The Open Agent Architecture: A Framework for Building Distributed Software Systems

David L. Martin
Adam J. Cheyer
Douglas B. Moran
Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA
+1 650 859 5552
{martin,cheyer,moran}@ai.sri.com

Abbreviated title: *Open Agent Architecture*

Abstract

The Open Agent Architecture (OAA), developed and used for several years at SRI International, makes it possible for software services to be provided through the cooperative efforts of distributed collections of autonomous agents. Communication and cooperation between agents are brokered by one or more facilitators, which are responsible for matching requests, from users and agents, with descriptions of the capabilities of other agents. Thus, it is not generally required that a user or agent know the identities, locations, or number of other agents involved in satisfying a request. OAA is structured so as to minimize the effort involved in creating new agents and "wrapping" legacy applications, written in various languages and operating on various platforms; to encourage the reuse of existing agents; and to allow for dynamism and flexibility in the makeup of agent communities. Distinguishing features of OAA as compared with related work include extreme flexibility in using facilitator-based delegation of complex goals, triggers, and data management requests; agent-based provision of multimodal user interfaces; and built-in support for including the user as a privileged member of the agent community.

This paper explains the structure and elements of agent-based systems constructed using OAA. The characteristics and use of each major component of OAA infrastructure are described, including the agent library, the Interagent Communication Language, capabilities declarations, service requests, facilitation, management of data repositories, and autonomous monitoring using triggers. To provide technical context, we describe the motivations for OAA's design, and situate its features within the realm of alternative software paradigms. A summary is given of OAA-based systems built to date, and brief descriptions are given of several of these.

1 Introduction

The evolution of models for the design and construction of software systems is being driven forward by several closely interrelated trends:

- The adoption of a *networked computing model* is leading to a greatly increased reliance on distributed sites for both data and processing resources. Indeed, with a reported 1800 new computers being added to the Internet every day, a paradigm shift for computing is well under way, one which moves away from requiring all relevant data and programs to reside on the user's desktop machine. The data now routinely accessed from computers spread around the world has become increasingly rich in format, comprising multimedia documents, and audio and video streams; with the popularization of JAVA, it may also include programs that can be downloaded and executed on the local machine. As we become increasingly reliant on networked computing, we need approaches to software design that allow for flexible composition of distributed processing elements in a dynamically changing and relatively unstable environment.

- In an increasing variety of domains, application designers and users are coming to expect the deployment of *smarter, longer-lived, more autonomous, software applications.* Push technology, persistent monitoring of information sources, and the maintenance of user models, allowing for personalized responses and sharing of preferences, are examples of the simplest manifestations of this trend. Commercial enterprises are introducing significantly more advanced approaches, in many cases employing recent research results from artificial intelligence, data mining, machine learning, and other fields.

- More than ever before, the increasing complexity of systems, the development of new technologies, and the availability of multimedia material and environments are creating a demand for *more accessible, more intuitive user interfaces.* Autonomous, distributed, multicomponent systems providing sophisticated services will no longer lend themselves to the familiar "direct manipulation" model of interaction, in which an individual user masters a fixed selection of commands provided by a single application. Ubiquitous computing, in networked environments, has brought about a situation in which the typical user of many software services is likely to be a nonexpert, who may access a given service infrequently or only a few times. Accommodating such usage patterns calls for new approaches. Fortunately, input modalities now becoming widely available, such as speech recognition and pen-based handwriting/gesture recognition, and the ability to manage the presentation of systems' responses by using multiple media provide an opportunity to fashion a style of human-computer interaction that draws much more heavily on our experience with human-human interactions.

The Open Agent Architecture Architecture (OAA), [1] a framework for constructing multiagent systems developed at the Artificial Intelligence Center of SRI International, arose from a

[1] Open Agent Architecture and OAA are trademarks of SRI International. Other brand names and product names herein are trademarks and registered trademarks of their respective holders.

desire to accommodate developments in these three areas in an integrated framework, which is suitable for practical use. In Sections 2 and 3 of this paper, we first review various approaches to distributed computing, and then situate our own approach within the scope of this related work. Following that, we briefly characterize the range of OAA-based systems built to date. Subsequent sections provide detailed descriptions of the inner workings of OAA. Whereas the motivating concepts for an early version of OAA were presented in (Cohen et al., 1994), and certain OAA-based systems have been described in (Cheyer and Julia, 1995; Martin et al., 1996; Martin et al., 1997; Moore et al., 1996; Moran et al., 1997; Moran and Cheyer, 1995), this is the first paper to present a detailed technical explanation of the system-building resources provided by OAA.

2 Technologies for Distributed Computing

We briefly review the overall concepts, advantages, and disadvantages of several relevant approaches to distributed computing, including distributed objects, mobile objects, blackboard-style architectures, and agent-based software engineering.

2.1 The Distributed Object Approach

Object-oriented languages, such as C++ or JAVA, provide significant advances over standard procedural languages with respect to the reusability and modularity of code:

- Encapsulation: encourages the creation of library interfaces that minimize dependencies on underlying algorithms or data structures. Changes to programming internals can be made at a later date with requiring modifications to the code that uses the library.
- Inheritance: permits the extension and modification of a library of routines and data without requiring source code to the original library.
- Polymorphism: allows one body of code to work on an arbitrary number of data types.

Whereas "standard" object-oriented programming (OOP) languages can be used to build monolithic programs out of many object building blocks, distributed object technologies (DOOP) such as OMG's CORBA ((OMG), 1997) or Microsoft's DCOM (Microsoft, 1996) allow the creation of programs whose components may be spread across multiple machines. To implement a client-server relationship between objects, distributed object systems use a registry mechanism (CORBA's registry is called an Object Request Broker, or ORB) to store the interface descriptions of available objects. Through the ORB's services, a client can transparently invoke a method on a remote server object; the ORB is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results. The client does not have to be aware of where the object

is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface.

Although distributed objects offer a powerful paradigm for creating networked applications, certain aspects of the approach are not perfectly tailored to the constantly changing environment of the Internet. A major restriction of the DOOP approach is that the interactions among objects are fixed through explicitly coded instructions by the application developer. This implies that it is very difficult to reuse an object in a new application without bringing along all its inherent dependencies on other objects (embedded interface definitions and explicit method calls). Another restriction of the DOOP approach is the result of its reliance on a remote procedure call (RPC) style of communication. Although easy to debug, this single thread of execution model does not facilitate programming to exploit the potential for parallel computation that one would expect in a distributed environment. In addition, RPC uses a blocking (synchronous) scheme that does not scale well for high-volume transactions.

2.2 Mobile Objects

Mobile objects, sometimes called mobile agents, are bits of code that can move to another execution site (presumably on a different machine) under their own programmatic control, where they can then efficiently interact with the local environment. Commercial instantiations of this technology include Aglets from IBM, Concordia from Mitsubishi, and Voyager from ObjectSpace.

For certain types of problems, the mobile object paradigm offers advantages over more traditional distributed object approaches. These advantages include

- Network bandwidth: for some database queries or electronic commerce applications, it is more efficient to perform tests on data by bringing the tests to the data than by bringing large amounts of data to the testing program.
- Parallelism: mobile agents can be spawned in parallel to accomplish many tasks at once.

Disadvantages (or inconveniences) of the mobile agent approach are that

- In a fashion similar to that of DOOP programming, an agent developer must programmatically specify where to go and how to interact with the target environment.
- There is generally little coordination support to encourage interactions among multiple (mobile) participants.
- Agents must be written in the programming language supported by the execution environment, whereas many other distributed technologies support heterogeneous communities of components, written in diverse programming languages.

2.3 Blackboard Architectures

Blackboard approaches, such as Schwartz's FLiPSiDE (Schwartz, 1995) or Gelernter's LINDA (Gelernter, 1993), allow multiple processes to communicate by reading and writing tuples from a global data store. Each process can watch for items of interest, perform computations based on the state of the blackboard, and then add partial results or queries that other processes can consider.

Blackboard architectures provide a flexible framework for problem solving by a dynamic community of distributed processes. A blackboard approach provides one solution to eliminating the tightly bound interaction links that some of the other distributed technologies require during interprocess communication. This advantage can also be a disadvantage: although a programmer does not need to refer to a specific process during computation, the framework does not provide programmatic control for doing so in cases where this would be practical.

2.4 Agent-based Software Engineering

Several research communities have approached distributed computing by casting it as a problem of modeling communication and cooperation among autonomous entities. Effective communication among independent actors requires four components: (1) a transport mechanism carrying messages in an asynchronous fashion, (2) an interaction protocol defining various types of communication interchange and their social implications (for instance, a response is expected of a question), (3) a content language permitting the expression and interpretation of utterances, and (4) an agreed-upon set of shared vocabulary and meaning for concepts (often called an *ontology*). Such mechanisms permit a much richer style of interaction among participants than can be expressed using a distributed object's RPC model or a blackboard architecture's centralized exchange approach.

Undoubtably, the most widely used foundation technology for agent-based software engineering is the Knowledge Query and Manipulation Language (KQML) (Labrou and Finin, 1997; Finin et al., 1997). KQML, which specifies an interaction protocol, is often used in conjunction with the Knowledge Interchange Format (KIF) (Genesereth and Fikes, 1992) as content language, and either ad hoc or more formalized ontologies. KQML introduced the use of symbolic *performatives* to capture information about the purpose of a communication, and its place within a conversation. Although creating a standardized representation for conversational interactions is one important aspect of multiagent cooperation, KQML is limited by its reliance on a fixed core set of atomic performatives, and the inevitable difficulty in arriving at just the right set capable of expressing every kind interaction and service request.

Another influential approach, which makes stronger assumptions about the knowledge and processing used within individual agents, is based on the structuring of the agents' activities around the concepts of Belief, Desire, and Intention (BDI) (Rao and Georgeff, 1995). While BDI's emphasis on a higher level of abstraction has been extremely important in giving direction to work on agent-based systems, its applicability may be limited by the structural requirements imposed on individual agents, and by difficulties in interoperating with legacy

systems.

