Development Tools for the Open Agent Architecture*

David L. Martin, Adam Cheyer
SRI International

Gowang-Lo Lee
ETRI

Abstract

The agent-based paradigm for software systems cannot realize its full potential, and will not become widespread, until adequate agent development tools and environments are available. To address this need, an exploration of the requirements for such tools and environments has been conducted in the context of the Open Agent Architecture (OAA) project, and has resulted in the creation of the Agent Development Toolkit (ADT). The ADT provides a variety of mechanisms that support the specification and implementation of individual agents, as well as cooperating communities of agents. Special attention has been given to tools that enable an agent developer to construct intelligent user interfaces, which allow users to express their requests of agents using spoken and written natural language in combination with other modalities. This paper discusses a number of general requirements that were identified for agent development environments, reports on the design and functionality of the ADT, and shows how the ADT addresses those requirements. In addition, we describe our experience to date in constructing OAA-based agent systems, and future directions in extending the ADT.

*This paper was supported by a contract from the Electronics and Telecommunications Research Institute (Korea), and will be presented at The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 96), London, April 1996. The first author can be reached by email at martin@ai.sri.com.

1 Introduction

A number of important and interesting investigations have recently been made into the languages, architectures, algorithms, and formal analyses of agent-based systems, and substantial agent-based systems are being fielded in a variety of domains. There are good reasons for this. The notion of autonomous, cooperative, and intelligent agents as fundamental system building blocks provides an evocative metaphor and a natural paradigm for harnessing explosive increases in interconnectivity and information access. From a system developer's perspective, this paradigm holds the promise of constructing flexible, adaptable systems that provide intelligent services based on the cooperative efforts of the most capable and most appropriate agents for the job at hand, selected from a potentially vast array of distributed software and hardware resources. [1]

While the results of these investigations provide many valuable elements of infrastructure for agent-based systems, it must be recognized that the agent-based approach cannot realize its full potential, and will not become widespread, until adequate agent development tools and environments are available. To date, very little has been done to address this need.

There are a number of interesting questions to be addressed: What new requirements and challenges arise for development tools that are unique to agent-based systems? How does the inherent autonomy and loose coupling of agents affect the development process and the resulting artifacts such as documentation? How can we best facilitate the construction of a collection of interoperable agents written in various languages and operating on various platforms, and agents derived from existing applications and legacy information sources? How much of the creation of an agent-based system can be automated?

An agent system that provides an intelligent user interface — allowing users to express their requests by using spoken and written natural language in combination with other modalities — raises additional challenges regarding development environments. For example, one important question is how best to provide support for the agent developer, who is not likely to be a computational linguist, in tailoring the linguistic processing components of the system to handle the domain-specific expressions that may be expected to appear in users' requests. [2]

An exploration of these questions has been conducted in the context of the Open Agent Architecture (OAA) project, and has resulted in the creation of the Agent Development Toolkit (ADT). This paper is concerned with the requirements that motivated the creation of the ADT, and the functionality that evolved to meet those requirements. The following section presents a general discussion of requirements that are characteristic of development environments for agent-based systems. In Section 3 we give an overview of the OAA, and of results to date in constructing OAA-based agent systems. Section 4 shows in some detail how many of the requirements mentioned earlier have been addressed by the ADT. Finally, in Section 5, we draw conclusions and mention some current and proposed work to extend the OAA and the ADT.

[1] Because of the wide variety of systems to which the word 'agent' has been applied recently, it may be helpful to indicate what we mean by 'agent-based system'. The type of system we have in mind is one in which the services provided are accomplished through the cooperative efforts of a number of independent software processes, each of which is persistent and acts with a high degree of autonomy.

[2] Most other important areas of exploration in agent-based systems — learning, mobility, negotiation, and so forth — also introduce new challenges for development environments.

2 Challenges for Agent Development Environments

In highlighting some of the general requirements and challenges that can be identified for development environments for agent-based systems, we are not attempting to give an exhaustive list. We do believe that the points mentioned here are applicable to most agent-based systems. In describing the Open Agent Architecture in Section 3, we will be able to show in greater detail how these requirements arise in that particular context, and in Section 4 we show how they are addressed by the ADT.

2.1 Supporting Conformance

Because of the emphasis on interoperability inherent in agent-based systems, there is a critical need for each agent to be designed so as to interact correctly (that is, in accordance with protocol) with the other agents in the system. Thus, an agent development environment should guide the developer in adhering to the protocols used by the system.

