SodaBot: A Software Agent Environment and Construction System

Michael H. Coen*
MIT Artificial Intelligence Laboratory
545 Technology Sq. NE43-823
Cambridge, MA 02139
mhcoen@ai.mit.edu

September 14, 1994

Extended Abstract

Introduction

In this extended abstract, 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.

*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.

Software Agent?

What exactly is a software agent? The answer depends on whom you ask and can vary quite widely (see [Coen, 1994, Chap. 5]). For our purposes here, we will focus exclusively on *application agents*, i.e. agents that automate simple on-line, repetitive and time-consuming tasks. For example, a popular type of application agent is the time scheduler which arranges group or individual meetings among a set of people with respect to individual's personal temporal constraints ([Kautz *et al.*, 1994, Dent *et al.*, 1992, Maes and Kozierok, 1993]).

More generally, we favor the following as a set of minimum criteria for establishing that a piece of software is indeed a "software agent." Typically, software agents:

1. engage in dialogs; we don't issue commands to agents, rather we have conversations with them.
2. are autonomous and intelligent; they respond to complex stimuli with sophisticated (and appropriate) behaviors.
3. must be robust. They must be able to respond to unexpected changes in their computational world.
4. are not time invariant; they have memory and can change what they do over time.
5. are distributed over networks. Abstraction barriers can become confused if agents are responsible for too many non-local events.

The Problems with Software Agents

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, e.g. [Shoham, 1993]; and (2) applied construction of specific agents, e.g. [Etzioni *et al.*, 1994]. (See also [Riecken, 1994, AAAI Spring Symp, 1994].) However, researching software agents 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 elsewhere.

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?

The SodaBot Software Agent Approach

SodaBot is a general-purpose software agent user-environment and construction system. Its primary components are:

1. The *basic software agent* which implements an agent operating system
2. The high-level SodaBot *agent programming language* for programming the basic software agent. It provides the right level of abstraction to allow simple construction of complex agent applications.
3. A graphic user interface.
4. Automated distribution of software agents across the Internet.

The Basic Software Agent

In SodaBot, each user is given a personal *basic software agent* (BSA) which typically runs in the background of her home workstation. The BSA is an agent operating system — think of it as the "Unix of the software agent world." By this we mean it is a universal computational framework for creating and using software agent applications. The BSA is programmed in the SodaBot *agent programming language* (SodaBotL)[1]. As a quick sanity check, see if the following rough analogy makes sense:

[1]Pronounced "Soda-Bottle."

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.

The BSA disconnects agent applications from the specific computational environment in which they run. It handles both low-level operational tasks (e.g. accessing TCP/IP, reading e-mail) and architectural and system configuration compatibility, thereby pushing system-specific aspects of agent creation beyond the abstraction barrier. Thus, agent programs no longer need to be "hard-coded" with details of the specific computational environment in which they will run. This makes agent programs highly portable — they don't need to be disconnected from their "birth-site" or hooked-up to anything when being installed.

The BSA paradigm also assists in the development of user confidence in agent applications, because it provides the only interface to critical system resources, e.g. sending and receiving e-mail . Users can selectively disable and enable a particular application's access to these resources. Also, 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. For example, if a user has left for the day, it may be more appropriate for the BSA to fax her a message at home than display it on the computer screen in her office. (See below.)

The 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";}

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.[2] 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 have a much high-level of discourse and are 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.

The Graphic User Interface

The user and her BSA interact through the SodaBot graphical user interface (GUI). Figure 1 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.

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*

[2] [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.

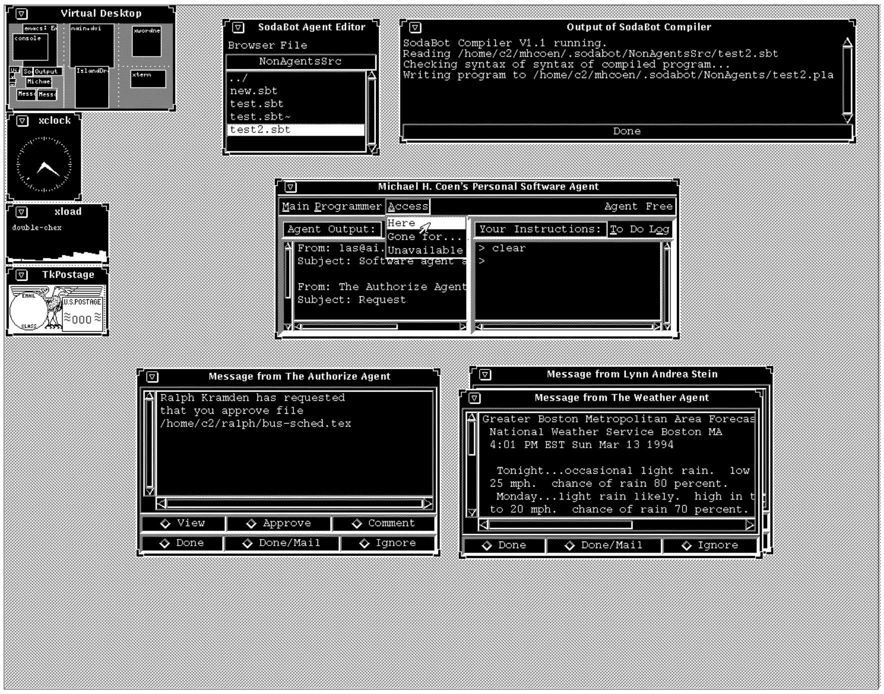

Figure 1: The BSA running on a Unix workstation

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 or from a centralized SodaBot agent depository. Note that this distribution can even be across platforms and operating systems. Thus, acquisition of new SodaBot agents does not require that the user do anything.

Also, 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.

Discussion

The SodaBot system has been built and tested, and it is in current development and use at the MIT AI Lab.[3] Several application agents were written during testing, including one with functionality similar to the "VisitorBot" system presented in [Kautz *et al.*, 1994].[4] Whereas our version of their VisitorBot is several pages of SodaBotL code, the implementation described there is well over 50 pages of Perl and C.

Summary

This abstract has outlined SodaBot, a software agent user-environment and construction system. The basic software agent was presented as a foundation for constructing SodaBot application agents. We then introduced 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. Finally, we briefly described SodaBot's graphical user interface and automatic distribution of agent applications across the Internet.

References

[AAAI Spring Symp, 1994] *Working Notes of the AAAI Spring Symposium on Software Agents*, Stanford, CA. 1994.

[Coen, 1994] Coen, Michael. SodaBot: A Software Agent Environment and Construction System. SM Thesis. MIT. Cambridge, MA. 1994.

[3] We expect that it will soon be available through public ftp.

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

[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.

[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.

[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.

[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.

[Riecken, 1994] Riecken, Doug (ed). *CACM - Special Issue on Intelligent Agents*. 37:7. 1994.

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