Building Distributed Software Systems with the Open Agent Architecture

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

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 how agent-based systems are constructed with OAA. To provide technical context, we describe the motivations for its design, and situate its features within the realm of alternative software paradigms. A summary is given of OAA-based systems built to date. 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.

1 Introduction

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, audio and video streams, and with the popularization of Java, may include programs which can be downloaded and executed on the local machine. As the world continues to evolve towards a more networked computing model where remote computers take on an expanded role not only for storing public data but also for providing processing services, we will need new programming models which allow for flexible composition of distributed processing elements in a dynamically changing and relatively unstable environment.

In Section 2 of this paper, we first review various approaches to distributed computing, and then situate our own approach, the Open Agent Architecture (OAA)[1], within the scope of this related work. Subsequent sections will provide detailed descriptions of the inner workings of OAA. Whereas the motivating concepts for an early version of OAA were presented in [4], and certain OAA-based systems have been described in [2], [12], [13], [15], and [17], this is the first paper to present a detailed technical explanation of how systems are constructed using OAA.

2 Technologies for Distributed Computing

In this section, we will briefly review the overall concepts, advantages and disadvantages of several approaches to distributed computing, including distributed objects, mobile objects, agent-based software engineering, and blackboard-style architectures.

2.1 The Distributed Object Approach

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

- encapsulation: encapsulation encourages the creation of library interfaces which minimize dependencies on underlying algorithms or data structures. Changes to programming internals can be made at a later date with requiring changes to the interfaces and of the code code which 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.

[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.

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 [19] or Microsoft's DCOM [14] allow the creation of programs whose components may be spread across multiple machines. To implement a client-server relationships 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 distributed environment. In addition, RPC uses a blocking (synchronous) scheme which does not scale well for high-volume transactions.

2.2 Mobile Objects

Mobile objects, sometimes called mobile agents, are bits of code which 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 Odyssey by General Magic, Concordia by Mitsubishi, and Voyager by ObjectSpace.

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

- Network bandwidth: for certain types of 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 similar fashion to 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 [20] or Gelernter's LINDA [6], 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 powerful 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 inter-process communication. This advantage is also a disadvantage sometimes: although a programmer does not need to refer to a specific process during computation, the framework does not providing support 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 among autonomous entities. Efficient communication among participants requires four components: 1) a transport mechanism which will carry messages in an asynchronous fashion; 2) an interaction protocol which defines various types of communication interchanges and their social implications (for instance, a response is expected of a question); 3) a communication language which permits 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 distributed object's RPC model or a blackboard architecture's centralized exchange approach.

Undoubtably the most widely-used framework for agent-based software engineering is that of the Knowledge Query and Manipulation Language (KQML) [11, 22]. KQML, which implements a transport and interaction protocol, is often used in conjunction with the Knowledge Interchange Format (KIF) communication language, and either ad hoc or more formalized ontologies.

2.5 Open Agent Architecture

The motivations for the our approach to distributed computing share much in common with the paradigms we have outlined above. Like distributed object frameworks, the primary goal of the OAA is to provide a means for integrating heterogeneous, commercial applications in a distributed infrastructure. However, we would like to add the dynamic nature and extensibility of the blackboard approaches, the efficiency of mobile objects, and the rich and complex interactions of communicating agents.

Perhaps the best way to obtain an intuitive sense of the characteristics and strengths of a programming methodology is to look at how it has been applied to one or more real applications. Table 1 provides several OAA-based applications from which we will take examples to illustrate qualities important to the framework.

User Delegation. Users want to be able to delegate a task to the agent community without having to specify how and who will perform each subpiece of every command. As an example from the Automated Office application, the request "When mail arrives for me from David, get it to me immediately" produces coordinated activity among 15 independent agents to achieve the goal. What the user can say and do is a function of the agents who are connected to the network at that moment.

Parallel competition and cooperation. In an example from the Multimodal Map application, in which a user issues commands on a map by drawing, writing and speaking, the spoken phrase "Show a photo of the hotel." will create competitive interactions among several agents. Is the hotel the one the user has been talking about (natural language agent), the one he is looking at (map interface), or the one he will point at in a few seconds (gesture recognition)? Given "Show a photo of the hotel on Smith Street", the database agent must cooperate with the other competing agents to help resolve this reference.

Reuse. Although the Multimodal Map application was designed for a travel planning task, it was reused without code changes to monitor and control multiple robots [10], to display map positions for objects being tracked in a live video [3], and to perform Wizard-of-Oz user study simulations [1].

Adaptive. When the monitor broke during a demonstration of the Automated Office system, the agent community found an unanticipated way to present answers to user queries – using text-to-speech over the telephone.