3 Philosophy and Goals of OAA

Our approach to distributed computing shares much in common with the paradigms outlined above. As with distributed object frameworks, the primary goal of OAA is to provide a means for integrating heterogeneous applications in a distributed infrastructure. However, we have also sought to incorporate some of the dynamism and extensibility of blackboard approaches, the efficiency associated with mobile objects, and the rich and complex interactions of communicating agents. Here, we spell out in greater detail the goals of OAA, which may be categorized under the general headings of *interoperation and cooperation, user interfaces*, and *software engineering.*

Versatile mechanisms of interoperation and cooperation. *Interoperation* refers to the ability of distributed software components – agents – to communicate meaningfully. While every system-building framework must provide mechanisms of interoperation at some level of granularity, agent-based frameworks face important new challenges in this area. This is true primarily because autonomy, the hallmark of *individual* agents, necessitates greater flexibility in interactions within *communities* of agents. *Coordination* refers to the mechanisms by which a community of agents is able to work together productively on some task. In these areas, the goals for our framework are to

- *Provide flexibility in assembling communities of autonomous service providers* — both at development time and at runtime. Agents that conform to the linguistic and ontological requirements for effective communication should be able to participate in an agent community, in various combinations, with minimal prerequisite knowledge of the characteristics of the other players. Agents with duplicate and overlapping capabilities should be able to coexist within the same community, with the system making the best possible use of the redundancy.

- *Provide flexibility in structuring cooperative interactions* among the members of a community of agents. A framework should provide economical means of setting up a variety of interaction patterns among agents, without requiring an inordinate amount of complexity or infrastructure within the individual agents. The provision of a service should not be dependent upon a particular configuration of agents.

- *Impose the right amount of structure* on individual agents. Different approaches to the construction of multiagent systems impose different requirements on the individual agents. For example, because KQML is neutral as to the content of messages, it imposes minimal structural requirements on individual agents. On the other hand, the BDI paradigm is likely to impose much more demanding requirements, because it makes assumptions about the nature of the programming elements that are meaningful to individual agents. OAA falls somewhere between the two; our goal has been to provide a rich set of interoperation and coordination, but without precluding any of the software engineering goals defined below.

- *Include legacy and "owned-elsewhere" applications.* Whereas *legacy* usually implies reuse of an established system fully controlled by the agent-based system developer, *owned-elsewhere* refers to applications to which the developer has partial access, but no control. Examples of the latter are data sources and services available on the World Wide Web, via simple form-based interfaces, and applications used cooperatively within a virtual enterprise, which remain the property of separate corporate entities. It must be possible for both classes of application to interoperate, more or less as full-fledged members of the agent community, without requiring an overwhelming integration effort.

Human-oriented user interfaces. Systems composed of multiple distributed components, and possibly dynamic configurations of components, require the crafting of intuitive user interfaces to

- Provide *conceptually natural* means of interacting with multiple distributed components. When there are numerous disparate agents, and/or complex tasks implemented by the system, the user should be able to express requests without having detailed knowledge of the individual agents. With speech recognition, handwriting recognition, and natural language technologies becoming more mature, an agent architecture must be prepared for these forms of input to play an increased role in the tasking of agent communities.
- Treat *users as privileged members* of the agent community. By providing an appropriate level of task specification within *software* agents, and reusable means of translating between this level and the level of *human* requests, it should be possible to construct interactions that seamlessly incorporate both types of "agent".
- Support *collaboration* (simultaneous work over shared data and processing resources) between users and agents.

Realistic software engineering requirements. To be successful, a system-building framework must address the practical concerns of real-world applications, as expressed by these goals:

- *Minimize the effort* required to create new agents, and to wrap existing applications.
- *Encourage reuse,* both of domain-independent and domain-specific components. The concept of *agent orientation*, like that of object orientation, provides a natural conceptual framework for reuse, so long as mechanisms for encapsulation and interaction are structured appropriately.
- *Support lightweight, mobile platforms.* Such platforms should be able to serve as hosts for agents, without requiring the installation of a massive environment. It should also be possible to construct individual agents that are relatively small and modest in their processing requirements.

- *Minimize platform and language barriers.* Creation of new agents, as well as wrapping of existing applications, should not require the adoption of a new language or environment.

4 Overview of OAA

In this section, we present an overview of OAA, first describing the basic components and structure of the framework, and then illustrating these concepts with a sample application.

4.1 OAA System Structure

Figure 1 presents the structure typical of a small OAA system, showing a user interface agent and several application agents and meta-agents, organized as a community of peers by their common relationship to a facilitator agent.

The facilitator is a specialized server agent that is responsible for coordinating agent communications and cooperative problem-solving. In many systems, the facilitator is also used to provide a global data store for its client agents, which allows them to adopt a blackboard style of interaction. Note that a system configuration is not limited to a single facilitator. Larger systems can be assembled from multiple facilitator/client groups, each having the sort of structure shown in Figure 1.

The other categories of agents illustrated here – application agents, meta-agents, and user interface agents – are categories recognized by convention only; that is, they are not formally distinguished within the system. Application agents are usually specialists that provide a collection of services of a particular sort. These services could be domain-independent technologies (such as speech recognition, natural language processing, email, and some forms of data retrieval and data mining) or user-specific or domain-specific (such as a travel planning and reservations agent). Application agents may be based on legacy applications or libraries, in which case the agent may be little more than a wrapper that calls a pre-existing API.

Meta-agents are those whose role is to assist the facilitator agent in coordinating the activities of other agents. While the facilitator possesses domain-independent coordination strategies, meta-agents can augment these by using domain- and application-specific knowledge or reasoning (rules, learning algorithms, planning, and so forth).

The user interface agent plays an extremely important and interesting role in many OAA systems. In some systems, this agent is implemented as a collection of "micro-agents", each monitoring a different input modality (point-and-click, handwriting, pen gestures, speech), and collaborating to produce the best interpretation of the current inputs. These micro-agents are shown in Figure 1 as Modality Agents.

All agents that are *not* facilitators are referred to as *client* agents — so called because each acts (in some respects) as a client of some facilitator, which provides communication and other essential services for the client. When invoked, a client agent makes a connection to

a facilitator, which is known as its *parent facilitator*. Upon connection, an agent informs its parent facilitator of the services it can provide. When the agent is needed, the facilitator sends it a request expressed in the Interagent Communication Language (ICL). The agent parses this request, processes it, and returns answers or status reports to the facilitator. In processing a request, the agent can make use of a variety of capabilities provided by OAA. For example, it can use ICL to request services of other agents, set triggers, and read or write shared data on the facilitator (or other client agents that maintain shared data).

The common infrastructure for constructing agents is supplied by an *agent library*, which is available in several different programming languages. The library has been designed to minimize the effort required to construct a new system, and to maximize the ease with which legacy systems can be agentified.

4.2 Sample Interactions

Perhaps the best way to obtain an intuitive sense of how the OAA typically functions is to briefly look at an example of how OAA has been applied to a real application. In the Automated Office system, a mobile executive with a telephone and a laptop computer can access and task commercial applications such as calendars, databases, and email systems running back at the office. As depicted in Figure 2, an application agent provides a wrapper for each program, making its functionality and natural language vocabulary available to the agent community through registration with a facilitator.

A user interface (UI) agent, shown in Figure 3, runs on the user's local laptop and is responsible for accepting user input, sending requests to the facilitator for delegation to appropriate agents, and displaying the results of the distributed computation. The user may interact directly with a specific remote application by clicking on active areas in the interface, calling up a form or window for that application, and making queries with standard interface dialog mechanisms. Conversely, a user may express a task to be executed by using typed, handwritten, or spoken (over the telephone) English sentences, without explicitly specifying which agent or agents should perform the task. For instance, if the question "What is my schedule?" is written in the user interface, this request will be sent by the UI to the facilitator, which in turn will ask a natural language (NL) agent to translate the query into ICL. To accomplish this task, the NL agent may itself need to make requests of the agent community to resolve unknown words such as "me" (the UI agent can respond with the name of the current user) or "schedule" (the calendar agent defines this word). The resulting ICL expression is then routed by the facilitator to appropriate agents (in this case, the calendar agent) to execute the request. Results are sent back to the UI agent for display.

The spoken request "When mail arrives for me about security, notify me immediately." produces a slightly more complex example involving communication among all agents in the system. After translation into ICL as described above, the facilitator installs a trigger on the mail agent to look for new messages about security. When one such message does arrive in its mail spool, the trigger fires, and the facilitator matches the action part of the trigger to capabilities published by the notification agent. The notification agent is an

example of a meta-agent, as it makes use of rules concerning the optimal use of different output modalities (email, fax, speech generation over the telephone) plus information about an individual user's preferences to determine the best way of relaying a message through available media transfer application agents. After some competitive parallelism to locate the user (the calendar and database agents may have different guesses as to where to find the user) and some cooperative parallelism to produce required information (telephone number of location, user password, and an audio file containing a text-to-speech representation of the email message), a telephone agent can call the user, verify identity through touchtones, and then play the message.

Some key ideas illustrated by the above examples are the following:

1. As new agents connect to the facilitator, registering capability specifications and natural language vocabulary, what the user can say and do dynamically changes.
2. The interpretation and execution of a task is a distributed process, with no one agent defining the set of possible inputs to the system.
3. A single request can produce cooperation and flexible communication among many agents, written in different programming languages and spread across multiple machines.

In our following detailed view of the Open Agent Architecture, we order the presentation top-down, beginning with the means by which a group of agents works together, then considering the mechanisms that support the use of shared data repositories and triggers, and finally describing some of the basic infrastructure underlying the construction of individual agents. To illustrate the technical aspects of the approach, we describe several applications implemented within the OAA.

5 Mechanisms of Cooperation

Cooperation among the agents of an OAA system is achieved via messages expressed in a common language, ICL, and is normally structured around a three-part approach: providers of services register capabilities specifications with a facilitator, requesters of services construct goals and relay them to a facilitator, and facilitators coordinate the efforts of the appropriate service providers in satisfying these goals.