Some form of this requirement has existed in all software development paradigms; after all, even in the simplest programs, procedure calls must match the appropriate procedure declarations. However, the need for conformance is likely to be more strenuous in agent-based systems, in two respects. First, agent programming interfaces and interactions between agents — and hence, the protocols for specifying these — tend to be more complex than interfaces and interactions between the elements of systems built using traditional approaches. Second, it is a goal of most agent systems that the development teams of the various agents be able to work independently, remotely, and on widely heterogeneous platforms — but while incurring as little overhead as possible due to the interdependencies of agents.

This requirement of conformance applies as strongly to agent *documentation* as it does to agent coding. In particular, the ongoing evolution of an agent-based system by widely distributed and independent groups of developers will require documentation of available agents and their capabilities in a consistent, automatically searchable format.

2.2 Supporting Heterogeneity

In agent development, as in most software development, conformance and heterogeneity are two sides of the same coin: it is precisely because of the need to achieve a meaningful level of interoperability between widely heterogeneous agents that it is critical for agents to conform to the same protocols.

Many different types of heterogeneity can occur in an agent-based system. Three that are of concern from the agent developer's point of view are the multiplicity of implementation languages, the multiplicity of execution platforms, and the mixture of newly created agents with those that have been adapted from legacy applications or information sources.

Thus, the design of an agent development environment (as well as the design of the architecture) should allow for an equal level of support for an agent's development, regardless of its language, platform, or origin.

2.3 Construction of Agent Communities

An agent-based approach encompasses a new definition of "system" (or at least a definition modified in some important ways), and consequently calls for new conceptualizations of what it is to create a "system". Agent-based system construction involves the identification of a set of agents that can do a job together. Wherever possible, parts of a system's functionality are provided by reuse of existing agents, but in any case the determination of what services are provided by existing agents is an essential prerequisite to the design of new agents. Thus, a development environment should make it as easy as possible to manipulate (e.g., locate, browse, inspect, visualize) agents as the basic building blocks of systems. In particular, it should provide support for identifying the capabilities of existing agents. It should also provide support for specifying new configurations of agents for interoperation.

2.4 Running and Debugging Systems

Agent-based approaches also entail changes in what is meant by "system execution". Invoking — and monitoring — an agent-based system can become much more involved than it is under today's predominant software paradigms. Rather than focusing on the behavior of a single process, or a tightly regimented series of client-server interactions, the agent-based system developer needs to be able to initiate and ensure the continued availability of an entire collection of processes running in diverse environments. He must be able to view the global activity of the collection, as well as the local activities of specific agents. These needs call for more powerful execution and debugging aids than currently exist. Thus, an agent-based development environment should provide new mechanisms for instantiating, monitoring, and debugging operational configurations of agents. Agent-based debugging aids will most likely be constructed on models borrowed from the field of simulation.

2.5 Facilitating Use of Support Agents

In our terminology, a *support agent* is one that provides services of great importance to many, if not most, agents operating in a system. Thus, while not a fixed part of the agent system infrastructure, a support agent is thought of as having a more fundamental status than an ordinary application agent, because of the widespread demand for its use. Because of the emphasis in the OAA on intelligent user interfaces, speech recognition and natural language understanding agents have become two very important examples of support agents in the OAA.

Support agents pose special problems for agent development tasks because in many cases they employ sophisticated techniques. As a result, customizing a support agent for a particular task domain is likely to require substantial expertise — a level of expertise that the average agent developer may not possess and may not have the time to acquire.

Because of their quasi-standardized use with the system, however, support agents offer an opportunity to provide knowledge-acquisition tools that support their use. For example, as we show in Section 4.2, the use of speech recognition and natural language understanding agents can be supported with tools for the introduction of natural language vocabulary and concepts relevant to each agent that employs their services.

3 The Open Agent Architecture

The Open Agent Architecture provides a framework for integrating a society of software agents, each possessing a high degree of independence and autonomy, within a distributed environment. A collection of agents satsifies requests from users, or other agents, by acting cooperatively, under the direction of one or more facilitators (which are themselves agents of a special type).

The system's architecture, based loosely on Schwartz's FLiPSiDE system [7], uses a hierarchical configuration in which each application agent connects as a client of a facilitator. Facilitators provide content-based message routing, global data management, and process coordination for their set of connected agents. Facilitators can, in turn, be connected as clients of other facilitators. Each facilitator records the published capabilities of their subagents, and when requests arrive (expressed in the Interagent Communication Language, described below), the facilitator is responsible for breaking them down and for distributing subrequests to the appropriate agents. An agent satisfying a request may require supporting information, and the OAA provides numerous means of requesting data from other agents or from the user.

