address in [Kautz *et al.*, 1994] how a new agent is released to the world. The VisitorBot was a collection of random C and Perl files which had to be installed and configured by a skilled human being. The difficulties inherent in encouraging use of new agents are thus enormous. Not only would a new user need to be convinced that the files are safe to install and to use, but she would additionally have to be willing to trust the system could, for example, handle her e-mail properly. Finally, installing and running the VisitorBot also sometimes required root access, which generally would prevent the average user from installing it herself.

How to distribute new agents was the subject of much discussion over the summer. The approach in SodaBot was motivated by a discussion of distributed agent planning at a Bell Lab's Bot meeting, where Ron Brachman suggested that planning agents could e-mail STRIPS operators to each other. I was quite taken with this idea and it eventually found its way into SodaBot (where agents instead send SodaBot programs).

Chapter 6

Conclusions

This chapter evaluates the SodaBot system; we discuss it strengths, weaknesses, and future work.

6.1 SodaBot's Report Card

We could evaluate the SodaBot system by the following criteria:

1. It solves the problems listed in section 1.2.
2. Naive users enjoyed interacting with it.
3. We learned something building it.

However, only the first and third are currently capable of being assessed, because we did not have genuinely naive users test SodaBot. Nonetheless, we examine each of the three criteria in turn and discuss how it is addressed by various components of the SodaBot system.

It solves the problems listed in section 1.2:

1. Software agents can be technically challenging to write in traditional programming languages and operating systems:

- The BSA provides the right foundation for software agent creation. It removes system specific aspects of agent creation from the domain of the agent programmer.
- SodaBotL offers a level of discourse appropriate for the types of on-line activities in which agents engage. The high level primitives in SodaBotL allow agents to be written more quickly and in less space.

  For example, we can approximately implement the "VisitorBot" system described in [Kautz *et al.*, 1994] in several pages of SodaBotL code.[1] The VisitorBot implementation described there is well over 50 pages of Perl and C code.

2. Software agents are difficult to distribute because they may have site-specific dependencies; for the same reason, they can be difficult to install.

   - The BSA disconnects application agents from the specific computational environment in which they run. Agent programs no longer need to be "hard-coded" with site-specific information. The problem of configuring many software agents is reduced to the problem of configuring a single agent, i.e., the BSA.
   - Agents in SodaBot distribute themselves over the Internet and/or local area networks. Although there is currently only a Unix platform for SodaBot, this distribution can theoretically be across operating systems. Simply put, the acquisition of new agents in SodaBot does not require that the user do anything.

3. People may be uncomfortable with the amount of responsibility given to an unknown (and possibly buggy) agent.

[1] In terms of features, the two scheduling agents are not strictly comparable yet. Kautz *et al.*'s looks better. Ours is more robust.

- The BSA allows the user to gradually establish confidence in its behavior and to selectively disable and enable access to specific system resources. It provides a stable agent-framework over which the user can exert ultimate control.

  The system as a whole has seen four months of use, testing, and debugging, and its longest continuous operation without restarting has lasted approximately one week. It has been reliable handling e-mail, i.e., it has not lost messages, and it has had low system overhead.

- Users can inspect SodaBotL programs for security threats more easily than would be possible with agent programs written in other programming languages, because: (1) the level of discourse is so much higher; and (2) the programs are smaller.

We feel SodaBot successfully presents solutions to the specified problems. However, most of our time has been spent developing the system, not using it. We have built many "toy agents" but very few large-scale ones. Therefore, we can't (yet) claim a flood of agent development has resulted from the SodaBot system. Hopefully, when the system is "bullet-proofed" and released for general use, more agent applications will be forthcoming.

Finally, we acknowledge that evaluating the SodaBot system is not a necessarily objective process. We have received the critique "[the authorize agent in section 4.1] would be very easy to implement in very few lines of [non-SodaBotL] code." Even though this agent is a quite simplified version of what an "end product" would require, we disagree and suggest that those who are skeptical of our position actually go ahead and implement the agent in "very few lines."[2]

---

[2]Some things to consider include handling: (1) network downage; (2) e-mail lossage; (3) user I/O; (4) distributing the agent so that people can use it; etc.

Naive users enjoyed interacting with it.

