Designing Cooperating Agents for Office Automation

P. G. Chander, R. Shinghal, and T. Radhakrishnan
Department of Computer Science, Concordia University, Montreal
{gokul, shinghal, krishnan}@cs.concordia.ca

Abstract
Office automation requires controlled coordination between information management and collaborative problem solving. Often, this entails the cooperation of agents executing diverse tasks in an office environment. Designing cooperating agents for office automation, however, is non-trivial: issues such as agent heterogeneity, communication protocols, context sensitivity, task coordination, and concurrency present considerable challenges. In this paper, we describe the design and implementation of a cooperating agent framework called GAME for office automation. More specifically, we focus on how GAME sets up communication between its agents, and achieves cooperation by transferring tasks between its agents, whenever necessary, to avoid duplication of work by an agent.

Keywords: Software Agents, Expert Systems, Office Automation.

1. Introduction
Recent years have witnessed an ever increasing demand for computer support in office environments. Consequently, the number of task-support tools and utilities have proliferated in meeting this demand. However, most of these support tools tend to work in a “stand alone” fashion: that is, the tools require explicit user invocation and customization. The lack of interoperability (exchange of information and services) between tools can be cumbersome because user activities for accomplishing a task often require auxiliary help from utilities supporting other tasks.

Office automation can be broadly categorized into two major activities (Hewitt 1986; Lai and Malone 1988a; Malone et al. 1995): information management (IM), and collaborative problem solving (CPS). Information management requires sharing, generating, transmitting, and using information pertinent to office functions. Collaborative problem solving occurs through meetings and initiates cooperation between office personnel. IM and CPS are not exclusive activities. Rather, they characterize a spectrum of overlapping tasks: that is, a given task execution can require cooperation between tools designated for accomplishing other tasks. Thus, offices are task oriented domains (Hewitt 1986; Rosenschein and Zlotkin 1994).

An agent is a software module built to monitor, assist, and act on behalf of a user in order to accomplish a task. An agent is designed to interoperate with other co-existing agents, and is perceived to be intelligent, whenever it can mimic a personal assistant (Gensereth and Ketchpel 1994; Maes 1994; Mitchell et al. 1994).

In this paper, we describe a cooperating agent framework called GAME (Goal-oriented Agent Managed Environment) for office automation. Motivation for GAME arises from the following observations: (1) the isolated functionality of most of the existing tools for office environments is unsatisfactory; (2) the drive for inter-operability between tools has increased; (3) expert system technology can be better exploited for agent construction; and (4) the existing agents impose too much constraints on the user, or lack proper coordination to prevent duplication of work (Lai and Malone 1988b; Pollock 1988; Gensereth and Ketchpel 1994; Maes 1994; Chander et al. 1997; Chander 1996).

This paper is organized as follows. In section 2, we describe the GAME architecture, and how communication is set up between its agents. In section 3, we discuss the design issues involved in developing such a framework and its agents. Section 4 reports on the current implementation status of GAME and our observations. Concluding remarks appear in Section 5.

2. GAME Architecture
We believe that an agent acting as a personal assistant to automate routine office duties should be able to exhibit the following elements of functionality (Lai and Malone 1988b; Maes 1994; Etzioni and Weld 1994; Chander 1997b):

1. autonomous mode: provide services as a stand alone program for the task for which it has been designed;

2. cooperative mode: provide services to other agents on a mutual give-and-take basis;
3. **expert mode**: use a knowledge base to apply rules selectively for intelligent response; and
4. **learning mode**: learn through user and (co-existing) agent interactions to adapt itself better to the user.

Agents in an environment can be partitioned into two disjoint sets: **homogeneous agents** and **heterogeneous agents**. Two agents \( A_1 \) and \( A_2 \) are homogeneous, iff their task requirements are identical; otherwise, they are heterogeneous. Distinguishing between homogeneous and heterogeneous agents at the task, rather than their implementation level, provides flexibility to a designer in choosing methods and techniques for agent construction.

The design philosophy of GAME shares the contemporary belief that communication between agents automating a diverse set of tasks can significantly improve the overall productivity of an environment (Hendler et al. 1991; Edmonds et al. 1994). Communication between agents take place through protocols that define the interfaces and service definitions of an agent to facilitate their utilization by another agent. Clearly, for inter-agent communication and interaction, the list of agents in an environment, the nature of their tasks, and the protocols for communication need to be maintained. Such a maintenance and coordination module is called an **agent registry**.