Agents share a common communication language and a number of basic structural characteristics and capabilities. An agent library provides this common functionality. For example, every agent can install local or remote triggers on data, events or messages; manipulate global data stored by facilitators; and request solutions for a set of goals, to be satisfied under a variety of different control strategies. In addition, the agent library provides functionality for parsing and translating expressions in the Interagent Communication Language, and for managing network communication using TCP/IP. Agents may be implemented (or derived from existing applications) in any programming language to which the agent library has been ported, and may run on any network-linked platform.

The OAA has been described in greater detail in [4].

3.1 The Interagent Communication Language

The OAA's Interagent Communication Language (ICL) is the interface language shared by all agents, no matter what machine they are running on or what computer language they are programmed in. The ICL has been designed as an extension of the Prolog programming language, in order to take advantage of the power of unification and backtracking during interactions among agents.

Every agent participating in an OAA-based system defines and publishes a set of capabilities specifications, expressed in the 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 our use of Prolog as the basis of the ICL, we refer to these capabilities specifications as *solvables*.

For example, in creating an agent for a mail system, 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 solvable corresponding to each of the relations present in the database.

3.2 StartIt

As mentioned in Section 2.4, agent-based architectures introduce strenuous requirements for invoking and monitoring systems of agents. StartIt addresses these requirements, and provides an important bridge between the functionality of the ADT and that of the OAA.

Once a collection of interoperable agents has been assembled to work on a set of tasks, StartIt provides the means of invoking each of the agents on the correct platform, according to the system protocols of that platform, and ensuring that the agent makes the required connection to an OAA facilitator. Of equal importance, StartIt monitors the status of each agent to see that it continues to function correctly. In the event that StartIt detects a failure of one of the agents, it is able to take steps to recover from the failure and automatically restart the agent.

Startup specifications for each agent and instructions on how to deal with failures are contained in configuration files which, as described below, can be automatically generated by a component of the ADT.

3.3 OAA-Based Prototype and Fielded Systems

The OAA has been used as the framework for a number of applications in several domain areas. The first OAA-based system was a multifunctional "office assistant", in which fourteen autonomous agents provide monitoring, communication and management capabilities for business applications such as online calendars, electronic mail, or databases [4]. In a typical scenario, agents with expertise in email processing, text-to-speech translation, notification planning, calendar and database access and telephone control cooperate to find a user and alert him or her of some important message.

The OAA has also been used to construct flexible and natural user interfaces to agent-based and conventional applications. In the CommandTalk system, currently installed at the Marine Corps Air Ground Combat Center at Twentynine Palms, CA, a collection of OAA-enabled agents provide a spoken-English interface to a map-based simulation of armed forces. Another OAA-based multimodal user interface project focuses on techniques for merging simultaneous streams of pen and voice input to form multimedia queries about data retrieved from commercial Internet web sites [2].

4 The Agent Development Toolkit

The Agent Development Toolkit, or ADT, is built around three loosely coupled core components, and presents itself via a user interface component.

- The Programmer's Agent Construction Tool (ProACT) is used by an agent designer to define and maintain the capabilities and other properties of an agent, to manage documentation for the agent, and to generate a code template for the agent.
- The Linguistic Expertise Acquisition Program (LEAP) facilitates the task of interfacing a new agent with existing linguistic support agents such as natural language parsers and speech recognition systems. This involves obtaining semantic information about the domain in which the agent operates, the services provided by the agent, and the English words that will be

useful in composing requests for these services. To make these words useful to the system, LEAP extracts from the agent developer information about their linguistic attributes; it does so by asking the developer simple questions about how and when those words are used. Once the linguistic knowledge has been acquired, LEAP generates or updates the appropriate knowledge bases needed by the linguistic support agents.

- PROJECT allows the developer to create and maintain repositories of reusable agents, and to choose from available repositories to produce an operable configuration of agents for a particular application domain. Once the configuration has been selected, PROJECT can produce a configuration file for use by StartIt, the OAA's system execution manager.
- The user interface component provides integrated access to the features of all three core components. It provides editing capabilities for the artifacts of each core component, such as agent specifications, iconic representations of agents, source code, domain classes and vocabulary, agent repositories, and project configurations.