SodaBot is not intended to be a "Unix wizards only" tool. A major consideration while designing it was to make it as user-friendly as possible. For example, the compiler gives quite instructive error messages; it points out explicitly the offending statement and suggests what might be the problem and how to fix it. However, we can't say any genuinely naive users have used it, so the assessment of this criterion will have to wait.

There are some definite aspects of the system that need improvement, particularly with respect to mail filters. For example, users will almost certainly insist on some way of prioritizing mail filter firing so they don't get flooded by window's popping up on their screen after a long absence. This does not constitute a major addition, and implementing user demands can only increase the value of the system.

We learned something while building it.

Certainly, a fair amount (perhaps too much) of Unix, X-windows, C language, etc. knowledge was acquired building SodaBot. It provided ample opportunity to learn about the many obscure aspects of building large "real-world" systems.

However, perhaps the best indication that we learned something is that we are immediately setting out to rewrite it. We want to make SodaBot more user-extensible: she should be able to declare new SodaBotL primitives, enhance the GUI, and easily hook the BSA up to arbitrary system components, e.g. a speech synthesizer.

We do not plan on distributing the current SodaBot implementation outside of the MIT AI Lab. We will continue rewriting it, playing with it, and introducing it to the local community. We feel it was a very good first step towards making a general purpose software agent construction system. We very much look forward to completing the next release and seeing what people end up doing with it.

6.2 Closing Summary

This thesis has presented SodaBot, a software agent user-environment and construction system. The basic software agent was introduced as a foundation for constructing SodaBot application agents. We then presented SodaBotL — the software agent programming language — whose primitives are designed around human-level descriptions of agent activity. Via this programming language, users can easily implement a wide-range of typical software agent applications. Along the way we also discussed how people go about writing application agents and how SodaBot automatically distributes them.

Appendix A

Details of the Current Implementation

The primary difference between the current SodaBotL syntax and the one presented in the paper is in the structural division of agent programs. In both versions, application agents are stored in unique directories in "~/.sodabot/AgentsSrc/" However, the current version requires that each group of an agent be stored in a seperate sub-directory; thus, no single file contains an entire agent having 2 or more (non-trivial) groups. Each group is acutally treated as a unique application agent.

Also, in the current implementation:

1. All numbers must be inside quotes.
2. The types listed in the *agent input specification* on page 73 do not exist.
3. Order is not preserved for *required inputs*.
4. The "SodaBot-Parser:" header does not work for multiline strings.
5. The $message variable is not preserved if the agent is placed in one of the three databases.
6. Subroutines are identical to requests. An agent simply directs a request to its host BSA to call a subroutine, so variables must be passed explicitly.
7. The semi-colon and bracketing syntax are slightly different.

Appendix B

SodaBotL BNF Specification

Functional Declarations:

<Program>	::=	<Mail filter> [<Program>] | <Agent> [<Program>]
<Mail filter>	::=	*Mail filter:* [<Declaration>*] <Mail description>*
<Mail description>	::=	*Received Mail* [{<Mail specification>}]*:*
		<Statements>*
<Agent>	::=	*Agent* <Agent name>*:*
		[<Input>]
		<Statements>*
		<Agent requests>*
		<Subroutine>*
<Agent request>	::=	*Request* <Request string>: [{ *group* <string> }] <Statements>*
<Subroutine>	::=	*Subroutine* <Sub name>([<Arg list>]): [{*group* <string>}]
		{<Statements>*}
<Declaration>	::=	*Library* <Library name> | <Declaration>

Statements:

<Statement> ::= <Assignment> | <System> | <Save> | <Load> | <Reply> |
<Mail> | <Contact agent> | <Sub call> | <If> | <GUI call>

<Assignment> ::= <Variable> = <String value>*;*

<System> ::= *System* <String value>*;*

<Save> ::= *Save* [{*Append*}] <¡Filename> <String value>*;*

<Load> ::= *Load* <Filename> <Variable>*;*

<Reply> ::= *Reply with* <String value>*;*

<Mail> ::= *Mail to* <Address>*:* <String value>*;*

<Contact agent> ::= *Contact agent* <<Agent name>*;*<Request string>>
{*users:* <Address list>} [<String value>]*;*

<Sub call> ::= *&*<Sub name>([<Arg List>]);

<If> ::= *If* (<Condition>) { <Statement> }
[*elsif* (<Condition>) {<Statement>}]*
[*else* {<Statement>}]