**Definition 1 (Agent Registry)** An agent registry records information on the set of agents, their types, and their protocols in order to coordinate inter-agent communication in an environment.

GAME agents must enroll with the registry in order to communicate with other agents in the domain; this process is called **registering**. After every successful registration, the registry spawns a concurrently running **service component** to service requests from the registered agent. The registry also maintains information about each registered agent in a data structure called the **coordination structure**. The coordination structure records an agent's identifier, location, and other information pertinent to establishing communication with that agent. The registry returns this structure after establishing communication between two agents.

**Definition 2 (Coordination Structure)** The coordination structure in GAME is a 4-tuple \( \langle A, L, S, P \rangle \), where \( A \) is the agent identifier, \( L \) is the set of communication protocols supported by \( A \), \( S \) is the set of services offered by \( A \), and \( P \) is the port (a communications object) to which other agents must send their messages.

The absence of an agent's coordination structure in the registry simply indicates that the agent is (possibly) working in an autonomous mode and cannot be communicated with.

The GAME architecture is shown in Figure 1. When two agents wish to communicate with each other, then each agent should receive the other agent's coordination structure. However, each agent must know of the other agent's intention to communicate and should agree to do so prior to establishing communication. Figure 2 shows the protocol to set up communication between two agents. A number \( i \) on a line in Figure 1 corresponds to the description in step \( i \) of the protocol described in Figure 2.

In order to mimic a personal assistant, an agent should maintain certain **contexts**: for example, it should keep track of recent user commands, and the types of messages received from the registry and other agents. In GAME, context represents a set of object properties and their range. Contexts can be maintained for objects that are user manipulatable (such as a button in the user interface) as well as objects that are not visible to the user (such as received messages).

**Definition 3 (Context of an Object)** The context of an object represents a set of object properties and the range of values for each property. The current context of an object with respect to a property is the value represented by that property.

For example, let **Message** be an object that is used to represent messages received by an agent. One of its properties called **State** can be used to record the state of the last received message. Thus, if an OK message was last received, then the **State** property of **Message** would be set to a value, say, "Okay Reply." This (string) value represents the current **Message** context for property **State**. The use of current object contexts will be apparent in the next section, which describes how an agent can use them to intelligently
respond to user commands.

GAME agents are said to be goal-oriented because their behavior is directed by their current set of goals; goals can be set for (sub)task accomplishment, in response to user commands, and in response to messages received from the registry or from other agents.

3. GAME Design Issues

The GAME framework delegates a set of agents to each user, which can interact among themselves and with other user agents. Currently, GAME is designed to accommodate three frequently executed tasks: email management, interactive point-to-point talk, and scheduling meetings. Owing to space constraints, we focus only on the email and the talk agents in this paper. A brief description of their requirements appears below. As GAME is an evolving framework, additional features will be added upon user feedback.

An email agent should perform several services such as automatically classifying emails into folders, reminding users of emails that require response, asynchronously monitoring emails from specific users, reporting on status of emails in a folder and suggesting appropriate action to be taken on each, and providing automated responses for selected emails. The email agent should also make use of the services of an underlying email package available, whenever possible.

The talk agent was inspired by the UNIX talk utility. However, a talk agent in our framework is meant for conversations that span more than one session and complement face-to-face communications. For example, it should record and maintain talk transcripts between users and provide cross references from previous transcripts (for continuity over previous sessions) during the current talk session. Interestingly, the talk agent also exhibits potential for telephone call processing, but its discussion is beyond the scope of this paper.

The crucial aspect in GAME is how the talk and the email agents cooperate with each other whenever one agent detects an event that requires the services of the other. For instance, the talk agent detecting user actions or keywords in a talk session for transmitting documents from one user to another should automatically communicate with the email agent to do so.

Designing an interacting, heterogeneous agent environment presents considerable challenges from a design as well as an implementation perspective. Several design issues that arose while developing this framework are outlined below.

1. Design Methodology GAME adopts an object oriented design and also uses an underlying knowledge base containing a set of facts and rules to control the behavior of its agents. Object oriented design allows reuse of code even when developing agents that are heterogeneous: for example, an abstract communication interface provides all communication services. By inheriting and overriding a message response function in this class, agents selectively respond to messages appropriate for their tasks. In addition, messages and system-call interfaces when implemented as classes make it easy to change message formats and to improve portability.

