Agent-Oriented Programming Idioms

Jeffrey C. Schlimmer SCHLIMMER@EECS.WSU.EDU
School of Electrical Engineering and Computer Science,
Washington State University, Pullman, WA 99164-2752, U.S.A.

Abstract

Agent-oriented programming is a new paradigm for conceptualizing a computational system (Shoham, 1993), differing primarily from object-oriented programming in the inclusion of mental state in each of the processes. This paper offers two special-case versions (or idioms) of agent-oriented programming that encapsulates master-slave and peer-peer type interactions in simple protocols. These two idioms are demonstrated with a distributed version of the popular learning system ID3 (Quinlan, 1986) and with a hybrid system for studying ant behavior.

1. Introduction

New programming paradigms bring fresh perspectives and tools to computation. Shoham (1993) has identified a promising new paradigm termed “agent-oriented” programming. Similar to object-oriented programming, it offers a way of conceptualizing the role of different computational units either within the same computer or distributed across a network. This new paradigm lays the groundwork for a view of computing in which the state of each process may include mental constructions such as beliefs, preferences, capabilities, and obligations. Key to the paradigm is the identification of a vocabulary of interaction, a language of communication between processes viewed as agents. This vocabulary includes `DO` for private actions, `INFORM` to tell another agent a fact, and `REQUEST` to ask a favor of another (as well as `UNREQUEST` and `REFRAIN` to cancel a favor either initiated by oneself or another). Within this context, this paper identifies two useful, special cases of general agent-oriented programming which I will call *idioms*. It also demonstrates each of these idioms in simple, implemented systems.

2. Agent-Oriented Programming Abstractions

In its fullest sense, an agent-oriented programming system provides two basic components: (1) a formal language for describing the mental state of an agent, and (2) a programming language with agent-oriented primitives. Shoham (1993) presents a restricted but powerful formalism for characterizing time, commitment, and belief. This is paired with an interpreter for agent-oriented programs called AGENT-0. This paper builds on the latter programming language component by adding a pair of abstractions above it. Instead of providing a view of the system in terms of `INFORM`, and `REQUEST`, these idioms deal with roles (like master versus slave) and stylized actions (like `REGISTER-ANSWER` and `SELECT-PROBLEM`).

2.1 Operating system protocol

Figure 1 depicts a specific set of abstractions, both more concrete and more abstract than agent-oriented programming. It may be useful to think of these in terms of protocols. The lowest-most in the figure, and the most concrete, is based directly on an underlying operating system's primitives for process-to-process communications. (In this case, AppleEvents for the Apple Macintosh System 7 operating system. Event handlers for four specific ID's are defined corresponding to the four primitives in the next protocol.) Viewed at this level, agents are doing little more than sending labeled packets to each other.

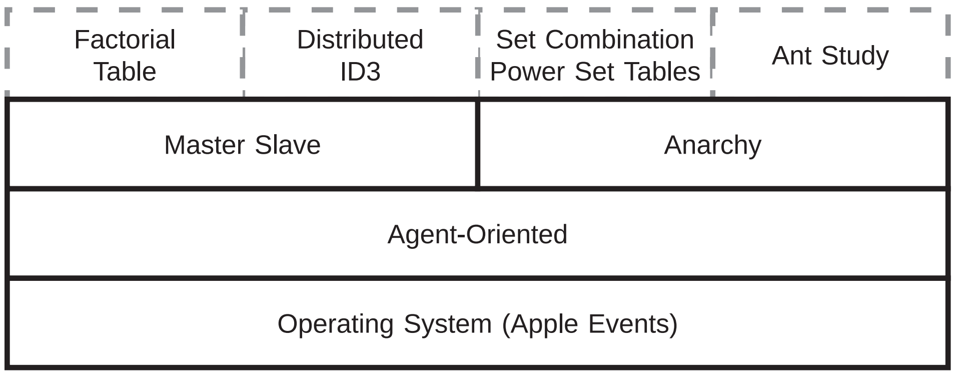

Figure 1: Levels of abstraction in agent-oriented programming. The lowest level is the most concrete. The highest level is applications.