The ADT has itself been constructed within the OAA. That is, each of the three core components, as well as the user interface, is instantiated as one or more OAA agents. Thus, in constructing the ADT, we were able to take advantage of the benefits of the agent-based paradigm. For example, we were readily able to use a mixture of languages and platforms (some under UNIX [3] and some under Microsoft Windows) in implementating the components. In particular, the user interface benefited from the use of rapid development user interface tools available under Microsoft Windows, and LEAP benefited from being implemented under UNIX, where we were able to make good use of our Prolog development environment and some existing source code from related projects. Further, the use of the OAA ensures future extensibility via the addition of new agents.

In the following discussions of the three core components, the use and appearance of the user interface component is not covered in detail, but parts of it are mentioned in the core component descriptions, and parts are shown in the accompanying figures.

4.1 ProACT: Defining and Constructing Agents

ProACT guides an agent developer through the various phases of agent creation and maintenance.

An agent developer starts creating a *new* agent by defining, in ProACT, its name, author, title, version number, and icon. To inspect or modify an *existing* agent, the agent can be opened using either of two familiar techiques: the existing agent's specification file can be selected from a file navigation dialog, or its icon can be selected from those in the currently selected agent repository. (Agent repositories are selectable using PROJECT.)

The agent programmer can then use ProACT to enumerate the agent's capabilities in terms of the Interagent Communication Language. The ICL editing window provides an opportunity to ensure conformance to protocol, by performing syntax checks and prompting the developer for missing syntactic elements. [4]

Once the capabilities of the agent have been specified, ProACT encourages the agent programmer to provide documentation for the agent, in a standardized format. Information may be entered using

[3]All product names mentioned in this document are the trademarks of their respective holders.

[4]As of this writing, these syntax checks are under development.

built-in documentation editors, which provide templates for describing the agent itself, and each of the agent's capabilities specifications. After documentation has been edited, ProACT automatically generates HTML representations of the information that can be published on the World Wide Web, and thus can be made readily available to other agent developers collaborating on the project, or those who may add agents to the project at some future time.

The use of HTML as a documentation medium is motivated by the requirement, discussed earlier, to support widely distributed teams of agent developers with up-to-date specifications that can automatically be searched for reusable agents providing some needed service. Publishing documentation in HTML allows developers to employ any of a wide variety of available Web tools. For example, ProACT interfaces with Harvest [1], an Internet tool for indexing and searching Web pages. In the Harvest framework, brokers and gatherers can be set up to collect all published OAA documentation from anywhere in the world, or from selected subgroups of agent development sites — thus providing an efficient query mechanism to search for appropriate agents for reuse.

ProACT supports heterogeneity by generating code templates for agents in several programming languages, currently Prolog, C, C with X Windows, and Visual Basic. Delphi and Lisp will be added soon, as libraries in these languages have recently been added to the OAA. Code template generation is a useful function for the novice programmer, who may not know all the intricacies of building a new agent, as well as being a timesaver for the expert user. Code template generation is also convenient when an existing agent is ported to another programming language.

A ProACT screen is shown in Figure 1. In this figure, code template generation, in C, has just been completed for a new agent.

4.2 LEAP: Adding Speech and Natural Language Understanding to Agents

Agents provide functionality that can be accessed by other agents, by the user through a graphical user interface, or sometimes by the user through a natural language (spoken or written) interface. As mentioned in Section 2.5, speech recognition and natural language processing capabilities are made available to all agents in the OAA by specialized support agents.

To provide a natural language interface to an agent, the agent designer must generate linguistic knowledge bases for the Natural Language and Speech Recognition agents, which enables these agents to handle spoken and written requests that are appropriate for the agent. LEAP is a tool for guiding the user through this process, and is primarily concerned with the requirements expressed in Sections 2.1 and 2.5.

It is important to realize that the roles of the Speech Recognition and Natural Language agents can be played by different agents in different OAA configurations (indeed, it is possible to have several different Speech Recognition and/or Natural Language agents operating within a single configuration). These Speech Recognition and Natural Language agents can be of varying levels of sophistication, and in some configurations, there are advantages to using relatively simple approaches (for example, some configurations have employed Natural Language agents based on Prolog Definite Clause Grammars). However, in most settings, one wants to use the most powerful, flexible approaches available, and thus our efforts have been focused on the use of two very sophisticated systems developed at SRI: the Decipher [3] speech recognition system, and the Gemini [5] natural language understanding system, both of which have been used as agents in a number of OAA-based systems. Consequently, the requirements for LEAP have largely been driven by these