2. World, Domain and User Knowledge Base An office typically functions as part of an organization, and an user works in an office. Thus, operational knowledge of an office can be abstracted at three levels of generality: world, domain, and user. The world knowledge base encodes facts and rules that pertain to a given organization, while the domain knowledge base encodes facts and rules that pertain to a given office environment. An agent in a domain applies a set of domain specific rules when trying to satisfy user request in a domain as long as that request does not violate any constraints imposed by the world knowledge. The user knowledge base should be generated by an agent in its learning mode. An example of world and domain knowledge base applicable to an email agent is shown in Figure 3.

3. Embedding Rules Though expert system shells such as CLIPS (Giarratano and Riley 1993) can allow ease of knowledge encoding, embedding the shell in a program can lead to performance degradation be-
The world rules for the email agent would set up the maximum size of emails that will be buffered, policy to follow if this buffer overflows, etc. A sample set of rules in allotting disk quota are shown below:

\[
\begin{align*}
&\text{GRAD}(x) \rightarrow \text{LIMIT}(x, 1\text{Meg}) \\
&\text{FACULTY}(x) \rightarrow \text{LIMIT}(x, 10\text{Meg}) \\
&\text{FACULTY}(x) \land \text{LIMIT}(x, \text{Limit}) \land \text{SIZE}(x) > \text{Limit} \rightarrow \text{WARN}(x) \\
&\text{GRAD}(x) \land \text{LIMIT}(x, \text{Limit}) \land \text{SIZE}(x) > \text{Limit} \rightarrow \text{BOUNCE-EMAILS}(x)
\end{align*}
\]

The domain rules would be based upon the package that provides specific email facilities. Below, we assume a UNIX domain containing the MH mail package.

\[
\begin{align*}
&\lnot\text{SCANFORMAT}(x) \land \text{GRAD}(x) \rightarrow \text{USE-FORMAT('scan.default')} \\
&\lnot\text{SCANFORMAT}(x) \land \text{FACULTY}(x) \rightarrow \text{USE-FORMAT('scan.timely')} \\
\end{align*}
\]

Figure 3: Illustrating world and domain knowledge bases for an email agent.

cause the entire shell needs to be invoked even to run a small set of rules (Giarratano 1993). Performance degradation becomes pronounced when the number of rules applicable at any time is small, but the number of times such small rule sets should be applied is large. The overall system also becomes larger and complicates maintenance. In an environment where a quick agent response is desirable, it is prudent to embed rules directly in those modules that require their application. GAME follows this approach. In such a case, however, rule application depends upon their representation.

4. Encapsulating Rules In GAME, rules are encapsulated by classes, which are instantiated whenever a set of rules should be applied. The domain rules inherit from the class encapsulating the world knowledge base. Thus, world facts, constraints, and rules are automatically inherited by domain rule objects. However, domain rule objects cannot override any fact or rule in the world object. This restriction can be enforced by making every fact, constraint and rule in the world class immutable. However, rule application should take into account recent user action(s) and the status of the received messages. For example, a rule that transmits talk text should not do if the talk agent at the other end has closed its connection.

5. Context Sensitivity The response of an agent to user commands should be sensitive to the current contexts of certain objects. For example, the operation of the talk agent is affected by the current context of the Talk button object in its user interface and the current context of the Message object that records the status of received messages. This is portrayed in Figure 4 that shows some of the rules in the talk agent using these object contexts.\(^1\) The rules are written in English using if...then notation for ease in understanding.

\(^1\)In expert system terminology, object contexts refer to initial evidence and generated intermediate hypotheses.

<table>
<thead>
<tr>
<th>No.</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>if User is currently talking, and A talk-request comes in, then Record request (to show later) and reply that Request is denied.</td>
</tr>
<tr>
<td>2.</td>
<td>if Talk is in Auto-respond mode, and a talk-text comes in, then Update transcript pad, and Send appropriate auto-response.</td>
</tr>
<tr>
<td>3.</td>
<td>if User enters talk text, or clicks Talk Button, and Talk Context is Auto-respond, then Reset State to Talking, and Send any talk text entered.</td>
</tr>
<tr>
<td>4.</td>
<td>if A talk request message comes in, and Talk Context is Send-Request, then Reset Talk Context to Talk-Request, and Tell user of incoming request.</td>
</tr>
<tr>
<td>5.</td>
<td>if Talk Context is Talk-request, and User clicks on Talk-Button, then Reset Talk Context to Talking, and Send Talk-granted to received request.</td>
</tr>
<tr>
<td>6.</td>
<td>if A talk text arrives, and Talk Context is Talking, then Accept text and display on transcript pad.</td>
</tr>
</tbody>
</table>

Figure 4: Illustrating how context sensitive rule application can change the response of the talk agent.