2.2 Agent-oriented protocol

The next most abstract protocol is based directly on the agent-oriented communication vocabulary of AGENT-0. It includes the following primitive actions:

```
REQUEST-NO-WAIT!(request, target-agent)
REQUEST-WAIT-INFORM!(request, target-agent)
INFORM-NO-WAIT!(inform, target-agent)
INFORM-WAIT-REQUEST!(inform, target-agent) .
```

The `NO-WAIT!` primitives initiate asynchronous communication from an originator agent *O* to a target agent *T*; a reply is sent in due time by *T* back to *O*. The `WAIT` primitives initiate synchronous communication. `REQUEST-WAIT-INFORM!` is the more intuitive of the two, where *O* waits for an answer to a request, either information or notification that an action has been successfully completed. `INFORM-WAIT-REQUEST!` corresponds to the opposite ordering; *O* provides an answer to a previous request by *T* and implicitly accepts a request from it.

To see why both of these dual primitive actions are useful, consider an extended dialog between two cooperative agents. *O* regularly makes requests of *T*. Using the shorthand of `R` for request, `A` for acceptance, and `I` for information, the dialog between *O* and *T* would look something like ...`RAIRAIRAIR`... If *O* introduces delays before making requests of *T*, then their dialog would look like ...`RAI..RAI..RAI`... (where `".."` signifies a delay). In this case, `REQUEST-WAIT-INFORM!` would be an appropriate action so *T* is freed for other activities. Alternatively, if *T* introduces delays before fulfilling requests for *O*, then their dialog would look like ...`IRA..IRA..IRA`... Here, `INFORM-WAIT-REQUEST!` would be more appropriate because it frees *O* from synchronizing with *T* while the latter is delaying. If both introduce delays, the asynchronous `NO-WAIT!` primitives are appropriate.

When *O* and *T* reside on different computers, this constitutes distributed computing. In this situation, the agent-oriented level protocol incurs approximately 300 milliseconds of overhead for each primitive action using Ethernet connections.

2.3 Idiom protocols

Above the agent-oriented protocol are two idiomatic protocols. One is based on a master-slave relationship common in distributed computing. The other is based on a more egalitarian model; multiple peers may interact, but no agent is in authority over another. These idioms are specializations of the underlying agent-oriented generality because they provide only part of the functionality available at lower levels. They are also abstractions. The idioms provide simple methods for tasks that would otherwise require many separate operations at the agent-oriented level. These idioms represent the main contribution of the paper and are discussed at length in the following sections.

2.4 Application level

Finally, above the idioms lay applications. These are constructed using operations provided by the idioms to facilitate communication between two or more agents. A simple and a more interesting application are demonstrated for each of the idioms. For the master-slave idiom, a master uses slaves to compute a table of factorials. Another master uses slaves to evaluate alterative tests in a distributed version of the decision tree learning method ID3 (Quinlan, 1986). For the anarchy idiom, a pair of agents help each other compute tables listing set combinations and power sets. A trio of agents also collaborate to assist a person studying ant behavior. This latter demonstration uses agents with expertise in simulation, diagnosis, and statistics. These demonstrations will be revisited in more detail after discussing their idioms.

Fully commented Macintosh Common Lisp source code for each of these protocols and applications is included in on-line Appendix 1.

3. Master-Slave Idiom

A familiar computational idiom is that of a master *M* directing the work of several slaves *S*. *M* must initialize *S* by defining how tasks are to be completed. *M* must also assign new tasks to *S* and must record the results of those tasks when completed.

3.1 Agent-oriented functionality abstracted by the master-slave idiom

To initialize *S*, *M* must define `DO!` for *S*, so *S* knows how to fulfill the task requests of *M*. Perhaps the simplest way to do this is to pass a file to *S*, and ask *S* to evaluate the contents. Because of contention for network resources among numerous *S*, if *M* and *S* are on different computers, then for each *S*, the file is copied during a synchronous `REQUEST-WAIT-INFORM!` and then loaded with an asynchronous `REQUEST-NO-WAIT!`. The master-slave idiom encapsulates this functionality with the call `INITIALIZE-MASTER` which takes a code file, slaves, and a main function for *M* as arguments.

M (acting as originator) incurs minimal delays in issuing new requests; *S* (acting as target), however, incurs delays as it completes each task. Thus their dialog looks like `R..IRA..IRA..IRA`… (using our previous notation). Consequently, the bulk of the communication between *M* and *S* will be via `INFORM-WAIT-REQUEST!`.