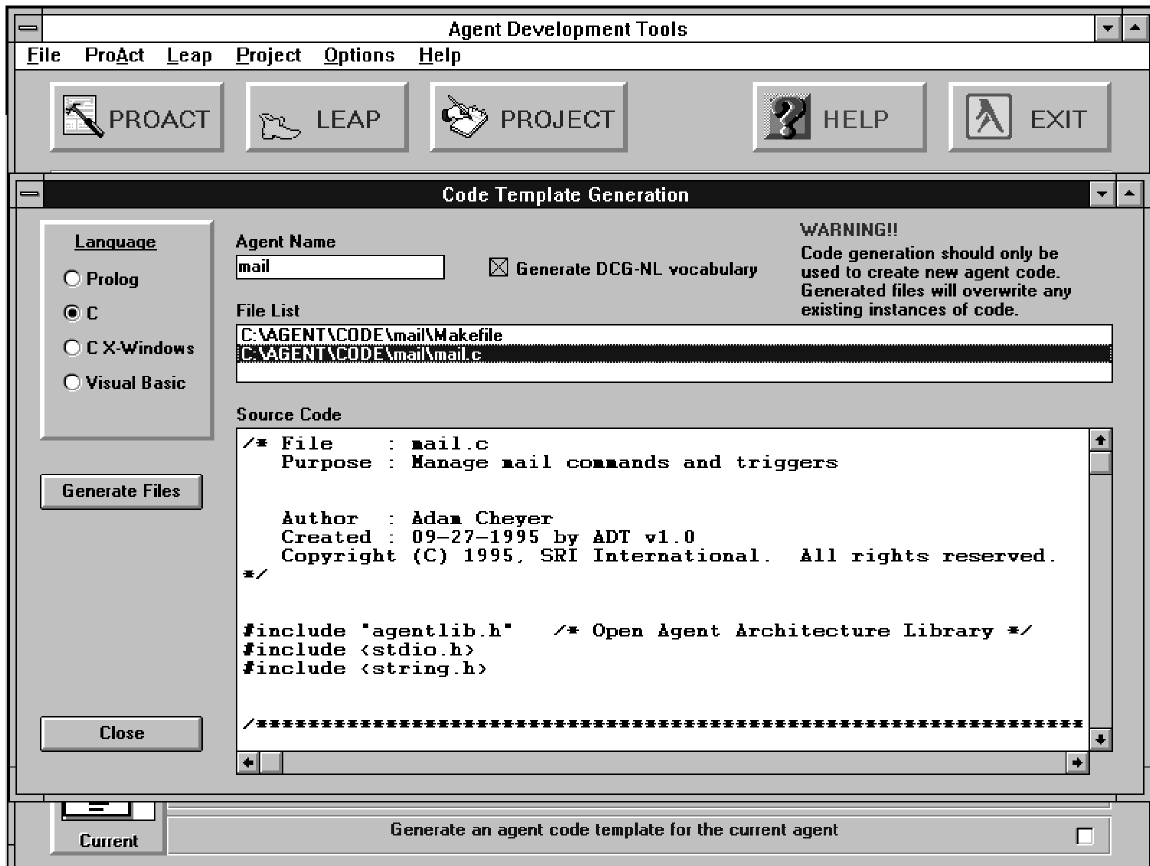

Figure 1: Using ProACT to generate source code for an agent.

two systems.

Although the Speech Recognition and Natural Language agents provide considerable flexibility in specifying knowledge for new domains, they were written by and for computational linguists. Consequently, extending the domain knowledge and linguistic knowledge of these support agents (as is true of most powerful speech recognition and natural language systems) has heretofore been a complex task requiring expertise in computational linguistics. This has been an acceptable requirement in their original context of use. However, their use within the OAA creates a new context, characterized by the following conditions:

- New and widely varying domains are added frequently.
- As agents are introduced and developed in a domain, the knowledge needed by the Speech Recognition and Natural Language agents changes rapidly and may continue to evolve over a long period. This change involves knowledge of linguistic usage as well as knowledge of the solvables (agent capabilities descriptions) currently made available in the domain.
- Agent developers, rather than linguists, will introduce new domain knowledge to the Speech Recognition and Natural Language agents.

LEAP's goal, then, is to assist the nonlinguist in introducing new domain and linguistic knowledge to Speech Recognition and Natural Language agents.

4.2.1 LEAP's Subcomponents and General Approach

LEAP's mission involves acquiring four types of knowledge: domain knowledge, as captured in a class hierarchy; knowledge of the solvables provided by the agents being used in an OAA-based system; some types of linguistic information (morphological, syntactic, and semantic) about the vocabulary that may be used in formulating requests of the agents; and phonetic (pronunciation) information about this vocabulary.