5.1 The Interagent Communication Language

OAA's Interagent Communication Language (ICL) is the interface, communication, and task coordination language shared by all agents, regardless of what platform they run on or what computer language they are programmed in. ICL is used by an agent to task itself or some

subset of the agent community, either using explicit control or, more frequently, in an underspecified, loosely constrained manner. OAA agents employ ICL to perform queries, execute actions, exchange information, set triggers, and manipulate data in the agent community.

One of the fundamental program elements expressed in ICL is the *event*. The activities of every agent, as well as communications between agents, are structured around the transmission and handling of events. In communications, events serve as messages between agents; in regulating the activities of individual agents, they may be thought of as goals to be satisfied.

Each event has a type, a set of parameters, and content. For example, the agent library procedure *oaa_Solve* can be used by an agent to request services of other agents. A call to *oaa_Solve*, within the code of agent *A*, results in an event having the form

ev_post_solve(Goal, Params)

going from *A* to the facilitator, where *ev_post_solve* is the type, *Goal* is the content, and *Params* is a list of parameters. The allowable content and parameters vary according to the type of the event.

The ICL includes a layer of conversational protocol, similar in spirit to that provided by KQML, and a content layer, analogous to that provided by KIF. The conversational layer of ICL is defined by the event types, together with the parameter lists associated with certain of these event types. The content layer consists of the specific goals, triggers, and data elements that may be embedded within various events.

The conversational protocol is specified using an orthogonal, parameterized approach. That is, the conversational aspects of each element of an interagent conversation are represented by a selection of an event type, in combination with a selection of values for an orthogonal set of parameters. This approach offers greater expressiveness than an approach based solely on a fixed selection of *speech acts*, such as embodied in KQML. For example, in KQML, a request to satisfy a query can employ either of the performatives *ask_all* or *ask_one*. In ICL, on the other hand, this type of request is expressed by the event type *ev_post_solve*, together with the *solution_limit(N)* parameter – where *N* can be any positive integer. (A request for all solutions is indicated by the omission of the *solution_limit* parameter.) The request can also be accompanied by other parameters, which combine to further refine its semantics.

In KQML, then, this example forces one to choose between two possible conversational options, neither of which may be precisely what is desired. In either case, the performative chosen is a single value that must capture the entire conversational characterization of the communication. This requirement raises a difficult challenge for the language designer, to select a set of performatives that provides the desired functionality without becoming unmanageably large. Consequently, the debate over the right set of performatives has consumed much discussion within the KQML community.

The content layer of the ICL has been designed as an extension of the PROLOG programming language, to take advantage of unification and other features of PROLOG. OAA's agent libraries (especially the non-PROLOG versions) provide support for constructing, parsing, and manipulating ICL expressions.

While it is possible to embed content expressed in other languages within an ICL event, it is advantageous to express content in ICL wherever possible. The primary reason for this is to allow the facilitator access to the content, as well as the conversational layer, in delegating requests. Not only does this give the facilitator more information about the nature of a request, but it also makes it possible for the facilitator to decompose compound requests, and individually delegate the subrequests.

Important declarations and other program elements represented using ICL expressions include, in addition to events, capabilities declarations, requests for services, responses to requests, trigger specifications, and shared data elements.

5.2 Providing Services

Every agent participating in an OAA-based system defines and publishes a set of capabilities declarations, expressed in ICL, describing the services that it provides. These declarations establish a high-level interface to the agent. This interface is used by a facilitator in communicating with the agent, and, most important, in delegating service requests (or parts of requests) to the agent. Partly due to the use of PROLOG as the basis of ICL, we refer to these capabilities declarations as *solvables*.

Two major types of solvables are distinguished: *procedure* solvables and *data* solvables. Intuitively, a procedure solvable performs a test or action, whereas a data solvable provides access to a collection of data. For example, in creating an agent for a mail system, procedure solvables might be defined for sending a message to a person, testing whether a message about a particular subject has arrived in the mail queue, or displaying a particular message onscreen. For a database wrapper agent, one might define a distinct data solvable corresponding to each of the relations present in the database. Often, a data solvable is used to provide a *shared* data store, which may be not only queried, but also updated, by various agents having the required permissions.

Technically, the primary differences between the two types of solvables are these: First, each procedure solvable must have a handler declared and defined for it, whereas this is not necessary for a data solvable. (The handling of requests for a data solvable is provided transparently by the agent library.) Second, data solvables are associated with a dynamic collection of facts (or clauses), which may be modified at runtime, both by the agent providing the solvable, and by other agents (provided they have the required permissions). Third, special features, available for use with data solvables, facilitate maintaining the associated facts. Some of these features are mentioned in Section 6.

In spite of these differences, it should be noted that the means of *use* (that is, the means by which an agent requests a service) is the same for the two types of solvables. Requesting of services is described in Section 5.3.

A request for one of an agent's services normally arrives in the form of an event from the agent's facilitator. The appropriate handler then deals with this event. The handler may be coded in whatever fashion is most appropriate, depending on the nature of the task, and

the availability of task-specific libraries or legacy code, if any. The only hard requirement is that the handler return an appropriate response to the request, expressed in ICL. Depending on the nature of the request, this response could be an indication of success or failure, or a list of solutions (when the request is a data query).

The agent library provides a set of procedures allowing an agent to add, remove, and modify its solvables, which it may do at any time after connecting to its facilitator.

5.2.1 Specification of Solvables

A solvable has three parts: a *goal*, a list of *permissions*, and a list of *parameters*, which are declared using the format

```
solvable(Goal, Parameters, Permissions)
```

The goal of a solvable, which syntactically takes the form of an ICL structure, is a logical representation of the service provided by the solvable. (An ICL structure consists of a *functor* with 0 or more arguments. For example, in the structure a(b,c), 'a' is the functor, and 'b' and 'c' the arguments.) As with a PROLOG structure, the goal's arguments may themselves be structures.

Various options can be included in the parameters list, to refine the semantics associated with the solvable. First and foremost, the *type* parameter is used to say whether the solvable is *data* or *procedure*. When the type is *procedure*, another parameter may be used to indicate the handler to be associated with the solvable. Some of the parameters appropriate for a *data* solvable are mentioned in Section 6.

In either case (procedure or data solvable), the *private* parameter may be used to restrict the use of a solvable to the declaring agent. This parameter is valuable when the agent intends the solvable to be solely for its internal use and wants to take advantage of OAA mechanisms in accessing it, or when the agent wants the solvable to be available to outside agents only at selected times. In support of the latter case, it is possible for the agent to change the status of a solvable from private to nonprivate at any time.

The permissions of a solvable provide the means by which an agent may control access to its services. They allow the agent to restrict calling and writing of a solvable to itself and/or other selected agents. (*Calling* means requesting the service encapsulated by a solvable, whereas *writing* means modifying the collection of facts associated with a data solvable.) The default is for every solvable to be callable by anyone, and for data solvables to be writable by anyone. A solvable's permissions can be changed at any time, by the agent providing the solvable.

For example, the solvables of a simple email agent might include

```
solvable(send_message(email, +ToPerson, +Params),
         [type(procedure), callback(send_mail)],
         [])
```

```
solvable(last_message(email, -MessageId),
         [type(data), single_value(true)],
         [write(true)]),
solvable(get_message(email, +MessageId, -Msg),
         [type(procedure), callback(get_mail)],
         [])
```

The symbols '+' and '-', indicating input and output arguments, are at present used only for purposes of documentation. Most parameters and permissions have default values, and specifications of default values may be omitted from the parameters and permissions lists.

A programmer who defines an agent's capabilities in terms of solvable declarations is, in a sense, creating the vocabulary with which other agents will communicate with the new agent. The problem of ensuring that agents will speak the same language and share a common, unambiguous semantics of the vocabulary, is called the *ontology problem*. The OAA provides a few tools (see more about agent development tools in (Martin et al., 1996)) and services (automatic translations of solvables by the facilitator) to help minimize this issue; however, the OAA still must rely on vocabulary from either formally engineered ontologies for specific domains (for instance, see http://www-ksl.stanford.edu/knowledge-sharing/ontologies/html/) or on ontologies constructed during the incremental development of a body of agents for several applications.

Although OAA imposes no hard restrictions (other than the basic syntax) on the form of solvable declarations, two common usage conventions illustrate some of the utility associated with solvables.

- Classes of services are often tagged by a particular type. For instance, in the example above, the "last_message" and "get_message" solvables are specialized for email, not by modifying the *names* of the services, but rather by the use of the 'email' parameter, which serves during the execution of an ICLrequest to select (or not) a specific type of message.
- Actions are generally written using an imperative verb as the functor of the solvable, the direct object (or item class) as the first argument of the predicate, required arguments following, and then an extensible parameter list as the last argument. The parameter list can hold optional information usable by the function. The ICLexpression generated by a natural language parser often makes use of this parameter list to store prepositional phrases and adjectives.

As an illustration of the above two points, "Send mail to Bob about lunch" will be translated into an ICLrequest send_message(email, 'Bob Jones', [subject(lunch)]), whereas "Remind Bob about lunch" would leave the transport unspecified (send_message(KIND, 'Bob Jones', [subject(lunch)])), enabling all available message transfer agents (e.g., fax, phone, mail, pager) to compete for the opportunity to carry out the request.

5.3 Requesting Services

An agent requests services of the community by delegating tasks or goals to its facilitator. Each request contains calls to one or more agent solvables, and optionally specifies parameters containing advice to help the facilitator determine how to execute the task. It is important to note that calling a solvable does *not* require that the agent specify (or even know of) a particular agent or agents to handle the call. While it is possible to specify one or more agents using an address parameter (and there are situations in which this is desirable), in general it is advantageous to leave this delegation to the facilitator. Programming in this style greatly reduces the hard-coded dependencies among components that one often finds in other distributed frameworks.

The OAA libraries provide an agent with a single, unified point of entry for requesting services of other agents: the library procedure *oaa_Solve*. In the style of logic programming, *oaa_Solve* may be used both to retrieve data and to initiate actions. To put this another way, calling a *data* solvable looks the same as calling a *procedure* solvable.