Immediately after loading the initialization code defining `DO!`, slaves are put into an infinite `INFORM-WAIT-REQUEST!` loop. Each iteration of this loop involves registering a solution to the current problem with *M*, accepting a new problem from *M*, and using `DO!` to complete the task. The master-slave idiom expects the functions `REGISTER-ANSWER` and `SELECT-PROBLEM` to be defined by the master-slave application (at the next layer up). *S* calls these functions, and *M* executes them. *S* is initialized with a request of `:NO-REQUEST`; this value is also to be used when *M* temporarily has no tasks for *S*. The loop is terminated with the special value `:RELEASE-COMMITMENT`. Both of these special values may be returned when *M* executes `SELECT-PROBLEM` for *S*. Figure 2 summarizes the four functions in the master-slave idiom.

Function	Who Calls	Who Executes
`INITIALIZE-MASTER`	*M*	*M*
`SELECT-PROBLEM`	*S*	*M*
`DO!`	*S*	*S*
`REGISTER-ANSWER`	*S*	*M*

Figure 2: Calling versus executing the four functions in the master-slave idiom.

3.2 Filling a table of factorials

Consider a simple master-slave agent application. *M* wishes to compute a table listing several small integers and their factorials. It calls `INITIALIZE-MASTER` with: (1) a file defining a factorial function, (2) *S*, and (3) simple code to initialize the table. The master-slave idiom initializes *S* which immediately begin their loop of `IRA..IRA..IRA`... At first *S* calls `REGISTER-ANSWER` with `:NO-REQUEST` but is soon given a factorial problem to solve by `SELECT-PROBLEM`. After executing `DO!` (and computing the factorial), *S* loops and calls `REGISTER-ANSWER`, now with a value to be entered into the table. When the table is full, *M* issues `:RELEASE-COMMITMENT` when *S* calls `SELECT-PROBLEM` instead of another factorial problem. *S* exits its loop, ready to communicate with another agent. Figure 3 lists pseudo code for the master-slave factorial table application.

```
INITIALIZE-MASTER(codeFile, slaves, initTable).

REGISTER-ANSWER(problem, answer)
  If problem is :NO-REQUEST
    Then do nothing;
    Else add the answer to the table of results.

SELECT-PROBLEM()
  Find the next unanswered problem P;
  Mark P as assigned in the table of results;
  Return P.

DO!(problem)
  Compute the factorial of the problem.
```

Figure 3: Pseudo code for the master-slave factorial table application.

3.3 Distributed ID3

A more sophisticated master-slave application demonstrates repeated use of *S* and a nontrivial *M* main function. *M* wishes to construct a decision tree that accurately classifies some examples on the basis of values of a set of attributes. *M* roughly follows the algorithm of ID3 (Quinlan, 1986). To construct a tree, ID3 applies an information-theoretic evaluation function to heuristically select the most discriminating attribute. This is the root of the decision tree. ID3 then constructs a branch for each possible value of this attribute, partitions the examples according to their value for this attribute, and recursively repeats the process at each subtree until there are no more attributes to test or until all examples are of the same class. (Some learning methods use a pruning heuristic to stop tree construction when the examples cannot be reliably discriminated.) *M* builds the same tree as ID3 does, but it uses *S* to assist it.

M calls `INITIALIZE-MASTER` with: (1) a file consisting of code and the examples, (2) *S*, and (3) a main function that recursively builds decision trees. The master-slave idiom initializes *S* by loading the code (to define `DO!`) and the examples. *S* is soon assigned a task to perform. In this case, each task is an attribute to be evaluated as a candidate for a decision tree as well as the examples relevant to the decision. (Recall that subtrees are built using only part of the data.) When *S* executes `DO!` on these problems, *S* computes a quality function (entropy) for the candidate attribute. When every candidate attribute for a tree has been evaluated, *M* issues `:NO-REQUEST` to *S*, picks the best attribute, splits the examples into partitions as ID3 does, and recurses on each new partition. As it processes each new subtree, *M* constructs a new empty table and issues problems to *S*. When the entire tree is constructed, *M* issues `:RELEASE-COMMITMENT` via `SELECT-PROBLEM` to *S*. Figure 4 lists pseudo code for the master-slave distributed ID3 application.

```
INITIALIZE-MASTER(codeDataFile, slaves, BuildTree).

BuildTree(examples, attributes)
  If examples all have the same class
    Then return a leaf
    Else
      Compute BestAttribute(examples, attributes);
      Make a new decision node that tests BestAttribute;
      Partition examples according to their value for BestAttribute;
      Recursively BuildTree for each partition.

BestAttribute(examples, attributes).
  Initialize a new table.
  Define REGISTER-ANSWER
    to update the table of entropy scores for each attribute;
  Define SELECT-PROBLEM
    to return the next unevaluated attribute;
  Wait for slaves to fill out the table;
  Return the best attribute from the table.

DO!(examples, attribute)
  Compute the entropy score for the attribute on the examples.
```

Figure 4: Pseudo code for the master-slave distributed ID3. The functions `BuildTree` and `BestAttribute` are executed by *M*. Only the latter is significantly different from a serial implementation.

