April 1991 Report No. STAN-CS-91-1389

The AGENT0 Manual

by

Mark Torrance and Paul Viola

Department of Computer Science

Stanford University
Stanford, California 94305

The AGENT0 Manual

Mark C. Torrance
Program in Symbolic Systems
Stanford University
Stanford, California 94305
torrance@cs.stanford.edu

Paul A. Viola
Artficial Intelligence Laboratory
Massachussets Institute of Technology
Cambridge, Massachussets
viola@ai.mit.edu

April 9, 1991

This document describes an implementation of AOP, an interpreter for programs written in a language called AGENTO. AGENT0 is a first stab at a programming language for the paradigm of Agent-Oriented Programming. It is currently under development at Stanford under the direction of Yoav Shoham. This implementation is the work of Paul A. Viola of MIT and Mark C. Torrance of Stanford.

1 Introduction

AOP, *Agent Oriented Programming,* is a programming paradigm proposed by Professor Yoav Shoham of Stanford University. It imposes certain constraints on the nature of agents and of their communication. AGENT0 is a more restrictive language, in which programs in the spirit of AOP can be written. Both AOP and AGENT0 are described in [1].

This document describes an implementation of an interpreter for agent programs written in the AGENT0 language. This implementation purports to be a complete implementation of the AGENT0 language as defined in Shoham's paper. This implementation is still under development, but the most recent released version should be fairly complete and accurately described by this document.

2 Obtaining AGENT0

This implementation of AGENT0 is available for Allegro Common Lisp running on UNIX workstations maintained at the Stanford Robotics Lab, AKCL running on Sparcstations maintained by AIR at Stanford, and for Macintosh Computers running Allegro Common Lisp. The code should be portable to any Common Lisp implementation, although there may be special purpose applications which run only on one or some of the supported platforms (e.g. a graphical front end in X). This manual describes only the main, portable interpreter/ simulator.

Hereafter in this manual, the notation `<a0>` will refer to the directory in which your AGENT0 files *are* stored. You should set up such a directory in a convenient place, and copy all of the AGENT0 files in'qto it.

If you are a member of the Nobotics group at Stanford, the AGENT0 files are available in the directory `~aop/lisp/a0`. You don't need to copy these; just run them from this directory. Hereafter, you should use `~aop/lisp/a0` wherever you see `<a0>`.

AGENT0is also available on the Andrew File System, a national network-transparent filesystem, in the directory `/afs/ir.stanford.edu/users/t/torrance/aop`. Copy all files from this directory into a directory (hereafter `<a0>`) on your own machine, and change the variable *sop-load-path* in `<a0>/load.lisp` to point to the place where you put the `<a0>` directory. This pathname should either be absolute for your machine, or relative to the `*default-pathname-defaults*` of Lisp as you start it, which is the pathname of the directory you are in when you started Lisp.

At the moment, uses a separate parser, Paren, to parse source files. You also need to copy the files from the directory `/afs/ir.` Stanford. edu/ users/ t/ torrance/ paren to a directory, usually a sister to `<a0>`. Change the variable *paren-load-path* in `<a0>/load.` lisp to point to the directory where you put the Paren files. Paren will not be used in future versions of this interpreter.

3 Running AGENT0

To run AGENT0, first start up Common Lisp. The command to do this will depend on your system, but could be cl, `akcl`, `acl`, or clicking on an icon for Allegro Common Lisp on a Macintosh.

Next, load the file `<a0>/load` by typing (load ' 'load' ') to Lisp. If you were not in the `<a0>` directory when you started Lisp, you should type the pathname of that directory to the load command, as in (load `''aop/lisp/a0/load''`), which works on Nobotics lab machines.

4 The (aop) function

After you have loaded AOP, calling the `(aop)` function will start the interpreter's read-eval-print loop. The prompt will be `<AGENT>`, representing the fact that you are "in the context of" a predefined agent named agent. This agent has no beliefs or commitment rules built in. It is useful mainly as a place from which to interact with other agents running under the interpreter.

5 Defining an Agent

Each agent has a distinct name. An agent program includes two files. One, agent-name. aop contains the AOP program for the agent. The other, agent-name. lsp, contains support functions written in Lisp which implement the agent's primitive actions. Currently, all of these functions should be written to expect one more argument than the arity of their call as written in commitment rules. This extra argument should come first. It corresponds to the data structure which holds the agent who is performing the action. This is useful for implementing private actions which modify the agent's own beliefs or commitments.