The context of a received message can alter the context of the talk button object and hence its subsequent response to its clicking. For instance, the default response of the talk button is “Send a talk request.” However, receiving a talk request changes this context into “Accepting an incoming talk request.” Once talk is established, the state of the talk button is set to “Talking” until one of the users ends the talk session. As long the context of the talk button is “Talking”, other incoming talk requests are ignored. A user can also choose to “Auto respond” from the agent’s action menu during a talk session. This sets the talk context to “Auto response” during which the agent sends out an automatic (currently fixed) reply to a talk text received from the other end. When the user returns and clicks on the talk button, or types talk text to be sent, the “Auto response” mode is is cancelled, and the talk context is reset to “Talking.”

In general, a set of domain specific rules should be applied before responding to a user command. A rule class instantiation is given the current contexts of certain objects, and rule application depends upon the values contained in these contexts. In this sense, the function of a user interface object is said to be ‘dynamic’ (that is, its function is determined by rule application), rather than 'static' (always executing the same function). Context sensitive user interface objects cre-
3. Deadlocks can occur between service components when locking shared objects during routing. (We elaborate more on this in the next section.)

Whenever concurrency exists and resources are shared, specific synchronization mechanisms are needed if producer-consumer or multiple update situations are detected. If the updates are not properly synchronized, it can lead to incorrect operation.

Action components of an agent represent concurrency within an agent (that is, intra-agent concurrency). Agents of a user can also run concurrently (inter-agent concurrency) and if they share resources they must be controlled. Inter-agent concurrency becomes an issue only when learning is implemented: for example, when different agents generate or modify a shared user rule base. Currently, GAME design does not let agents share any resources (except, if they are “read only”), and thus avoids inter-agent concurrency.

6. Robustness and Query Answering Heterogeneous agent communication protocols often entail service specifications, when one agent requires the services of another. However, an agent handling such service requests should be robust: for example, a message asking for a non-existent service should not make the receiving agent crash. To reduce communication costs, the services of an agent \( A \) can be recorded in its coordination structure. Other agents can then determine if \( A \) can perform their requests by directly querying the registry. Thus, the registry should be able to answer agent queries.

7. Concurrency Issues Information management in office environments should be able to cope up with the concurrency of office operations (Hewitt 1986). To support concurrent activities, a GAME agent executing a task can identify several sub-tasks that can be accomplished concurrently. Such concurrent execution units for sub-task accomplishment are called action components. For example, the email agent can concurrently let a user view emails, monitor the arrival of email from specific users, while working on a folder to suggest email classification. Concurrent action components, however, can result in unexpected agent behavior. A few instances are described below:

1. The user interface of the email agent provides a status area shared by many action components that display an appropriate message about their operation to the user; thus, this status area is a target for multiple updates from concurrent units of execution.

2. The context maintained by an agent is shared and is subject to multiple updates by action components. Consider the talk agent. If a user chooses Auto Respond in a talk session, then the talk context would be updated to “Auto-Respond.” However, an incoming talk request handled by the communication action component can overwrite this value with “Talk-Request” destroying an existing talk session, if proper concurrency controls are not in place.

3. Deadlocks can occur between service components when locking shared objects during routing. (We elaborate more on this in the next section.)

Whenever concurrency exists and resources are shared, specific synchronization mechanisms are needed if producer-consumer or multiple update situations are detected. If the updates are not properly synchronized, it can lead to incorrect operation.

8. Learning Learning arises by observing the usage patterns of the user and by information exchanged between other user agents. However, the usage patterns to record, information sharing formats between agents, and the specific machine learning features of an agent are design issues. The current version of GAME does not yet have a learning module built into any of its agents.

9. Task Coordination When a user command to an agent \( A_1 \) initiates a task that can be best accomplished by another agent \( A_2 \), then \( A_1 \) assigns the task to \( A_2 \); this is called task transfer in GAME. For example, if a user wishes to send a document to a person with whom the user has been conversing (using the talk agent), then clicking on Initiate email button on the talk agent causes a task transfer to the email agent with information about the document to send. Though task transfers and task sharing can be viewed as task coordination activities, task transfer differs from task sharing because only one agent works on a transferred task at a time. Task transfers prevent duplication of work and effort by an agent. GAME agents can also execute tasks (such as search) cooperatively through task sharing. The details, however, are beyond the scope of this paper. The reader is referred to Chander (1997a).