The first three of these knowledge types provide the critical connections that the Natural Language agent will need (at execution time, not at agent development time) to transform an English request into a formal goal that may be handled by an OAA facilitator. This goal, an expression in a first-order logical notation, contains solvables as subgoals. The facilitator, in satisfying the goal, will dispatch each solvable to an agent that can handle it. The fourth type of information will be used (also at execution time) by the Speech Recognition agent in recognizing spoken requests.

LEAP has a subcomponent corresponding to each of these four types of knowledge; these subcomponents are the Class Hierarchy Editor, the ICL-NL Linker [5], the Word Wizard, and the Pronunciation Wizard.

The sequence of events for telling LEAP about a new agent is as follows: First, using the Class Hierarchy Editor, inspect and edit the class hierarchy to ensure that the types of objects the agent deals with are represented in the hierarchy. Then, using the ICL-NL Linker, provide semantic information about the agent's solvables (these have already been entered, using ProACT). Next, using the Word Wizard, enter words that are expected to be contained in users' requests for the

[5] Interagent Communication Language — Natural Language Linker

agent. Finally, for any words for which the Pronunciation Wizard doesn't already have a phonetic description, use the Pronunciation Wizard to select and/or edit one.

In our presentation, here, of the first three subcomponents of LEAP, we are primarily concerned with operations that help to satisfy the knowledge base requirements of the Natural Language agent. This is because its knowledge base is considerably more complex than that required by the Speech Recognition agent. Indeed, most of the information required by the Speech Recognition agent can be viewed as a subset of that needed by the Natural Language agent. One notable exception to this, however, is the information gathered by the fourth subcomponent, the Pronunciation Wizard.

4.2.2 LEAP's Class Hierarchy Editor

Nearly all rules in the knowledge base of the Natural Language agent refer to the classes defined in the class hierarchy. The class hierarchy is a tree that contains what the Natural Language agent recognizes as the primitive conceptual categories to which entities may belong, and expresses the superclass and subclass relationships that hold between them. Higher levels of the hierarchy contain the more domain-independent classes, whereas lower levels tend to be more domain-specific. For example, the class *agent* — a class likely to be near the root of the hierarchy — might have subclasses *human-agent* and *software-agent*, both of which are considered to be domain-independent.

When a new domain (such as the corporate personnel domain) is introduced to the Natural Language agent, it is usually necessary to add new classes reflecting the distinctions made in that domain. For example, the *human-agent* class might have a domain-specific subclass *employee* that is broken into subclasses *manager*, *salesperson*, *researcher*, and *programmer* — reflecting the personnel structure of a particular organization. (These are some of the classes used in our office assistant domain.)

Because the class hierarchy is so central to the expression of the rules used by the Natural Language agent, it must be easy to understand and to edit. Thus, we have provided a Class Hierarchy Editor for browsing and modification of this hierarchy. This editor also allows drag-and-drop techniques to be used in selecting classes during operation of both the ICL-NL linker and the Word Wizard, as described later.

4.2.3 LEAP's ICL-NL Linker

The ICL-NL Linker acquires the knowledge needed by the Natural Language agent so that it can include a new solvable (capability specification) in the formal representations that it generates from English requests.

Two main types of information are requested from the user. First, the user is asked to provide an overall characterization of the solvable as an Entity, Relationship, or Attribute. This means of characterizing solvables was selected because, as a standard part of database methodology, it is likely to be familiar to most developers, and also because the characterization can be used to guide the selection of rules that the Natural Language agent can use in generating appropriate calls to the solvable.

Second, the user is asked to annotate each solvable with information from the class hierarchy; this is done by associating a class with the functor and with each argument of each solvable. This operation is facilitated by the ability to drag and drop class names between the Class Hierarchy

Editor and the ICL-NL Linker. Figure 2 shows the main window of the ICL-NL Linker being used in this way. In this example, the developer, who is characterizing the arguments of the solvables provided by an email agent, has just associated the first argument of the solvable *forward(Msg, Destination)* with the domain-specific class *message*.

In addition, the ICL-NL Linker provides several other utilities that are helpful in introducing new solvables to the Natural Language agent. For example, if a solvable represents a database relation, and thus can be queried for all the tuples in the relation, the ICL-NL Linker can be used to perform these queries and automatically create vocabulary entries corresponding to specific values of the relation's fields.