A sample agent named joetriv has been provided in the `<a0>` directory. You can examine the files j oetriv. aop and joetriv. `lsp` to get an idea of the format of this information. The next section describes how to load an agent such as joetriv into the environment.

6 Loading an Agent

To load both the . aop and the `.lsp` files into the current environment, say of an agent named joetriv, type `load` joetriv to the `<AGENT>` prompt. You can then type joetriv to the prompt to "enter" joetriv's context. The prompt will change to reflect the fact that you are now within joetriv. This sets the global variable *current-agent* to the internal data structure associated with joetriv, so that many of the commands understood by the AGENT0 main loop will function with respect to joetriv. For example, you could type state (now (alive john)) to assert a fact into joetriv's beliefs. Or you could type inform agent ((+ now `(* 10` m)) `(foo a b))` to send an inform message from joetriv to agent which says that the proposition `(foo a b)` becomes true ten minutes from the time the message is sent.

7 Commands

A number of commands are understood by the AGENT0 main loop. These include:

q or quit or exit	leave the AOP function
run	begin asynchronous mode (run ticks continuously)
walk or stop	return to synchronous mode
<return>	on a line by itself, runs one tick in synchronous mode
now	print the current time in 24-hour format
load <agent>	loads files <agent>. aop and <agent>. lsp
go <agent>	make <agent> the *current-agent*
inform <agent> (TIME PROP)	*current -agent * informs <agent > of <fact>
request <agent> <act>	*current-agent* requests <agent> to perform <act>
beliefs	list all beliefs of *current-agent*
cmtrules	list all commitment rules of *current-agent*
incoming	list all incoming messages of *current-agent* (to be processed at beginning of next tick)
bel? (TIME PROP)	tells whether *current-agent* believes PROP is true at TIME (now can be used as a TIME to indicate the current time. Functions of now can also be used, such as (+ now (* 2 m)) for 2 minutes later than the current time)
state (TIME PROP)	assert this fact as a belief of *current -agent*
clrbels	remove all beliefs of *current-agent*
cmtrule [, , ,]	add a new commit-rule
showmsgs	turn on display of messages as they are sent
noshowmsgs	turn off display of messages as they are sent
<any-lisp-form>	let Lisp evaluate <form>

8 Beliefs

An agent's beliefs consist of a set of facts. Each fact is associated with a predicate and a fact-status list. This list describes the truth-values of the fact over time. An example of a fact-status list would be the following:

```
[.. U] [10:00:00 T] [Sun Nov 24 12:00:00 F]
```

This indicates that the agent believes the predicate of the fact in question became true at 10am today, and will become false at 12 noon on Nov 24 of this year. If you ask this agent whether she believes the fact at some time between these two, she will answer t; if you ask about some time after this range, she will answer nil; if you ask about a time before this range, she will answer **nil** both to queries about the fact and to queries about its negation. This is because the truth value up until 10am today is "unknown".

As described in Yoav's paper, AGENT0 agents believe any new fact they *are* told. Facts, here, are really statements about the status of a proposition at a particular time. These are parsed as statements of the form (TIME (PRED args)), internally called fact-patterns. Each typically will give rise to a new fact-status record on the fact with the same proposition as was passed in the message, time equal to TIME (which must be bound), and truth-value taken from the presence or absence of a not before the PRED. Some special statements are allowed in place of TIME here; see Section 12 below for details.

9 Commitments

A commitment is just a particular kind of proposition which is stored in an agent's beliefs database. An agent can come to have a commitment either as a result of firing a commitment rule, triggered by some incoming request message, by taking the action of committing to do some other action, or by simply asserting the commitment into his beliefs. A commitment can be unrequested by the agent to whom it is made. An agent can commit to herself; this is considered a "choice".

Each tick, an agent performs all of her commitments which have matured. A commitment it is unasserted from the beliefs database (i.e., asserted with truth-value FALSE), at the time the commitment is performed. This gives the agent a record of the time at which she actually carried out the commitment.