GUI Statements:

<GUI call> ::= <Display> | <Get response> | <Query>

<Display> ::= [{<Display options>*}] <String value>;

<Get response> ::= [{<Query options>*}] <String value>;

<Query> ::= [{<Query options>*}] <String value>;

Conditions:

<Condition> ::= <Boolean>
::= <String value> *eq* <String value>
::= <String value> *neq* <String value>
::= <Reg exp> *in* <String value> | <String value> =~ <Reg exp>
::= <Reg exp> *nin* <String value> | <String value> *!~* <Reg exp>
::= (<Condition>) *or* (<Condition>) | (<Condition>) || (<Condition>)
::= (<Condition>) *and* (<Condition>) | (<Condition>) *&&* (<Condition>)

Data Types:

<String value> ::= <String> | <Variable>

<String> ::= <Multiline string> | <Simple string>

<Variable> ::= $<Simple string>

<Filename> ::= [/] <Simple string> [/] [<Filename>] | <Variable>

<Sub name> ::= <Simple string>

<Agent name> ::= <Simple string>

<Library name> ::= <Simple string>

<Arg list> ::= <String value> [, <Arg list>]

<Multline string> ::= <Rich string> \n [<Multiline string>]

<Rich string> ::= All characters except \n

<Simple string> ::= [*a-z, A-Z, 0-9, _*]*

<Reg exp> ::= See [Wall and Schwartz, 1990, p25].

Mail Filter Specification:

<Mail specification> ::= <Field>: /<Reg exp>/ [;<Mail Specification>]

<Field> ::= to | cc | bcc | from | sender | reply-to |
return-receipt-to | errors-to | date |
return-header | message-id | subject | status |
newsgroups | followup-to

Agent Input Specification:

<Input> ::= *Required input* {<Input spec>}

<Input spec> ::= <Rich string>: <Input type>

<Input type> ::= <Rich string> | **name* | **username* | **filename* |
**date* | **time* | **address* | **host* | **number*

Bibliography

[Brooks, 1991] Brooks, Rodney. Intelligence without representation. *Artificial Intelligence*, 47:139-160. 1991.

[Coen, 1994] Coen, Michael. Letter to the Editor. AI Magazine. Summer 1994.

[Dent *et al.*, 1992] Dent, Lisa; Boticario, Jesus; McDermott, John; Mitchell, Tom; and Zabowski, David. A personal learning apprentice. In *Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI-92*, San Jose, CA. p96-103. 1992.

[Dougherty, 1990] Dougherty, Dale. sed & awk. O'Reilly Associates. Sebastopol, CA. 1990

[Doyle *et al.*, 1991] Doyle, Jon; Shoham, Yoav; and Wellman, Michael. A logic of relative desire. In Z.W. Ras and M. Zemankova (eds.) Methodologies for Intelligent Systems, Springer-Verlag, Berlin. p16-31. 1991.

[Etzioni *et al.*, 1992a] Etzioni, Oren; Hanks, Steve; Weld, Daniel; Draper, Denise; Lesh, Neal; and Williamson, Mike. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, KR-92*, Cambridge, MA. p115-125. 1992.

[Etzioni and Segal, 1992] Etzioni, Oren and Segal, Richard. Sofbots as testbeds for machine learning. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, Menlo Park, CA. 1992.

[Etzioni *et al.*, 1993] Etzioni, Oren; Levy, Henry; Segal, Richard; and Thekkath, Chandramohan. OS Agents: Using AI Techniques in the Operating System Environment. Technical Report 93-04-04. University of Washington, Seattle, WA. 1993.

[Etzioni, 1993] Etzioni, Oren. Intelligence without Robots: A Reply to Brooks. In *AI Magazine*, Winter, 1993.

[Etzioni *et al.*, 1994] Etzioni, Oren; Lesh, Neal; and Segal, Richard. Building softbots for Unix (Preliminary Report). In *Working Notes of the AAAI Spring Syposium on Software Agents*, Stanford, CA. p9-16. 1994.

[Genesereth and Singh, 1994] Genesereth, Michael and Narinder Singh. A Knowledge Sharing Approach to Software Interoperation. Unpublished draft. 1994.