Given this introduction to some of the motivations and capabilities of the OAA, the rest of this paper will concern itself with providing a detailed description of how the OAA works and how to build distributed applications using it.

3 Overview of 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

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, etc.).
InfoWiz	An animated voice interactive interface to the web. [4]
ATIS-Web	Try out a live demo of speech recognition over the web!
CommandTalk	Spoken-language interface for controlling simulated forces. [15]
Spoken Dialog Summarization	A real-time system for summarizing human-human spontaneous spoken dialogues (Japanese).
Language Tutoring	Speech recognition for foreign language learning, incorporating user modeling for adaptive lessons.
Multimodal Map	Pen/Voice interface to distributed web data.
Disaster response	A collaborative, wireless map-based interface for emergency response teams.
MVIEWS	Integrating speech, pen, NL, image processing and other technologies for the video analyst. [3]
OAA InfoBroker	Mediated facilitation of heterogeneous structured and semi-structured (Web) datasources. [13]
OAA Rental Agent	Monitors the web and notifies you when housing classifieds meet your specifications.
Agent Development Tools	Guides the agent developer through the steps required to create new agents. [12]
Multi-Robot Control	A team of robots works together on assigned tasks (1st place, AAAI Office Navigation Event). [10]
Surgical Telepresence	Force feedback training simulator for endoscopic surgery. All physical and virtual entities are modeled as OAA agents.

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

common relationship to a facilitator agent.

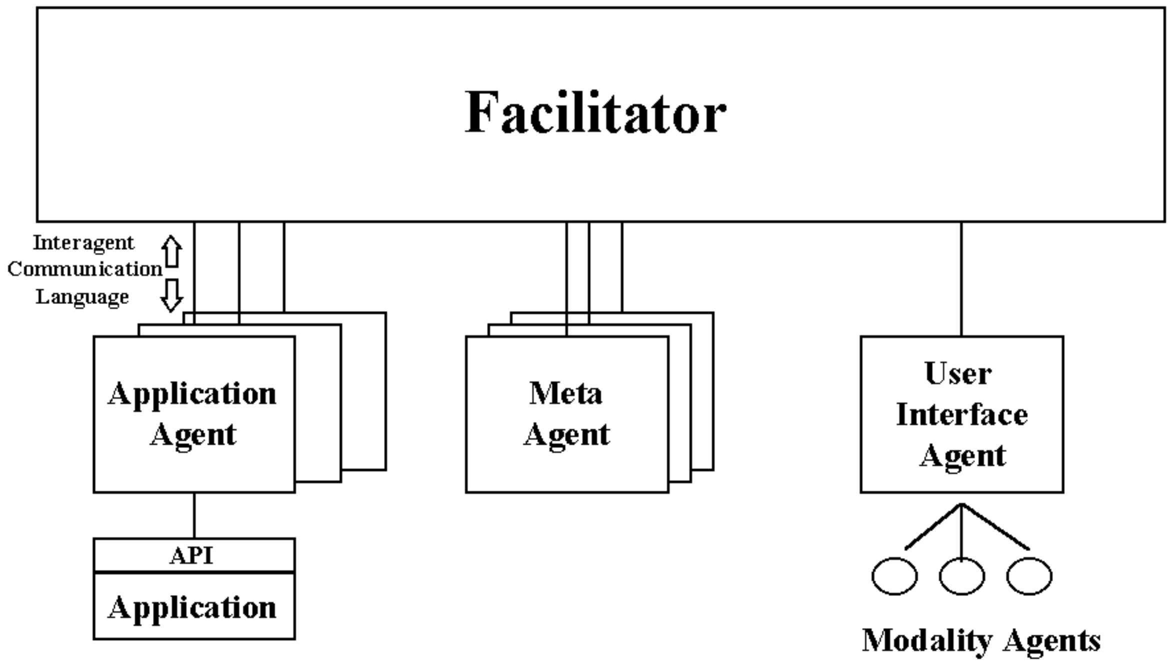

Figure 1: OAA System Structure.

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 (such as speech recognition, natural language processing, email, and some forms of data retrieval and data mining), 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 help advise the facilitator agent during its multi-agent coordination phase. While the facilitator possesses a number of domain-independent coordination strategies, meta-agents augment these using application-specific knowledge or reasoning (e.g. rules, learning algorithms, planning, and so forth). An example of a meta-agent would be the *notification* agent from the Office Assistant application. The notification agent makes use of rules concerning the optimal use of different modalities (email, fax, speech generation over the telephone) plus information about an individual user's preferences to determine the best way of of relaying a message using available media transfer application agents.