To get a feel for the master-slave idiom in operation, the master-slave distributed ID3 application was tested with varying numbers of *S* on the Wisconsin Breast Cancer Database

(January 8, 1991) available from the UCI machine learning repository. This database was obtained from the University of Wisconsin Hospitals, Madison from Dr. William H. Wolberg. It has 9 attributes describing 699 examples. On this data set, ID3 performs 164 evaluations of attributes as possible decisions for internal nodes of the decision tree. Hence, *M* has 164 problems to assign to *S*. When there is only one *S*, it must calculate all 164 of these attribute evaluations. For four *S*, the work is reduced to an average of 41 attribute evaluations per *S*. Figure 5 depicts the total computation of *M* as a function of the number of *S* for a Macintosh Common Lisp implementation running on an assortment of Apple Macintosh Quadra 610, 660av, and 950 computers (basically Motorola 68040-based microcomputers). The time apparently decreases exponentially with increasing *S*. No special effort was made to optimize this code or specially configure the *S* computers.

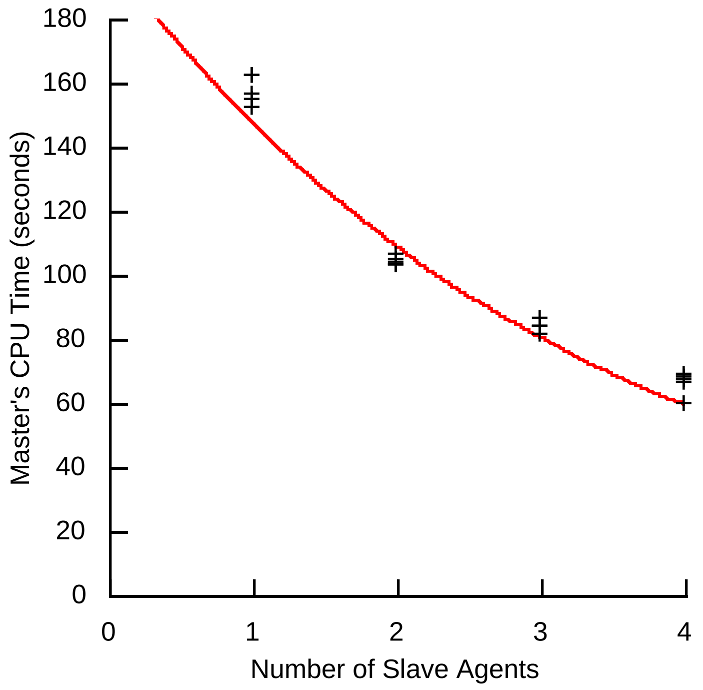

Figure 5: Execution time of the master ID3 agent as a function of the number of slave agents assisting. Each trial repeated five times. The data is fit with an exponential curve of $200 \times e^{-0.3 \times \text{slaves}}$.

4. Anarchy Idiom

A more egalitarian model for agent cooperation does not place any agent in an authoritative role over others. Rather, two agents *A* and *B* may trade requests from time to time. Lacking any central control, this anarchy of agents is appropriate when agents have different capabilities and are willing to cooperate.

4.1 Agent-oriented functionality abstracted by the anarchy idiom

If *A* makes a request of *B*, then *A* must be able to initialize *B*. As with the master-slave idiom, this is done by specifying a file of code and/or data which is transferred to *B* by sending a `REQUEST-WAIT-INFORM!` and then loaded by *B* as a result of a `REQUEST-NO-WAIT!`

message sent on behalf of *A* by the anarchy idiom. Because there could be a number of *B* that have the capability needed by *A*'s request, *A* will specify a target agent only by the class of capabilities it has. The anarchy idiom will ensure that once *B* has been identified as having a capability, *A* will be able to refer to *B* by its capability class rather than by name. This is implemented in the anarchy idiom by caching for each *A* the capability class and initialization status for each agent *B* it has sent requests to. Lastly, *A* needs to be able to send a specific request, and this is implemented as a `REQUEST-WAIT-INFORM!` with a generous patience parameter.