5.3.1 Compound Goals

One of the most powerful features of OAA is the ability of a client agent (or a user) to submit compound goals to a facilitator. A compound goal is composed using operators similar to those employed by PROLOG, that is, the comma for conjunction, the semicolon for disjunction, and the arrow for conditional execution. Three of the several significant extensions to PROLOG syntax and semantics are of particular interest here. First, a "parallel disjunction" operator indicates that the disjuncts are to be executed (by different agents) simultaneously. Second, it is possible to specify whether a given subgoal is to be executed breadth-first or depth-first. [2] Third, each subgoal of a compound goal can have an address and/or a set of parameters attached to it. Thus, each subgoal takes the form

Address:Goal::Parameters

where both *Address* and *Parameters* are optional.

An address, if present, specifies one or more agents to handle the given goal, and may employ several different types of referring expression: unique names, symbolic names, and shorthand names. Every agent has a unique name, assigned by its facilitator, which relies upon network addressing schemes to ensure its global uniqueness. Agents also have self-selected symbolic names (for example, "mail"), which are not guaranteed to be unique. When an address includes a symbolic name, the facilitator takes this to mean that all agents having that name should be called upon. Shorthand names include 'self' and 'parent' (which refers to the agent's facilitator). We emphasize that the address associated with a goal or subgoal is always optional. When an address is not present, it is the facilitator's job to supply an appropriate address, as explained in Section 5.5.

[2] This capability is under development.

The distributed execution of compound goals becomes particularly powerful when used in conjunction with natural language or speech-enabled interfaces, as the query itself may specify how functionality from distinct agents will be combined. As a simple example, the spoken utterance "Fax it to Bill Smith's manager." can be translated into the following compound ICL request:

```
oaa_Solve((manager('Bill Smith', M), fax(it,M,[])), [strategy(action)])
```

5.4 Refining Service Requests

The parameters associated with a goal (or subgoal) can draw on useful features to refine the request's meaning. For example, it is frequently important to be able to specify whether or not solutions are to be returned synchronously; this is done using the *reply* parameter, which can take any of the values *synchronous, asynchronous*, or *none*. As another example, when the goal is a noncompound query of a data solvable, the *cache* parameter may be used to request local caching of the facts associated with that solvable. Many of the remaining parameters fall into two categories: advice and feedback.

Feedback parameters allow a service requester to receive information from the facilitator about how a goal was handled. This feedback can include such things as the identities of the agents involved in satisfying the goal, and the amount of time expended in the satisfaction of the goal.

Advice parameters give constraints or guidance for the facilitator to use in completing and interpreting the goal. For example, the *solution_limit* parameter allows the requester to say how many solutions it is interested in; the facilitator and/or service providers are free to use this information in optimizing their efforts. Similarly, *time_limit* is used to say how long the requester is willing to wait for solutions to its request, and, in a multifacilitator system, *level_limit* may be used to say how remote the facilitators may be that are consulted in the search for solutions. The *priority* parameter is used to indicate that a request is more urgent than previous requests that have not yet been satisfied. Other advice parameters are used to tell the facilitator whether parallel satisfaction of the parts of a goal is appropriate, how to combine and filter results arriving from multiple solver agents, and whether the requester itself may be considered a candidate solver of the subgoals of a request.

As mentioned in section 5.1, advice parameters are intended to provide an extensible set of low-level, orthogonal parameters capable of combining with the ICL goal language to fully express how information should flow among participants. Multiple parameters can be grouped together and given a group name; the resulting *high-level advice parameters* can be used to express concepts analogous to KQML's performatives, but also to define classifications of problem types. For instance, KQML's "ask_all" and "ask_one" performatives would be represented as combinations of values given to the parameters *reply*, *parallel_ok*, and *solution_limit*. As an example of a higher-level problem type, the strategy "math_problem" might send the query to all appropriate math solvers in parallel, collect their responses, and signal a conflict if different answers are returned. The strategy "essay_question" would send

the request to all appropriate participants, and signal a problem (i.e., cheating) if any of the returned answers are identical.

When a facilitator receives a compound goal, its job is to construct a goal satisfaction plan and oversee its satisfaction in the most appropriate, efficient manner that is consistent with the specified advice.

5.5 Facilitation

Facilitation plays a central role in OAA. At its core, our notion of facilitation is similar to that proposed by Genesereth (Genesereth and Singh, 1993) and others. In short, a facilitator maintains a knowledge base that records the capabilities of a collection of agents, and uses that knowledge to assist requesters and providers of services in making contact. But our notion of facilitation is also considerably stronger in four respects.

First, it encompasses a very general notion of *transparent delegation*, which means that a requesting agent can generate a request, and a facilitator can manage the satisfaction of that request, without the requester needing to have any knowledge of the identities or locations of the satisfying agents. In some cases, such as when the request is a data query, the requesting agent may also be oblivious to the *number* of agents involved in satisfying a request. Transparent delegation is possible because agents' capabilities (solvables) are treated as an abstract description of a service, rather than as an entry point into a library or body of code.

Second, an OAA facilitator is distinguished by its handling of compound goals (introduced in Section 5.3.1). This involves three types of processing: *delegation*, that is, determination of who (which specific agents) will execute a compound goal and how (combination and routing of results from subgoals); *optimization* of the completed goal, including parallelization where appropriate; and *interpretation* of the optimized goal. The *delegation* step results in a goal that is unambiguous as to its meaning and as to the agents that will participate in satisfying it. Completing the addressing of a goal involves the selection of one or more agents to handle each of its subgoals (that is, each subgoal for which this selection has not been specified by the requester). In doing this, the facilitator uses its knowledge of the capabilities of its client agents (and possibly of other facilitators, in a multifacilitator system). It may also use strategies or advice specified by the requester, as explained below. The *optimization* step results in a goal whose interpretation will require as few exchanges as possible, between the facilitator and the satisfying agents, and can exploit parallel efforts of the satisfying agents, wherever this does not affect the goal's meaning. The *interpretation* of a goal involves the coordination of requests to the satisfying agents, and assembling their responses into a coherent whole, for return to the requester.

The third respect in which OAA facilitation extends the basic concept of facilitation is that the facilitator can employ strategies and advice given by the requesting agent, thus resulting in a variety of interaction patterns that may be instantiated in the satisfaction of a request. Some of these strategies are mentioned in Section 5.4, and additional possibilities under consideration are mentioned in Section 11.

Finally, the OAA concept of facilitation has been generalized so as to handle the distribution of both data update requests and requests for installation of triggers, using some of the same strategies that are employed in the delegation of service requests. (Triggers and data maintenance mechanisms are discussed in sections 7 and 6 respectively.)

It should be noted that the reliance on facilitation is not absolute; that is, there is no hard requirement that requests and services be matched up by the facilitator, or that interagent communications go through the facilitator. (Indeed, as mentioned elsewhere, there is support in the agent library for explicit addressing of requests, and planned support for peer-to-peer communications.) However, OAA has been designed so as to encourage developers to employ the paradigm of community, and to minimize their development effort in doing so, by taking advantage of the facilitator's provision of transparent delegation and handling of compound goals.

In summary, we stress that a facilitator is always viewed as a *coordinator*, not a controller, of cooperative task completion. The facilitator never initiates an activity, but rather responds to requests to manage the satisfaction of some goal, the update of some data repository, or the installation of a trigger by the appropriate agent or agents. This approach makes it possible for all agents to take advantage of the facilitator's expertise in delegation, and its up-to-date knowledge about the current membership of a dynamic community. In addition, in many situations, the facilitator's coordination services allows the developer to lessen the complexity of individual agents, resulting in a more manageable software development process, and enabling the creation of lightweight agents.

6 Maintaining Data Repositories

The agent library supports the creation, maintenance, and use of databases, in the form of data solvables. Creation of a data solvable requires only that it be declared, as explained in Section 5.2.1. Querying a data solvable, as with access to any solvable, is done using *oaa_Solve*. Here, we clarify the ways in which these solvables are maintained and used, and mention some of the features associated with them.

A data solvable is conceptually the same as a relation in a relational database. The facts associated with each solvable are maintained by the agent library, which also handles incoming messages containing queries of data solvables. It is possible to refine the default behavior of the library in managing these facts, using parameters specified with the solvable's declaration. For example, the parameter *single_value* is used to indicate that the solvable should only contain a single fact at any given point in time. The parameter *unique_values* indicates that no duplicate values should be stored.

Other parameters can allow data solvables to make use of the concepts of ownership and persistence. Because data solvables are often used to implement shared repositories, it can be useful to maintain a record of which agent created each fact of a solvable; this agent is considered to be the fact's owner. In many applications, it is useful to have an agent's facts removed when that agent goes offline (that is, the agent is no longer participating in

the agent community, whether by deliberate termination or by malfunction). When a data solvable is declared to be nonpersistent, its facts are automatically maintained in this way, whereas a persistent data solvable retains its facts until they are explicitly removed.

The agent library provides procedures by which agents can update (add, remove, and replace) facts belonging to data solvables, either locally or on other agents, given that they have the required permissions. These procedures may be refined using many of the same parameters that apply to service requests. For example, the *address* parameter is used to specify one or more particular agents to which the update request applies. In its absence, just as with service requests, the update request goes to *all* agents providing the relevant data solvable. This default behavior can be used to maintain coordinated "mirror" copies of a data set within multiple agents, and can be useful in support of distributed, collaborative activities.

Similarly, the *feedback* parameters, described in connection with *oaa_Solve*, are also available for use with data maintenance requests.

The ability to provide data solvables is not limited to client agents; data solvables can also be maintained by a facilitator, at the request of a client of the facilitator, and their maintenance and use shared by all the facilitator's clients. This can be a useful strategy with a relatively stable collection of agents, where the facilitator's workload is predictable.

6.1 Using a Blackboard Style of Communication

When a data solvable is publicly readable and writable, it may be thought of as a global data repository, which can be used cooperatively by a group of agents. In combination with the use of triggers, this allows the agents to organize their efforts around a "blackboard" style of communication.