4. GAME Implementation

The GAME framework is being developed in Java on the SOLARIS platform. It makes use of a C interface (linked via a system-call class) to obtain the services of the operating system or other utilities already available. A preliminary version of the agent registry, email, and talk agents have been implemented. The meeting agent would be added to this collection after collecting user feedback on GAME operation. A sample interaction between the talk and the email agent is shown in Figure 5.

The email agent uses the service of the MH mail package whenever possible (Peek 1995). Though MH is a powerful email package, individual customization is difficult owing to its cryptic syntax. The email agent currently provides different scan options to view emails, performs message selection based on user criteria, sends emails, and provides suggestions for folder classification and on the emails of a folder. Task transfer with the talk agent allows a user to cross reference a string in an email with the talk transcripts maintained by the talk agent.

The talk agent manages talk connections, saves talk transcripts over talk sessions, and provides cross reference facility (supporting both full and partial matches) with previous talk transcripts. It has an auto response function that sends a fixed message to a user at the
Scenario: User cross references a keyword from an email with some talk transcripts, and then decides to send (using email) one of the talk transcripts. GAME provides coordination by task transfer between the talk and the email agents.

Figure 5: The talk agent and the email agent. The other talk agent (not shown) was intentionally set in the auto response mode while taking this snapshot.

other end. This can be used if a user needs to briefly leave his/her terminal during a talk session. The talk agent also performs task transfer to a user's email agent, if desired. Its context sensitive operation was described in section 3.

The agent registry performs registration and de-registration of agents, performs talk and communication set up, and answers queries about registered agents. Currently, the services of an agent are known to other agents in the domain for simplicity in managing the coordination structures.

GAME uses typed messages and fixed message formats. Messages received that cannot be decoded meaningfully are ignored. When transferring tasks between agents, the task arguments in a task transfer message are encoded using a simplified LISP like language. An agent receiving the task transfer message parses the language to determine its action. Our intention is that this language should facilitate easy upgrading to a more acceptable format such as KQML (Gensereth and Ketchpel 1994; Mayfield et al. 1996). As GAME uses sockets, the connections are reliable reducing communication costs (no acknowledgments need be sent for received messages).

Every object in an agent whose context is needed by some rule in an agent's rule base has its context maintained in a global context object managed by the agent. Contexts are implemented by associating property lists to objects using the property list management methods provided in Java. As the context object maintained by an agent is shared, care is taken when updating this global context.

Action and service components are implemented as Java threads. For example, the communication component of an agent is a separate thread distinct from the (main) thread that handles user interaction. Similarly, task transfers can cause threads to be invoked and run concurrently. This allows an agent to interact with its user without waiting for independent sub-tasks to finish.

Finally, synchronization in GAME is provided using Java mechanisms to lock shared objects whenever they are accessed for an update. From our experience, this form of concurrency control is far from adequate. There were several instances during development when an inadvertent coding error resulted in a deadlock with subsequent time and effort spent on error fixing. For example, let $S_1$ ($S_2$) be the registry service component for agent $A_1$ ($A_2$). When $S_1$ routes a message from agent $A_1$ to $A_2$, it obtains the coordination structure of $A_2$ from the registry. An inadvertent locking of the socket data I/O streams of $A_2$ by $S_1$ for routing can cause a deadlock because they are already locked by $S_2$ servicing $A_2$. Our experience emphasizes the need for

2If the I/O streams are not locked, it can lead to a race condition.
general concurrency control mechanisms to aid agent-based applications. Its analysis is beyond the scope of this paper.

5. Concluding Remarks
Office automation through cooperating agents issues several challenges that require careful consideration and design. We described an agent framework called GAME and its architecture for office automation. A preliminary version of this framework has been implemented and section outlined its current implementation status and our observations. Agent communication observed in GAME is realistic because communication takes place using sockets and is not simulated. Cooperation between agents in GAME occurs through task transfers and prevents work and effort duplication.

GAME allows exploration of different types of agent interactions (by suitably modifying agent response to task transfers). For example, task transfers using agent "bidding", as opposed to being "free", can provide clues on agent utilization for real life transactions.

Finally, GAME is an open-ended network of agents: additional agents can be accommodated provided they obey the protocols while interacting with other agents. Future work involves agent authentication mechanisms, and task sharing in addition to task transfers to further reduce agent workload. In our belief, this research exploration provides not only deeper insights in agent-based problem solving, but can also culminate in end-user products useful to the society at large.

References