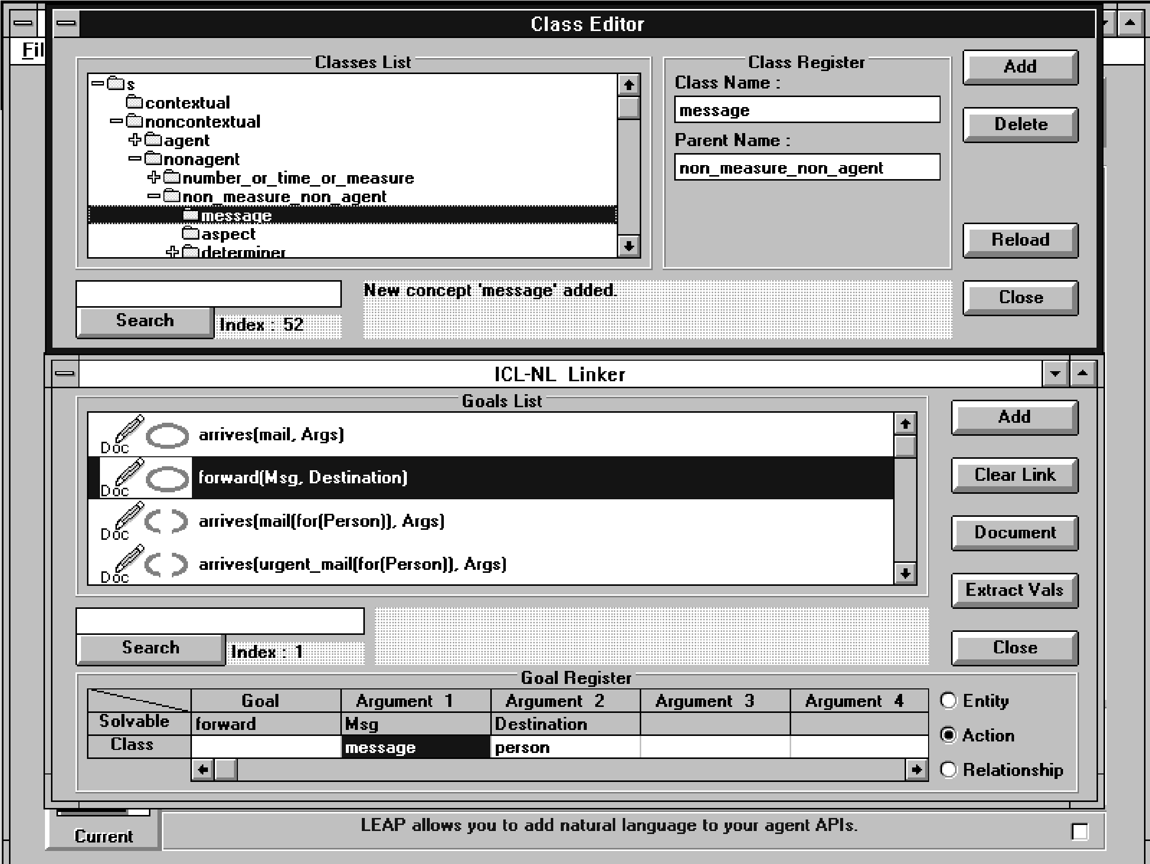

Figure 2: Using LEAP to link ontological classes to an agent specification.

Before moving on to LEAP's most linguistically specialized component, it is worth noting that the functionality of its Class Editor and ICL-NL Linker can be viewed in a nonlinguistic context, that is, as a means of developing domain-specific ontologies, and giving characterizations of agents' capabilities in terms of these ontologies. These characterizations are general enough to be of use to more sophisticated facilitators and information brokers, which are currently under development for use with the OAA.

4.2.4 LEAP's Word Wizard

LEAP's Word Wizard acquires the knowledge needed by the Natural Language agent to understand

sentences containing a particular word or phrase.

The Word Wizard's chief method of acquiring information from the user is exemplar-based; that is, it asks the user questions about the correctness of specific phrases or sentences, and draws the appropriate conclusions based on the responses. This approach is based on previous work done at SRI on the TEAM project [6].

The Wizard operates by obtaining a categorization of a new word, and by gradually refining the categorization through a series of questions. Each refinement of category, in turn, determines the subsequent questions to be asked. Each question asked is used to (1) refine the categorization of the word (roughly, by identifying the important patterns it can be used in), (2) obtain some specific data needed about the word (such as the plural form of a noun), or (3) both of these operations. The questions are simple ones that do not require any expert knowledge about natural language processing.