As an example, the "DCG-NL" agent (one of several existing natural language processing agents), which provides natural language processing services for a variety of its peer agents, expects those other agents to record, on the facilitator, the vocabulary to which they are prepared to respond, with an indication of each word's part of speech, and of the logical form (ICL subgoal) that should result from the use of that word. To make this possible, when it comes online, the NL agent installs a data solvable for each basic part of speech on its facilitator. For instance, one such solvable would be

```
solvable(noun(Meaning, Syntax), [], [])
```

(Note that the empty lists for the solvable's permissions and parameters are acceptable here, since the default permissions and parameters provide appropriate functionality.)

In the Office Assistant system, several agents make use of these services. For instance, the database agent uses the following call, to library procedure *oaa_AddData*, to post the noun 'boss', and to indicate that the "meaning" of boss is the concept 'manager':

```
oaa_AddData(noun(manager, atom(boss)), [address(parent)])
```

7 Autonomous Monitoring with Triggers

OAA triggers provide a general mechanism for requesting that some action be taken when some set of conditions is met. Each agent can install triggers either locally, for itself, or remotely, on its facilitator or peer agents. There are four types of triggers: communication, data, task, and time. In addition to a type, each trigger specifies a condition and an action, both expressed in ICL. The condition indicates under what circumstances the trigger should fire, and the action indicates what should happen when it fires. In addition, each trigger can be set to fire either an unlimited number of times, or a specified number of times, which can be any positive integer.

Triggers are used in a wide variety of ways within OAA systems, for example, for monitoring external sensors in the execution environment, tracking the progress of complex tasks, or coordinating communications between agents that are essential for the synchronization of related tasks. The installation of a trigger within an agent can be thought of as a representation of that agent's *commitment* to carry out the specified action, whenever the specified condition holds true.

The four types of triggers can be characterized informally as follows:

- *Communication triggers* allow any incoming or outgoing event (message) to be monitored. For instance, a simple communication trigger may say something like

  "Whenever a solution to a goal is returned from the facilitator, send the result to the presentation manager to be displayed to the user."

- *Data triggers* monitor the state of a data repository (which can be maintained on a facilitator or a client agent). Data triggers' conditions may be tested upon the addition, removal, or replacement of a fact belonging to a data solvable. An example data trigger is

  "When 15 users are simultaneously logged on to a machine, send an alert message to the system administrator."

- *Task triggers* contain arbitrary conditions that are tested after the processing of each incoming event and whenever a timeout occurs in the event polling. These conditions may specify any goal executable by the local ICL interpreter, and most often are used to test when some solvable becomes satisfiable.

  Task triggers are useful in checking for task-specific internal conditions. Although in many cases such conditions are captured by solvables, in other cases they may not be. For example, a mail agent might watch for new incoming mail, or an airline database agent may monitor which flights will arrive later than scheduled. An example task trigger is

  "When mail arrives for me about security, notify me immediately."

- *Time triggers* monitor time conditions. For instance, an alarm trigger can be set to fire at a single fixed point in time (e.g., "On December 23rd at 3pm"), or on a recurring basis (e.g., "Every three minutes from now until noon").

Triggers are implemented as data solvables, declared implicitly for every agent. When requesting that a trigger be installed, an agent may use many of the same parameters that apply to service and data maintenance requests.

One important feature of OAA triggers is that, in contrast with most programming methodologies, the agent on which the trigger is installed only has to know how to evaluate the conditional part of the trigger, not the consequence – when the trigger fires, the action is delegated to the facilitator for execution. Whereas many commercial mail programs allow rules of the form "When mail arrives about XXX, [forward it, delete it, archive it]", the possible actions are hard-coded and the user must select from a fixed set. In OAA, the consequence may be any compound goal executable by the dynamic community of agents. Since new agents define both functionality and vocabulary, when an unanticipated agent (for example, a fax agent) joins the community, no modifications to existing code is required for a user to make use of it – "When mail arrives, fax it to Bill Smith."

8 The Agent Library

OAA's agent library, which provides the necessary infrastructure for constructing an agent-based system, is available in several programming languages, including PROLOG, C, C++, JAVA, LISP, VISUAL BASIC, and DELPHI. As mentioned earlier, two goals of the library's design have been to minimize the effort required to construct a new system, and to maximize the ease with which legacy systems can be agentified.

The library's several families of procedures, provide all the functionalities mentioned in this paper, as well as many that are omitted, for lack of space. For example, declarations of an agent's solvables, and their registration with a facilitator, are managed using procedures such as *oaa_Declare*, *oaa_Undeclare*, and *oaa_Redeclare*. Updates to data solvables can be accomplished with a family of procedures including *oaa_AddData*, *oaa_RemoveData*, and *oaa_ReplaceData*. Similarly, triggers are maintained using procedures such as *oaa_AddTrigger*, *oaa_RemoveTrigger*, and *oaa_ReplaceTrigger*.

The essential elements of protocol (that is, the details of the messages that encapsulate a service request and its response) are provided by the library, and made transparent in so far as possible, so that application code can be simpler. This enables the developer to focus on the desired functionality, rather than on the details of message construction and communication. For example, to request a service of another agent, an agent calls the library procedure *oaa_Solve*. This call results in a message to a facilitator, which will exchange messages with one or more service providers, and then send a message containing the desired results to the requesting agent. These results are returned via one of the arguments of *oaa_Solve*. None of the messages involved in this scenario is explicitly constructed by the agent developer. (Note that this is a description of the *synchronous* use of *oaa_Solve*.)

The agent library provides both *intra*agent and *inter*agent infrastructure; that is, mechanisms supporting the internal structure of individual agents, on the one hand, and mechanisms of cooperative interoperation between agents, on the other. It is worth noting that most of

the infrastructure cuts across this boundary; that is, many of the same mechanisms support both agent internals and agent interactions in an integrated fashion. For example, services provided by an agent can be accessed by that agent through the same procedure (*oaa_Solve*) that it would employ to request a service of another agent (the only difference being in the *address* parameter accompanying the request). This, in turn, helps the developer to reuse code and avoid redundant entry points into the same functionality.

Both of the characteristics described above (transparent construction of messages and integration of *intra*agent with *inter*agent mechanisms) apply to most other library functionality as well, including data management and temporal control mechanisms.

9 OAA Applications

The OAA has been used to implement more than fifteen applications integrating such diverse technologies as image processing, speech recognition, multiuser collaboration, text extraction, planning, and virtual reality. Table 1 is a partial list of OAA-based applications from which we will take a few examples to illustrate qualities important to the framework.

9.1 Unified Messaging

The Unified Messaging application extends the Automated Office presented in section 4 with an emphasis on ubiquitous access and dynamic presentation of the information and services supported by the agent community. The agents used in this application are depicted in Figure 4.

A real dialog taken from the current system can provide insight into how systems are built using OAA. In this scenario, the user, with only a telephone as an interface, is planning a trip to Boston where he will soon give a presentation. Capitalized sentences are phrases spoken by the user into the telephone and processed by a phone agent. Responses, unless otherwise indicated, are spoken by the system using text-to-speech generation.

```
1.1  Welcome to SRI International.  Please enter your user ID and password.
     <Enters touchtone ID and password>
     Good to see you again Adam Cheyer.  I am listening to you.
```

Every user interface agent, including the telephone agent, should know the identify of its user. This information is used in resolving anaphoric references such as "me" and "I", and allows multiple user interfaces operated by the same user to work together.

```
1.2  WHAT IS TODAY'S SCHEDULE?
     Here is today's schedule for Adam Cheyer:
       At 10am for 1 hour, meeting with Dave.
```

```
      At 3pm for 1 hour, presentation about software agents.
    End of schedule.
```

If the user is operating both a graphical user interface and a telephone, as described in conjunction with the Automated Office application, the result of this spoken request is to display a calendar window containing the current schedule. In this case, with no graphical display available, the GEN_NL agent is tasked to produce a spoken response that can be played over the phone. GEN_NL shares the same dynamic vocabulary and phrasal rules as the natural language parser DCG_NL, and contains strategies for producing responses to queries using either simple or list-based multimedia utterances.

```
1.3  FIND FRIDAY'S WEATHER IN BOSTON.
     The weather in Boston for Friday is as follows:
       Sunny in the morning. Partly cloudy in the afternoon with a 20
       percent chance of thunderstorms late. Highs in the mid 70s.
```

In addition to data accessible from legacy applications, content may be retrieved by web-reading agents which provide OAA wrappers around useful websites.

```
1.4  FIND ALL NEW MAIL MESSAGES.
     There are 2 messages available.
     Message 1, from Mark Tierny, entitled ``OAA meeting.''

1.5  NEXT MESSAGE
     Message 2, from Jennifer Schwefler, entitled ``Presentation Summary.''

1.6  PLAY IT.
     This message is a multipart MIME-encoded message.  There are two parts.
     Part 1.  (Voicemail message, not text-to speech):
     Thanks for taking part as a speaker in our conference.
     The schedule will be posted soon on our homepage.

1.7  NEXT PART
     Part 2. (read using text-to-speech):
     The presentation home page is http://www....

1.8  PRINT MESSAGE
     Command executed.
```

Mail messages are no longer just simple text documents, but often consist of multiple subparts containing audio files, pictures, webpages, attachments and so forth. When a user asks to play a complex email message over the telephone, many different agents may be implicated in the translation process, which would be quite different given the request "print it." The challenge is to develop a system which will enable agents to cooperate in an extensible, flexible manner that alleviates explicit coding of agent interactions for every possible input/output combination.

In an OAA implementation, each agent concentrates only on what it can do and on what it knows, and leaves other work to be delegated to the agent community. For instance, a printer agent, defining the solvable print(Object,Parameters), can be defined by the following pseudocode, which basically says, "If someone can get me a document, in either POSTSCRIPT or text form, I can print it.".

```
print(Object, Parameters) {

   ' If Object is reference to ``it'', find an appropriate document
   if (Object = "ref(it)")
      oaa_Solve(resolve_reference(the, document, Params, Object),[]);

   ' Given a reference to some document, ask for the document in POSTSCRIPT
   if (Object = "id(Pointer)")
      oaa_Solve(resolve_id_as(id(Pointer), postscript, [], Object),[]);

   ' If Object is of type text or POSTSCRIPT, we can print it.
   if ((Object is of type Text) or (Object is of type Postscript))
      do_print(Object);
}
```

In our example, since an email message is the salient document, the mail agent will receive a request to produce the message as POSTSCRIPT. Whereas the mail agent may know how to save a text message as POSTSCRIPT, it will not know what to do with a webpage or voicemail message. For these parts of the message, it will simply send oaa_Solve requests to see if another agent knows how to accomplish the task.