All of these operations are encapsulated in a single `ASK-AGENT` function which takes two parameters: (1) a capability class to identify *B*, and (2) a specific request. The request may either be `:INITIALIZATION` with the name of an initialization file, or it may be an arbitrary function call. The anarchy idiom ensures that the request is sent to a *B* with the right capability and that *B* is appropriately initialized.

4.2 Filling a table of set combinations and power sets

Consider a simple anarchy application. *A* wishes to compute a table listing several small integers and the combinations of that size for a set, but *A* wants to avoid making any overly large entries. *B* wishes to compute a table listing the power set for prefixes of a small set. It too wishes to avoid adding any overly large entries. *A* has set operation capabilities and can compute combinations and power sets. *B* has combinatorial calculation capabilities and can estimate the size of a set of combinations and a power set.

A starts filling its set combination table, but before making the first entry, it wishes to know if it will be too large. It calls `ASK-AGENT` with: (1) the capability class `:MATH`, and (2) a specific request to calculate the number of combinations of a specific size of its specific set. The anarchy idiom prompts *A* to initially select a *B* with `:MATH` capability (and then the anarchy idiom caches this information). A `REQUEST-WAIT-INFORM!` is sent to *B* on *A*'s behalf; *B* suspends its processing, calculates the request, and an inform of the result is sent back to *A*. If the result is not overly large, *A* will go ahead and compute all the combinations of that size and add them to its table.

Meanwhile, *B* starts filling its power set table. As does *A*, it first calculates how big the entry will be. *B* has this capability, so it does the work itself. If the entry is not too large, *B* will want to compute the power set. To do this it calls `ASK-AGENT` with: (1) the capability class `:SET`, and (2) a specific request to calculate a power set. The anarchy idiom treats this request exactly as it did *A*'s request to *B* (only with the roles reversed). Figures 6 and 7 list the pseudo code for *A* and *B*, respectively.

```
setCombinationTable(set)
  Let table be empty;
  For k from 0 to the length of set loop
    ASK-AGENT(:MATH, Number of combinations of set of size k);
    If not too big
      Then compute combinations of set of size k and add to table;
      Else exit.
  Return table.
```

Figure 6: Pseudo code for an anarchy set combination table application.

```
powerSetTable(set)
  Let table be empty;
  For k from 0 to the length of set loop
    Compute size of power set of prefix of set of size k
    If not too big
      Then
        ASK-AGENT(:SET, power set of prefix of set of size k)
        Add it to table
      Else exit.
  Return table.
```

Figure 7: Pseudo code for an anarchy power set table application.

4.3 Distributed expertise for ant behavior

A more sophisticated example of an anarchy agent application demonstrates multiple agents with different capabilities and the use of initialization. There are three agents. *A* is testing hypotheses about the efficiency of ant food gathering behavior. It has simulation capabilities provided by Agar, a general behavioral simulator with modules for simulating the low-level, mechanistic behavior of ants, slugs, spiders, and mice by Michael Travers. *A* wishes to use a high-quality random number generator to initialize the simulation and to evaluate the correlation between simulation parameters and the time it takes for the ants to complete the food gathering task. Figure 8 depicts a screen snapshot of *A*'s simulation.