[Genesereth and Ketchpel, 1994] Genesereth, Michael and Ketchpel, Steven. Software Agents. *CACM - Special Issue on Intelligent Agents*. 37:7. 1994.

[Genesereth 1994] Genesereth, Michael. MIT AI Laboratory Revolving Seminar. February 3, 1994.

[Horton, 1983] Horton, Mark. Standard for interchange of usenet messages. Internet Request for Comment (RFC) 850. 1983.

[Kahn and Cerf, 1988] Kahn, Robert and Cerf, Vinton. An open architecture for a digital library system and a plan for its development. Technical report. Corporation for National Research Initiatives. 1988.

[Kautz *et al.*, 1994] Kautz, Henry; Selman, Bart; Coen, Michael; Ketchpel, Steven. An experiment in the design of software agents. *Proceedings of the Twelfth National Conference on Artificial Intelligence, AAAI-94*, Seattle, WA. 1994.

[Knoblock and Arens, 1994] Knoblock, Craig and Aren, Yigal. An architecture for information retrieval agents. In *Working Notes of the AAAI Spring Syposium on Software Agents*, Stanford, CA. p49-56. 1994.

[Kozierok, 1993] Kozierok, Robyn. A learning approach to knowledge acquisition for intelligent interface agents. Technical Report 93-01, Learning and Common Sense Group, MIT Media Lab. 1993.

[Krishnamurthy and Rosenblum, 1992] Krishnamurthy, Balachander and Rosenblum, David. Yeast: a general purpose event-action system. AT&T Bell Labs Technical Memorandum. 1992.

[Lansky, 1994] Lansky, Amy. A data analysis assistant. In *Working Notes of the AAAI Spring Syposium on Software Agents*, Stanford, CA. p57-63. 1994.

[Maes, 1990] Maes, Pattie (ed). Designing Autonomous Agents, MIT Press, Cambridge, 1990.

[Maes and Kozierok, 1993] Maes, Pattie and Kozierok, Robyn. Learning interface agents. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI-93*, Washington D.C., p459-464. 1993.

[Maes, 1994] Maes, Pattie. Social interface agents: acquiring competence by learning from users and other agents. In *Working Notes of the AAAI Spring Syposium on Software Agents*, Stanford, CA. p71-78. 1994.

[Neches *et al.*, 1991] Neches, Robert; Fikes, Richard; Finin, Tom; Gruber, Thomas; Patil, Ramesh; Senator, Tod; and Swartout, William. Enabling Technology for Knowledge Sharing. In *AI Magazine*, Fall, 1991.

[Rosenschein, 1993] Rosenschein, Jeffrey. Negotiation mechanisms for multi-agent systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence,* Chambery, France. p792-799. 1993.

[Sheth, 1994] Sheth, Beerud. Adaptive Agents for Information Processing. SM Thesis. MIT. Cambridge, MA. 1994.

[Shoham, 1992] Shoham, Yoav. Agent Alpha Programming Overview. 1992.

[Shoham, 1993] Shoham, Yoav. Agent oriented programming. *Artificial Intelligence*, 60:51-92. 1993.

[Stein, 1994] Stein, Lynn. Private communication. (Also from comments made during the third discussion section at the AAAI 1994 Spring Syposium on Software Agents. Patrick Hayes made a similar statement.)

[Torrance and Viola, 1991] Torrance, Mark and Viola, Paul. The AGENT0 Manual. Technical Report STAN-CS-91-1389. Stanford University Department of Computer Science. Stanford, CA. 1991.

[Vere and Bickmore, 1990] Vere, S and Bickmore, T. A basic agent. *Computational Intelligence*, 6(1). 1990.

[Waldrop, 1990] Waldrop, M. Mitchell. Learning to drink from a fire hose. *Science*, v248, pg 674. 1990.

[Wall and Schwartz, 1990] Wall, Larry and Schwartz, Randall. Programming Perl. O'Reilly & Associates. Sebastopol, CA. 1990

[Wayner, 1994] Wayner, Peter. Agents away. *Byte*, p113-118. May, 1994.

[Zlotkin and Roesnschein, 1994] Zlotkin, Gilad, and Rosenschein, Jeffrey. Coalition, cryptography, and stability: mechanisms for coalition formation in task oriented domains. In *Working Notes of the AAAI Spring Syposium on Software Agents*, Stanford, CA. p87-94. 1994.