Until now, the user has been using only a telephone as user interface. Now, he moves to his desktop, starts a web browser, and accesses the URL referenced by the mail message.

```
1.9  RECORD MESSAGE
     Recording voice message.  Start speaking now.

1.10 THIS IS THE UPDATED WEB PAGE CONTAINING THE PRESENTATION SCHEDULE.
     Message one recorded.

1.11 IF THIS WEB PAGE CHANGES, GET IT TO ME WITH NOTE ONE.
     Trigger added as requested.
```

In this example, a local agent which interfaces with the web browser can return the current page as a solution to the request "oaa_Solve(resolve_reference(this, web_page, [], Ref),[])", sent by the NL agent. A trigger is installed on a web agent to monitor changes to the page, and when the page is updated, the notify agent can find the user and transmit the webpage and voicemail message using the most appropriate media transfer mechanism.

This example based on the Unified Messaging application is intended to show how OAA concepts can be used to produce a simple yet extensible solution to a multiagent problem

that would be difficult to implement using a more rigid framework. The application supports adaptable presentation for queries across dynamically changing, complex information; shared context and reference resolution among applications; and flexible translation of multimedia data. In the next section, we will present an application which highlights the use of parallel competition and cooperation among agents during multimodal fusion.

9.2 Multimodal Map

The goal of the Multimodal Map application is to explore natural ways of communicating with a community of agents. Inspired by the way a professor would instruct his students at a blackboard, through combinations of drawing, writing, speaking, gesturing, circling, underlining and so forth, the Multimodal Map provides an interactive interface on which the user may draw, write or speak. In a travel planning domain (Figure 5), available information includes hotel, restaurant, and tourist-site data retrieved by distributed software agents from commercial Internet sites. The types of user interactions and multimodal issues handled by the application can be illustrated by a brief scenario from (Cheyer et al., 1998) featuring working examples taken from the current system.

```
Sara is planning a business trip to San Francisco, but would like to
schedule some activities for the weekend while she is there.  She turns
on her laptop PC, executes a map application, and selects San Francisco.

2.1   [Speaking] Where is downtown?
      Map scrolls to appropriate area.
2.2   [Speaking and drawing region] Show me all hotels near here.
      Icons representing hotels appear.
2.3   [Writes on a hotel] Info?
      A textual description (price, attributes, etc.) appears.
2.4   [Speaking] I only want hotels with a pool.
      Some hotels disappear.
2.5   [Draws a crossout on a hotel that is too close to a highway]
      Hotel disappears
2.6   [Speaking and circling] Show me a photo of this hotel.
      Photo appears.
2.7   [Points to another hotel]
      Photo appears.
2.8   [Speaking] Price of the other hotel?
      Price appears for previous hotel.
2.9   [Speaking and drawing an arrow] Scroll down.
      Display adjusted.
2.10  [Speaking and drawing an arrow toward a hotel]
      What is the distance from this hotel to Fisherman's Wharf?
      Distance displayed.
2.11  [Pointing to another place and speaking] And the distance to here?
```

```
      Distance displayed.

Sara decides she could use some human advice.  She picks up the phone,
calls Bob, her travel agent, and writes Start collaboration to synchronize
his display with hers. At this point, both are presented with identical
maps, and the input and actions of one will be remotely seen by the other.

3.1   [Sara speaks and circles two hotels]
      Bob, I'm trying to choose between these two hotels.  Any opinions?
3.2   [Bob draws an arrow, speaks, and points]
      Well, this area is really nice to visit. You can walk there from
      this hotel.
      Map scrolls to indicated area.  Hotel selected.
3.3   [Sara speaks] Do you think I should visit Alcatraz?
3.4   [Bob speaks] Map, show video of Alcatraz.
      Video appears.
3.5   [Bob speaks] Yes, Alcatraz is a lot of fun.
```

For this system, the main research focus is on how to generate the most appropriate interpretation for the incoming streams of multimodal input. Besides providing a user interface *to* a dynamic set of distributed agents, the application is built *using* an agent framework, with the OAA helping coordinate competition and cooperation among information sources, which work in parallel to resolve the ambiguities arising at every level of the interpretation process:

- Low-level processing of the data stream: Pen input may be interpreted as a gesture (e.g., 2.5: crossout) by one algorithm, or as handwriting by a separate recognition process (e.g., 2.3: "info?"). Multiple hypotheses may be returned by a modality recognition component.
- Anaphora resolution: When resolving anaphoric references, separate information sources may contribute to resolving the reference:
  - Context by object type: For an utterance such as "show photo of the hotel", the natural language component can return a list of the last hotels talked about.
  - Deictic: In combination with a spoken utterance like "show photo of this hotel", pointing, circling, or arrow gestures might indicate the desired object (e.g., 2.7). Deictic references may occur before, during, or after an accompanying verbal command.
  - Visual context: Given the request "display photo of the hotel", the user interface agent might determine that only one hotel is currently visible on the map, and therefore this might be the desired reference object.

- Database queries: Information from a database agent can be combined with results from other resolution strategies. Examples are "show me a photo of the hotel in Menlo Park" and 2.2.
- Discourse analysis: Discourse can provide a source of information for phrases such as "No, the other one" (or 2.8).

  This list is by no means exhaustive. Examples of other resolution methods include spatial reasoning ("the hotel between Fisherman's Wharf and Lombard Street") and user preferences ("near my favorite restaurant").

- Cross-modality influences: When multiple modalities are used together, one modality may reinforce or disambiguate the interpretation of another. For instance, the interpretation of an arrow gesture may vary when accompanied by different verbal commands (e.g., "scroll left" vs. "show info about this hotel"). In the latter example, the system must take into account how accurately and unambiguously an arrow selects a single hotel.

- Addressee: With the addition of collaboration technology, humans and automated agents all share the same workspace. A pen doodle or a spoken utterance may be meant for either another human, the system (3.1), or both (3.2).

The implementation of the Multimodal Map application exploits several features of the OAA:

- Reference resolution and task delegation are handled in a distributed fashion by the parallel parameters of oaa_Solve, with meta-agents encoding rules to help the facilitator make context- or user-specific decisions about priorities among knowledge sources.

- Basic multiuser collaboration is handled through OAA's built-in data management services. The map user interface publishes data solvables for elements such as icons, screen position, and viewers, and defines these elements to have the attribute "shareable". For every update to this public data, the changes are automatically replicated to all members of the collaborative session, with associated callbacks producing the visible effect of the data change (e.g., adding or removing an icon).

- Functionality for recording and playback of a session is easily implemented by adding agents as members of the collaborative community. These agents either record the data changes to disk, or read a logfile and replicate the changes in the shared environment.

- The domain-specific code for interpreting travel planning dialog is cleanly separated from the speech, natural language, pen recognition, database and map user interface agents. These components were reused without modification to add multimodal map capabilities to other applications for activities such as crisis management, multi-robot control, and the MVIEWS tools for the video analyst.

10 Related Work

Agent-based systems have shown much promise for flexible, fault-tolerant, distributed problem solving. Much of the foundational work on agent technology has focused on interagent communication protocols (Finin et al., 1997), patterns of conversation for agent interactions (FIPA, 1997), and basic facilitation capabilities, including agent name servers and other types of registry services (*e.g.*, brokers, matchmakers) (Sycara et al., 1996).

Because there is insufficient space here to cover the gamut of work on agent architectures, we restrict ourselves to mentioning several projects that have helped to evolve some notion of facilitation. Genesereth has emphasized the role of a facilitator (Genesereth and Singh, 1993; Genesereth and Katchpel, 1994), and in (Genesereth and Singh, 1993) describes a facilitator based on logical reasoning. This facilitator shares our emphasis on content-based routing and the synthesis of complex multistep delegation plans, but does not go as far as OAA in allowing the service requester to influence the strategies used by the facilitator. Similarly, the InfoSleuth system (Nodine and Unruh, 1997) employs matchmaking agents having the ability to reason deductively about whether expressions of requirements (by requesters) match with the advertised capabilities of service providers. KQML (Labrou and Finin, 1997; Finin et al., 1997) provides "capability-definition performatives", such as *advertise*, and "facilitation performatives", such as *broker_one* and *broker_all*. While these performatives may be suitable for structuring the basic interactions between the players in a facilitated system, it should be noted that they provide only a communication protocol. That is, the specific strategies employed by a facilitator, and the means of advising a facilitator in selecting a strategy, are beyond the scope of KQML specifications. Sycara et al. delineate the concepts of matchmaking, brokering, and facilitation in a useful way, and explore the tradeoffs inherent in the use of these approaches. Overall, they find that a brokered or facilitated system can exhibit dramatically better performance than one based on matchmaking.

11 Future Directions

Much work remains to be done, both at implementation and conceptual levels. Areas for further investigation include scalability, robustness (fault tolerance), improved development and runtime tools, and improved facilitation strategies and services.

The use of facilitators offers both advantages and weaknesses with respect to scalability and fault tolerance. On the plus side, the grouping of a facilitator with a collection of client agents provides a natural building block from which to construct larger systems. On the minus side, there is the potential for a facilitator to become a communication bottleneck, or a critical point of failure. In tasks requiring a sequence of exchanges between two agents, it is possible for a facilitator to assist them in finding one another and establishing communication, but then to step out of the way while they communicate over a direct, dedicated channel. This is a relatively straightforward extension to our approach, which we plan to incorporate. For more complex task configurations, we see three general areas to explore in addressing these issues. First, a variety of multifacilitator topologies can be exploited in constructing large systems.

It would be useful to investigate which of these exhibits the most desirable properties with respect to both scalability and fault tolerance. Second, it is possible to modularize the facilitator's key functionalities. For example, goal planning (delegation and optimization) can readily be separated from goal execution. Given this, one can envision a configuration in which the execution task is distributed to other agents, thus freeing up the facilitator. Third, we would like to incorporate mechanisms for basic transaction management, periodically saving the state of agents (both facilitator and client), and rolling back to the latest saved state in the event of the failure of an agent.