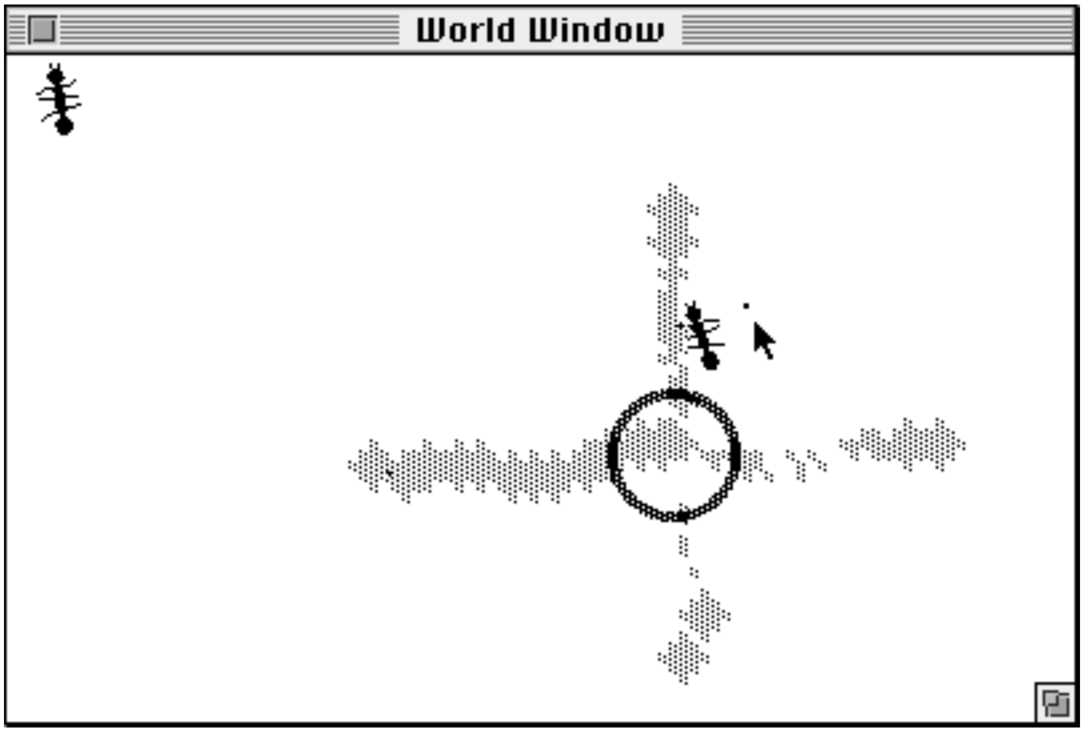

Figure 8: Screen snapshot of the ant simulation application Agar. The nest is the dark circle just off center. The gray areas are scent trails that fade over time. The arrow cursor is pointing at a piece of food close to one of the ants.

B is not otherwise engaged but has capabilities in diagnosis provided by TMycin, a miniature version of EMycin written by Gordon Novak. *C* is also not pursuing its own goals. It has statistical and numeric capabilities provided by CL Math, a numerical package written by Gerald Roylance consisting of over 250 functions drawn from various mathematical references.

A is repeatedly accepting single-variable hypotheses from its user about significant factors in the ant's food gathering efficiency. The user specifies one of nine independent variables, a minimum and maximum value, the number of samples in that interval, and the number of repetitions. *A* selects a default value for the remaining eight simulation parameters. The simulation is started with nesting ants placed in the nest location and foraging ants placed arbitrarily in the world. To place each unit of food, *A* uses *C*'s high-quality random number generator with a normal distribution centered on the center of the world [`ASK-AGENT(:STATISTICS, normalRandomNumber())`]. *A* runs each simulation until the ants locate all the food. Figure 9 lists the simulation parameters and their default values.

Simulation Parameter	Default Value
N of foraging ants	1
N of nesting ants	1
N food	10
Long range food sensor	150
Short range food sensor	75
Food in hand sensor	10
Nest proximity sensor	25
Nest location (X)	250
Nest location (Y)	150

Figure 9: Simulation parameters for Agar's ant simulation. The simulated world is 400 by 250 units.