10 Capabilities

The AGENT0 specification calls for a database of capabilities to be checked against automatically each time an agent considers making a commitment. This current implementation of AGENT0 does not include any capability database or checking of such a database.

11 Messages

Agents can send each other REQUEST and INFORM messages. The syntax is as follows:

```
<AGENT> inform joetriv (now (i-am-cool))

<AGENT> request joetriv (do (+ now (* 5 m)) (becool))
```

The "now" in the message refers to the moment when the message was sent, not the moment when it was received.

The next version of this interpreter will include support for a standardized message-passing format in terms of files or UNIX sockets, so that agents running under this implementation can communicate with agents running under other implementations or on other machines.

12 Time

AGENT0 can be run in either synchronous or asynchronous mode. The TIMEGRAIN given in each agent program is used to determine the frequency of simulating that agent. The guarantee made in general of AGENT0 programs is that they keep their commitments by the time they mature. In this implementation, the agents always perform their commitments during the' first tick which is begun after those commitments mature. If the simulation is being run in discrete, user-prompted tick cycles, as it will be when the user wants to interactively send messages and inspect agents. then it is up to the user to hit return often enough that the agents meet their commitments in a timely manner. To run the simulator in an asynchronous mode, type run to the prompt.. An asterisk will appear before the prompt to indicate that ticks are being run. To return to the synchronous mode, type q or quit to the prompt.

I have chosen to print real times to varying degrees of specificity, depending on how far the time is from the current time. Thus, if the time is in the format HH:MM: SS, it is some time during the current day, printed in 24-hour time format. If the time is within the next week, and in the same month as the current day, the day of the week is given with the time for display purposes. If it is not, but it is within the same year, then all but the year are shown. Otherwise, a full display of DAY MONTH DATE HH :MM: SS YEAR is shown.

When using the Lisp syntax, users specify times by using Lisp functions. These functions will operate on time in the internal (integer) format. So (+ ?time 30) is a time 30 seconds later than the time to which ?time is bound. Several useful constants are defined to make it easier to specify relative times. These include m for one minute, h for one hour, day, week, and yr. I do not yet provide functions for specifying absolute times, but I plan to soon. For now, you can generate a universal time integer by using the Lisp function (encode-universal-time args). Its syntax is as follows:

```
----------------------------------------------------------------------
```

ENCODE-UNIVERSAL-TIME [Function] Args: (second minute hour date month year &optional (timezone -9))

The correct time zone to use on the West Coast of the United States is 8.

Users and agent programs can also use the word now to refer to the current time. Thus, statements such as state ((+ now (* 2 m)) (i-am-cool)) are acceptable commands to the <AGENT> prompt.

13 Examples

For practice, try running through a few examples in AGENT0 to get the hang of using the interpreter and watching messages. This is an example of an interaction that exercises a subset of the part of AGENT0 which currently works.

```
<AGENT> load joetriv
Defining agent "JOETRIV"
Parsing file aop/joetriv.aop now

LOADED

<AGENT> inform joetriv (100 (on a b))  ; This means at time 100, (on a b)
JOETRIV will be informed next tick.

<JOETRIV> beliefs

<JOETRIV>                ; (press return to run a tick)

<JOETRIV> beliefs

(ON A B)   [.. U] [100 T]
<JOETRIV> state (200 (not (on a b)))
Belief added.

<JOETRIV> beliefs
(ON A B)   [.. U] [100 T] [200 F]

<JOETRIV> bel? (150 (on a b))
T
```

```
<JOETRIV> bel? (-500 (on a b))
NIL

<JOETRIV> bel? (-500 (not (on a b)))
NIL

<JOETRIV> agent

<AGENT> inform joetriv (now (i-am-cool))
JOETRIV will be informed next tick.

<AGENT>

<AGENT> request joetriv (do (+ now m) (i-am-cool))    ; one minute from now
JOETRIV will be requested next tick.

<AGENT>

<AGENT> run

*<AGENT>            ; * indicates asynchronous mode

<just under one minute passes>

This is the cool Joe Triv Agent.

q    ; user types q to quit run-mode

<AGENT> q

<cl>
```

References

[1] Y. Shoham. Agent Oriented Programming. Technical Report STAN-CS-90-1335, Computer Science Department, Stanford University, 1990.