With respect to agent development tools, we plan on updating our initial work in this area (described at PAAM96 in (Martin et al., 1996)) to a more group-oriented and web-centric design. Improvements to the linguistic tools, and a graphical monitoring agent would also be desirable.

While much work has been done by agent researchers to demonstrate increased autonomy of individual agents (particularly in the category of information filtering and personal assistants), smarter and more autonomous facilitators (or other means of coordinating multiple agents) are likely to be more critical to the evolution of multiagent systems. Our experience to date has shown value in the handling of compound goals, with advice parameters, by facilitators. However, the advice is still relatively simple, and the discretion exercised by the facilitator relatively limited. Thus, we are interested in exploring the use of more sophisticated strategies by the facilitator, guided by a higher level of advice. It may be possible to draw upon existing work in the (artificial intelligence) field of planning and the (database) field of query planning. Facilitation is also likely to benefit from richer representations of agents' capabilities.

12 Summary

The Open Agent Architecture provides a framework for the construction of distributed software systems, which facilitates the use of cooperative task completion by flexible, dynamic configurations of autonomous agents. We have presented the rationale underlying its design, compared its features to those of other distributed frameworks, and summarized the applications built to date using it. In addition, we have described the major components of OAA infrastructure, and the mechanisms used in assembling an agent-based system. These mechanisms include a general approach to achieving cooperation between agents, organized around the declaration of capabilities by service-providing agents, the construction of goals by users and service-requesting agents, and the role of facilitators in coordinating the satisfaction of these goals, subject to advice and constraints that may accompany them; facilities for creating and maintaining shared repositories of data; and the use of triggers to instantiate commitments within and between agents.

References

Adam Cheyer and Luc Julia. 1995. Multimodal maps: An agent-based approach. In *Proc. of the International Conference on Cooperative Multimodal Communication (CMC/95)*, Eindhoven, The Netherlands, May. Also available at http://www.ai.sri.com/~oaa/ + "Bibliography".

Adam Cheyer and Luc Julia. 1998. Mviews: Multimodal tools for the video analyst. In *Proceedings of the 1998 International Conference on Intelligent User Interfaces (IUI98)*, San Francisco, California, January.

Adam Cheyer, Luc Julia, and Jean-Claude Martin. 1998. A unified framework for constructing multimodal applications. In *Proceedings of the 1998 Conference on Cooperative Multimodal Communication (CMC98)*, San Francisco, California, January.

Philip R. Cohen, Adam J. Cheyer, Michelle Wang, and Soon Cheol Baeg. 1994. An open agent architecture. In O. Etzioni, editor, *Proc. of the AAAI Spring Symposium Series on Software Agents*, pages 1–8, Stanford, California, March. American Association for Artificial Intelligence.

Tim Finin, Yannis Labrou, and James Mayfield. 1997. KQML as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, Cambridge.

FIPA. 1997. Foundation for intelligent physical agents (FIPA) 1997 specification. Available online at http://drogo.cselt.stet.it/fipa/spec/fipa97.htm.

D. Gelernter. 1993. *Mirror Worlds*. Oxford University Press, New York.

Michael R. Genesereth and Richard E. Fikes. 1992. Knowledge interchange format version 3.0 reference manual. Technical Report Logic-92-1, Stanford University, Stanford, CA. Also available online at http://logic.stanford.edu/kif/kif.html.

M. R. Genesereth and S. P. Katchpel. 1994. Software agents. *Communications of the ACM*, 37(7):48–53.

M. R. Genesereth and N. P. Singh. 1993. A knowledge sharing approach to software interoperation. Technical Report Logic-93-1, Department of Computer Science, Stanford University, Stanford, CA.

Didier Guzzoni, Adam Cheyer, Luc Julia, and Kurt Konolige. 1997. Many robots make short work: Report of the SRI international moble robot team. *AI Magazine*, 18(1):55–64.

Yannis Labrou and Tim Finin. 1997. A proposal for a new KQML specification. Technical Report CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, February. Also available online at http://www.cs.umbc.edu/kqml/.

David L. Martin, Adam Cheyer, and Gowang-Lo Lee. 1996. Agent development tools for the open agent architecture. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 387–404, Blackpool, Lancashire, UK, April. The Practical Application Company Ltd.

David L. Martin, Hiroki Oohama, Douglas Moran, and Adam Cheyer. 1997. Information brokering in an agent architecture. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, Blackpool, Lancashire, UK, April. The Practical Application Company Ltd.

Microsoft. 1996. Distributed component object model protocol – DCOM/1.0. Available online at http://www.microsoft.com/activex/ + DCOM.

Robert Moore, John Dowding, Harry Bratt, J.M̃ark Gawron, Yonael Gorfu, and Adam Cheyer. 1996. Commandtalk: A spoken-language interface for battlefield simulation. Technical report, Artificial Intelligence Center, SRI International, 21 June. Also, http://www.ai.sri.com/natural-language/projects/arpa-sls/apps.html.

Douglas B. Moran and Adam J. Cheyer. 1995. Intelligent agent-based user interfaces. In *Proc. of International Workshop on Human Interface Technology 95 (IWHIT'95)*, pages 7–10, Aizu-Wakamatsu, Fukushima, Japan, 12-13 October. The University of Aizu. Also available at http://www.ai.sri.com/∼oaa/ + "Bibliography ...".

Douglas B. Moran, Adam J. Cheyer, Luc E. Julia, and David L. Martin. 1997. The open agent architecture and its multimodal user interface. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces (IUI97)*, Orlando, Florida, 6-9 January.

M. H. Nodine and A. Unruh. 1997. Facilitating open communication in agent systems: the InfoSleuth infrastructure. Technical Report MCC-INSL-056-97, Microelectronics and Computer Technology Corporation, Austin, Texas 78759, April.

Object Management Group (OMG). 1997. The complete CORBA/IIOP 2.1 specification. Available online at http://www.omg.org/corba/corbiiop.htm.

A. Rao and M. Georgeff. 1995. BDI agents from theory to practice. Technical Note 56, AAII, April.

David G. Schwartz. 1995. *Cooperating Heterogeneous Systems*. Kluwer Academic Publishers, Dordrecht.

K. Sycara, K. Decker, and M. Williamson. 1996. Matchmaking and brokering. In *Proc. of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, December.

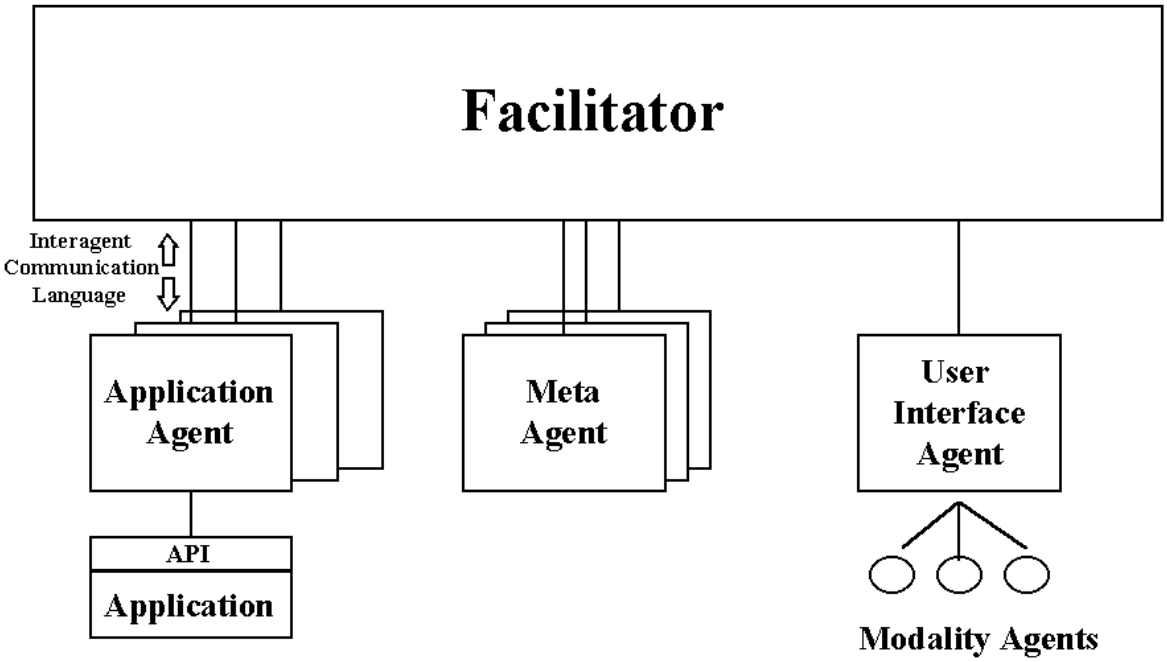

Figure 1: OAA System Structure.

Application	Description
Automated Office	Mobile interfaces (PDA with telephone) to integrated community of commercial office applications (calendar, database, email) and AI technologies (speech recognition, speaker identification, text to speech, natural language interpretation and generation). (Cohen et al., 1994)
Unified Messaging	Adaptable, ubiquitous access to email, fax, voice, and Web messages and services
Multimodal Map	Pen/voice interface to distributed Web data. (Cheyer and Julia, 1995)
InfoWiz	Animated voice interactive interface to the Web.
ATIS-Web	Try out a live demo of speech recognition over the Web! Available at http://www.speech.sri.com/demos/atis.html
CommandTalk	Spoken-language interface for controlling simulated forces. (Moore et al., 1996)
Spoken Dialog Summarization	Real-time system for summarizing human-human spontaneous spoken dialogs (Japanese).
Language Tutoring	Speech recognition for foreign language learning, incorporating user modeling for adaptive lessons.
Disaster response	Collaborative, wireless map-based interface for emergency response teams.
MVIEWS	Integrating speech, pen, natural language, image processing and other technologies for the video analyst. (Cheyer and Julia, 1998)
OAA InfoBroker	Mediated facilitation of heterogeneous structured and semistructured (Web) datasources. (Martin et al., 1997)
OAA Rental Agent	Monitors the Web and notifies user when housing classifieds meet user specifications.
Agent Development Tools	Guides the agent developer through the steps required to create new agents. (Martin et al., 1996)
Multi-Robot Control	Team of robots works together on assigned tasks (1st place, AAAI Office Navigation Event). (Guzzoni et al., 1997)
Surgical Telepresence	Force feedback training simulator for endoscopic surgery. All physical and virtual entities modeled as OAA agents.