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 is capable of providing. 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.

In the following sections, we give a more detailed description of how an OAA system is assembled. We order the presentation top-down, beginning with the means by which a group of agents work together, then considering the mechanisms that support the use of shared data repositories and temporal task specifications, and finally describing some of the basic infrastructure underlying communications between agents and the construction of individual agents. Although there is insufficient space to cover all of its technical aspects, we nevertheless hope to give the flavor of the technology using examples and discussion of the most important points.

4 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 3-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.

4.1 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 unspecified, 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 most important 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 that are 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 makes use of 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 which 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, in order 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 delegate the subrequests individually.

A number of important declarations and other program elements are represented using ICL expressions. These include, in addition to events, capabilities declarations, requests for services, responses to requests, trigger specifications, and shared data elements. In subsequent sections, we consider each of these elements.

4.2 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 establish a high-level interface to the agent, which 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 a number of 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, as well as by other agents (provided they have the required permissions). Third, there are a variety of special features available for use with data solvables, which facilitate maintaining the associated facts. Some of these are mentioned below, in Section 5.

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 services is described in Section 4.3 below.

A request for one of an agent's services normally arrives in the form of an event from the agent's facilitator. This event is then handled by the appropriate handler. 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.

4.2.1 Specification of Solvables

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

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

The goal of a solvable, which syntactically takes the form of an ICL structure, gives a logical representation of what service is 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, if desired.

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 callback function to be associated with the solvable. Some of the parameters appropriate for a *data* solvable are mentioned in Section 5.

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 has value in two types of situations: when the agent intends the solvable to be solely for its internal use, and wants to take advantage of OAA mechanisms in accessing it; and 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 non-private 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 if desired. (*Calling* means requesting the service encapsulated by a solvable, whereas *writing* means modifying the collection of facts associated with a data solvable[2].) 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.

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

```
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. Note that most parameters and permissions have default values, and specifications of default values may be omitted from the parameters and permissions lists.

When a programmer defines an agent's capabilities in terms of solvable declarations, he or she 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 [12]) 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 else on ontologies constructed during the incremental development of a body of agents for a number of applications.

Although the OAA imposes no hard restrictions (other than the basic syntax) on the form of solvable declarations created by a programmer, several recommendations try to promote "good OAA interface style". These include:

- Classes of items are often tagged by a particular type. For instance, in the example above, the 'email' parameter of "last_message" and "get_message" does not add specific information to the predicate, but serves during an ICL request 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 will often serve to hold optional information usable by the function. The ICL

[2]There are also permissions allowing an outside agent to *read* and/or *redefine* the handler associated with a procedure solvable, but these capabilities are not fully developed, and are presently available only with agents implemented in Prolog.

expression 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 ICL request 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 right to effectuate the action.

4.3 Requesting Services

An agent requests services by sending goals to its facilitator. Each goal contains calls to one or more solvables. It is important to understand that calling a solvable does *not* require that the agent specify (or even know of) a particular agent to handle the call. While it is possible to specify one or more agents to handle a call (and there are situations in which this is desirable), in general it is advantageous to leave this delegation task to the facilitator.

The OAA libraries provide an agent with a single, unified point of entry for requesting services of other agents: the library procedure *oaa_Solve*, which has been introduced above. 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.

4.3.1 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. There are also several significant extensions to Prolog syntax and semantics; three are of particular interest here. First, there is a "parallel disjunction" operator that indicates 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[3]. 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 expressions: unique names, symbolic names, and shorthand names. Every agent has a unique name, assigned by its facilitator, which relies upon network

[3]This capability is under development at the time of writing.

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 4.5.

4.4 Refining Service Requests

The parameters associated with a goal (or subgoal) can draw on a number of useful features to refine the request's meaning. We have mentioned elsewhere the ability to specify whether or not the 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 non-compound 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 meta-data.

- *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 multi-facilitator system, *level_limit* may be used to say how 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 a goal is appropriate, and whether the requester itself may be considered a candidate solver of the subgoals of a request.
- *Meta-data parameters* allow a service requester to receive feedback from the facilitator about the handling of a goal. 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.

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. In the following section, we describe the facilitator's approach to doing this.

4.5 Facilitation

Facilitation plays a central role in OAA. At its core, our notion of facilitation is similar to that proposed by Genesereth ([7]) 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 three respects.

First, it encompasses a very general notion of *transparent delegation*. This 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, which are introduced above (Section 4.3.1). This involves three types of processing: *delegation*, that is, completion of the addresses embedded within a compound goal; *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 multi-facilitator 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. Some of these are mentioned above (Section 4.4), and some additional possibilities under consideration are mentioned in Section 9.

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 inter-agent 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.

5 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 4.2.1. 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 is often 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 non-persistent, 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, distributed, agents.

Similarly, the *meta-data* 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; they can also be maintained by a facilitator if desired, 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. The following subsection provides an example of this usage.

5.1 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 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 that they are prepared to respond to, 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 for this case.)

In the Office Assistant system, a number of 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)])
```

6 Autonomous Monitoring Using 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 triggers. 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.