After running the specified number of simulations, *A* uses *C*'s regression function to compute a correlation between the independent variable and the time steps for the ants to locate the food [`ASK-AGENT(:STATISTICS, regressionCorrelation(independentSamples, dependentSamples, nTerms))`]. If the correlation is high, *A* reports the result and waits for the user's next hypothesis. If the correlation is low, *A* uses *B*'s diagnosis capability to try to explain why the hypothesis is rejected. *A* first initializes *B* by sending it a knowledge base for this domain [`ASK-AGENT(:DIAGNOSIS, :INITIALIZE "ant.kb")`]. The anarchy idiom ensures that *B* will only copy and load the knowledge base once. *A* then asks *B* to explain the rejection of the hypothesis [`ASK-AGENT(:DIAGNOSIS, doConsult(ANT,…))`]. The simple knowledge base sent to *B* includes 20 possible deficiencies with the simulation study, some of which are general statistical errors. It has 87 rules that model the limitations of the ant simulator, pointing out where the simulation is unreliable and when there is generally insufficient data for reliable conclusions. Two rules in the knowledge base request information from *C* about the strength of the correlation and its significance [`ASK-AGENT(:STATISTICS, regressionSignificance(independentSamples, dependentSamples, nTerms))`]. This is a situation where *B* is asking *C* a question on behalf of *A*. Figure 10 lists sample output from *A* regarding a rejected hypothesis.

```
? (hypothesis ‘foragers :min 10 :max 20)
;;; Not Confirmed: FORAGERS -> TIME-TO-FIND-ALL-FOOD [0.67]
The conclusions for AGAR-HYPOTHESES338 are as follows:
TOO-FEW-ANTS: NO (1.00)
TOO-MUCH-FOOD: NO (0.90)
INSUFFICIENT-LONG-RANGE-FOOD-SENSOR: NO (1.00)
UNREALISTIC-LONG-RANGE-FOOD-SENSOR: NO (1.00)
INSUFFICIENT-SHORT-RANGE-FOOD-SENSOR: NO (1.00)
UNREALISTIC-SHORT-RANGE-FOOD-SENSOR: NO (1.00)
NEST-INAPPROPRIATELY-SITUATED: NO (1.00)
INDEPENDENT-VARIABLE-SET-TOO-LOW: NO (1.00)
INDEPENDENT-VARIABLE-SET-TOO-HIGH: YES (1.00)
INSUFFICIENT-FOOD-AT-HAND-SENSOR: NO (1.00)
UNREALISTIC-FOOD-AT-HAND-SENSOR: NO (1.00)
INSUFFICIENT-NEST-SENSOR: NO (1.00)
UNREALISTIC-NEST-SENSOR: NO (1.00)
GENUINELY-UNCORRELATED-DATA: YES (0.50)
INSUFFICIENT-DATA: YES (0.90)
INSUFFICIENT-SAMPLING-OF-INDEPENDENT-VARIABLE: NO (1.00)
MISMATCHED-DEPENDENT-AND-INDEPENDENT-SAMPLES: NO (1.00)
RANDOM-VARIATION: NO (0.26)
TOO-FEW-MODEL-TERMS: YES (0.60)
WEAK-CORRELATION: YES (0.26)
:FALSE
?
```

Figure 10: Sample information displayed by the simulation agent retrieved from the statistics and diagnostic agents. The simulation agent lists factors that may explain why the experimental hypothesis was not confirmed. (I.e., why the number of foraging ants is not directly correlated with the speed at which food is gathered.) The simulation agent suggests that the maximum value for the number of foragers is set too high and that more samples are needed.

5. Discussion

The main contribution of this paper is the identification of two programming idioms that are more abstract than the basic agent-oriented level. They are specialized to a particular type of interaction between agents. Both offer the advantages associated with an abstract programming language, namely reduced tedium and likelihood of programmer error.

The two idioms are complementary. The anarchy idiom could be used to implement the master-slave idiom, but the result would be extremely slow. If the master were to send tasks to each slave via the synchronous ASK-AGENT, it would lose the advantages of parallelism in a distributed set of slaves. (The ID3 master would be waiting for slaves to evaluate attributes in serial.) Similarly, if the master-slave idiom were used to implement the anarchy idiom, an agent would be unable to serve multiple others looking for a capability class agent. (The statistics agent would be tied to either the simulation agent or the diagnosis agent.) These are not the only idioms. There are likely to be other useful specializations that abstract away from the (powerful) details of agent-oriented programming.

In terms of parallel computing, these two idioms represent coarse-grained parallelism and minimally-distributed computing in a (barely) heterogenous environment. If efficiency is a concern, the master-slave idiom would be useful when the problem naturally decomposes into problems of enough size that the relatively slow communication overhead is insignificant.

A secondary contribution of this paper is the demonstration of a distributed version of the popular learning method ID3 and a novel integration of a simulation agent, a statistics agent, and a diagnosis agent. The agent-oriented programming paradigm naturally leads to novel thinking and identifies innovative heterogenous systems.

Acknowledgments

I wish to thank the machine learning research group at Washington State University for their suggestions, encouragement, and comments on an earlier draft of this paper. John Ash wrote part of the master-slave factorial table application. Thanks also to William H. Wolberg for the breast cancer data, to Michael Travers for Agar, to Gordon Novak for TMycin, and to Gerald Roylance for CL Math. I would also like to thank the WSU EECS technical support group for their help in setting up our network and the Cambridge Macintosh Common Lisp group for a fine product. This research is funded by Digital Equipment Corporation, NASA, and the National Science Foundation.

References

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81–106.

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60, 51–92.