Table 1: A partial list of applications written using OAA.

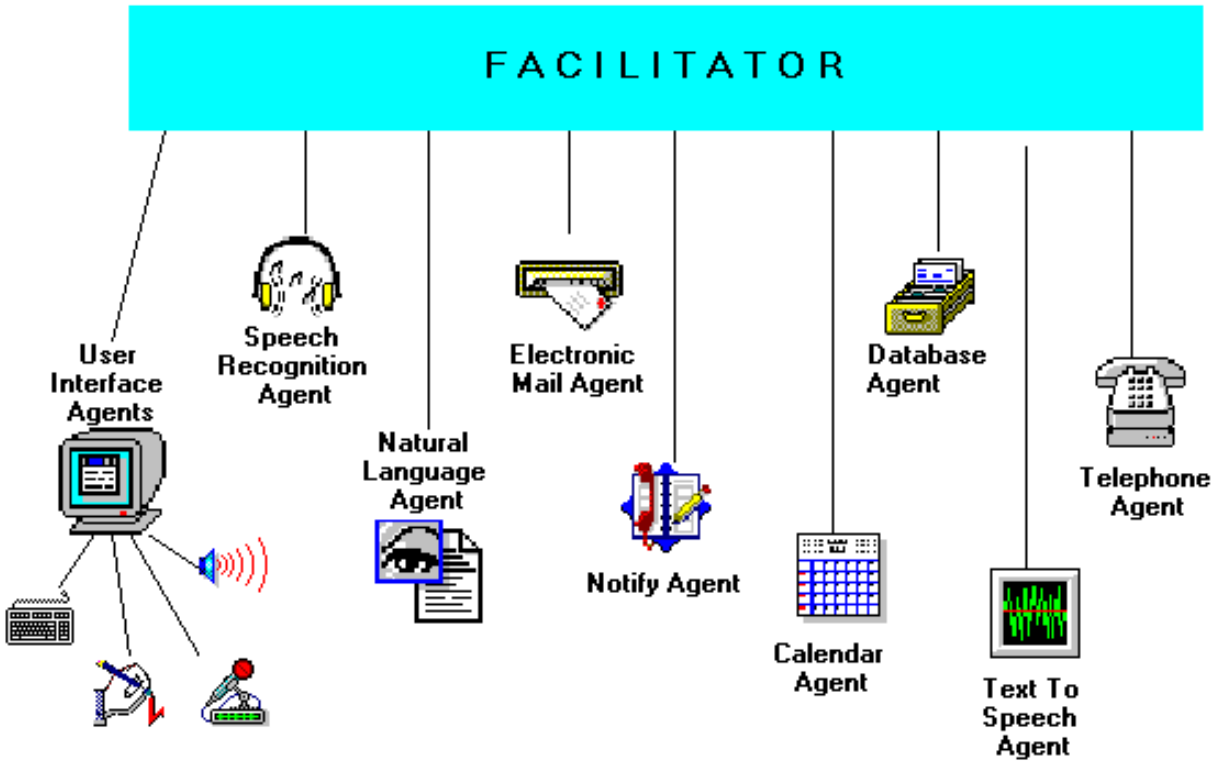

Figure 2: Automated Office Agents.

Figure 3: User Interface for Automated Office Application.

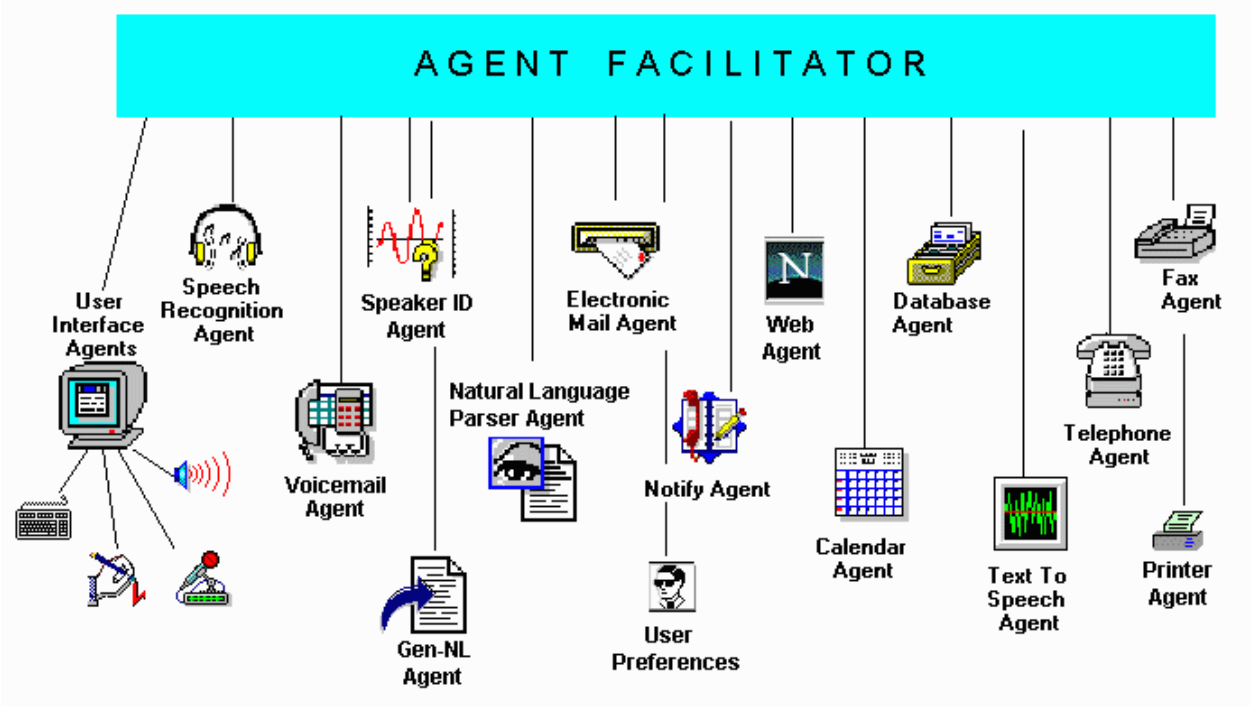

Figure 4: Unified Messaging Agents.

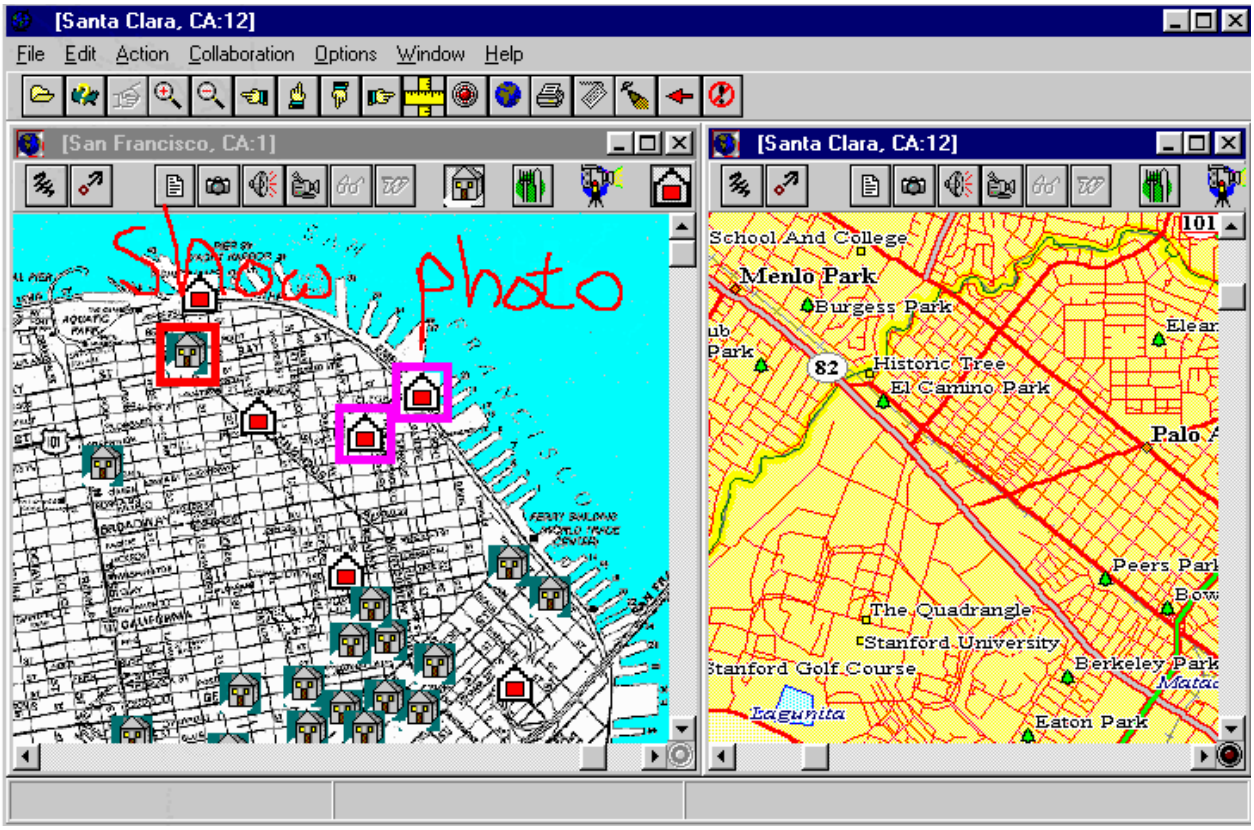

Figure 5: Multimodal Map Application.