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 solvable data set (which can be maintained on a facilitator or a client agent). 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* are monitored after the processing of each incoming event, and also whenever a timeout occurs in the event polling. Task triggers specify a test that must succeed for the trigger to fire. This test may specify any goal executable by the local ICL interpreter, and most often is used to test when some solvable becomes satisfiable.

  Task triggers are useful for 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 (eg. "On december 23rd at 3pm"), or on a recurring basis (eg. "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.

7 Basic Infrastructure

7.1 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, Java, Lisp, Visual Basic, and Delphi. As mentioned above, 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 provides several families of procedures, which 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 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 then 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, as is explained below). 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.

7.2 The Event Loop

Although there may be exceptions, each client agent is normally structured around an *event loop*, the functionality of which is provided by the agent library. The operation of the event loop is to repeatedly check the agent's event queue, to see if any events have arrived from the agent's facilitator. When an event arrives, it is handled in one of three ways, depending on its type:

- If it is a *built-in* event, it is handled automatically by the agent library.
- If it is a *task-specific* event, that is, an event corresponding to one of the agent's procedure solvables, the agent library calls the developer-defined callback procedure, with the event and a parameter list as arguments.
- Occasionally there is a *hybrid* event; that is, an event that is normally *built-in*, but its handling in certain situations is left to the developer. In these situations, the developer has access to these events (but may also ignore them if desired). An example of this is when results of a call to *oaa_Solve* are returned asynchronously.

For agents that provide a traditional user interface, support is provided for integrating the agent event loop with the event loops of X Windows and other graphical systems.

8 Related Work

Agent-based systems have shown much promise for flexible, fault-tolerant, distributed problem solving. Much of the work on agent technology has focused on interagent communication protocols [22], patterns of conversation for agent negotiation [5], and basic facilitation capabilities including agent name servers and other types of registry services (*e.g.*, brokers, matchmakers) [21].

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 [9, 8], and, in [9], 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 doesn't go as far as OAA in allowing the service requester to influence the strategies used by the facilitator. Similarly, the InfoSleuth system ([18]) 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 [11, 22] 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. ([21]) 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.

9 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 communications 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 communications, 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, there are a variety of multi-facilitator topologies that 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 [12]) 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.

10 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 how they are used in assembling an agent-based system.

References

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

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

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

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

[5] Federation for intelligent physical agents (FIPA) 1997 specification. Available online at http://http://drogo.cselt.stet.it/fipa/spec /fipa97.htm, November 1997.

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

[7] M. Genesereth and N. P. Singh. A knowledge sharing approach to software interoperation. Computer Science Department, Stanford University, unpublished ms., January 1994.

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

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

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

[11] Yannis Labrou and Tim Finin. 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 1997. Also available online at http://www.cs.umbc.edu/kqml/.

[12] David L. Martin, Adam Cheyer, and Gowang-Lo Lee. 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 1996. The Practical Application Company Ltd.

[13] David L. Martin, Hiroki Oohama, Douglas Moran, and Adam Cheyer. 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 1997. The Practical Application Company Ltd.

[14] Microsoft. Distributed component object model protocol – dcom/1.0. Available online at http://www.microsoft.com/activex/ + DCOM, November 1996.

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

[16] Douglas B. Moran and Adam J. Cheyer. 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 1995. The University of Aizu. Also http://www.ai.sri.com/∼oaa/ + "Bibliography ...".

[17] Douglas B. Moran, Adam J. Cheyer, Luc E. Julia, and David L. Martin. 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 1997.

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

[19] Object Management Group (OMG). The complete corba/iiop 2.1 specification. Available online at http://www.omg.org/corba/corbiiop.htm, September 1997.

[20] D. G. Schwartz. Cooperating heterogeneous systems: A blackboard-based meta approach. Technical Report 93-112, Center for Automation and Intelligent Systems Research, Case Western Reserve University, Cleveland, Ohio, April 1993. Unpublished Ph.D. thesis.

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

[22] Yannis Labrou Tim Finin and James Mayfield. Kqml as an agent communication language. In Jeff Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, 1997. To appear.