For example, in constructing an agent that extracts information from a personnel database, the developer might want the agent to be able to answer questions containing the verb 'occupy', as in "Who occupies office number EJ219?". After entering 'occupy' as a new verb, the developer would first be asked to identify one or more acceptable patterns of usage, from a list of available verb usage patterns. Assuming that he selects the pattern "A(n) _______ occupies a(n) _______", he would then be asked to fill in the classes, from the class hierarchy, of the things that can be referred to in the blanked positions. (In this case, he might fill in the classes 'employee' and 'office'.) Following this, LEAP would ask questions about the acceptability of different uses of 'occupy'. For instance, the developer would be asked to say whether the following construction sounds OK: "An office is occupied by an employee". From the answer, LEAP would know whether 'occupy' can be used in the passive form, and could use this information in generating the appropriate lexical entry for 'occupy', to be used by the Natural Language agent.

Once the final categorization for a new word is determined, the Wizard has all the information it needs to update the Natural Language agent's knowledge base. The information gathered by the Wizard for a new word, along with related information entered previously using the Class Hierarchy Editor and the ICL-NL Linker, typically results in a large number of changes (perhaps 10 to 25 detailed updates) to the knowledge base. These updates are transparent to the user, who sees only the command structure provided by the user interface and the commonsense questions that have been presented.

4.2.5 LEAP's Pronunciation Wizard

Much of the knowledge needed by the Speech Recognition agent (such as a word's part of speech) can be derived from the information acquired for the Natural Language agent. One type of linguistic knowledge that is used exclusively by the Speech Recognition agent is a word's phonetic specification, the description of how it is pronounced. Even though the Speech Recognition agent incorporates a large corpus of phonetic information for ordinary words, the vocabulary used by an agent can include domain-specific terminology, names, abbreviations, and acronyms, and thus it is frequently the case that additional phonetic specifications are needed. As a simple example, our office assistant agent system might be expected to answer the spoken question "What is the extension of Adam Cheyer", or to satisfy the request "Send a message to cheyer@ai.sri.com".

Since the Speech Recognition agent needs to have a phonetic specification for each new word introduced to it, and since these specifications employ a fairly specialized notation, LEAP includes

a Pronunciation Wizard to help the agent developer in entering these specifications. The Pronunciation Wizard operates in the background, checking each new word to see if its pronunciation is already known. When a word without a known pronunciation is encountered, it is placed on an action list, until the developer is ready to work on pronunciations. At that time, he can select a word from the action list, and the Pronunciation Wizard uses a sophisticated algorithm to generate a list of plausible phonetic specifications for the word. The developer is asked to select one of these, and also has the option to edit it. To assist in this task, the user can ask to see a phonetic specification for any other word known to the system. For instance, in selecting a phonetic specification for the name "Cheyer", it might be helpful to have a look at the specification for the rhyming word "buyer".

One other way in which the Pronunciation Wizard can be helpful, but which has not yet been implemented, is that a selected phonetic specification could be submitted to the OAA's text-to-speech support agent for audio playback.

4.3 PROJECT: Configuring Communities of Agents

The PROJECT tool, which addresses many of the requirements expressed in Section 2.3, is used to define particular configurations of agents for a given application domain. Using PROJECT, a programmer can graphically construct an agent project by adding members to a conference table, selecting participants from repositories of available agents, and then tailoring agent execution parameters to the task at hand. These execution parameters include such things as what specific machine to execute an agent on, what facilitator the agent should connect to, and what steps to take if the agent unexpectedly crashes. Once a configuration has been specified, the PROJECT tool can generate data files for use by StartIt (Section 3.2).

In Figure 3, PROJECT's main screen is shown, with construction of a project configuration in progress.

5 Conclusions and Future Directions

The main theme of this paper has been that agent-based software paradigms introduce challenging new requirements for development environments, which will need to be addressed before these paradigms are able to realize their full promise. We began by identifying some important general requirements for agent development environments which are relevant to most, if not all, agent-based systems. We have outlined the architecture and functionality of one particular agent-based paradigm, the Open Agent Architecture (OAA), in order to illustrate how these general requirements arise in that context. In our presentation of the Agent Development Toolkit — a prototype development environment for OAA-based systems, which itself consists of a collection of OAA agents — we have shown how many of these requirements have been addressed.

In building the ADT, our initial focus has been on capabilities that provide the greatest gains in productivity, and that are readily accessible to novice agent developers. We recognize that there are many possibilities for additional functionality that can be introduced into the ADT framework, and consequently we have designed the ADT for extensibility.

We have not yet taken full advantage of the fact that the ADT is itself implemented within the OAA.

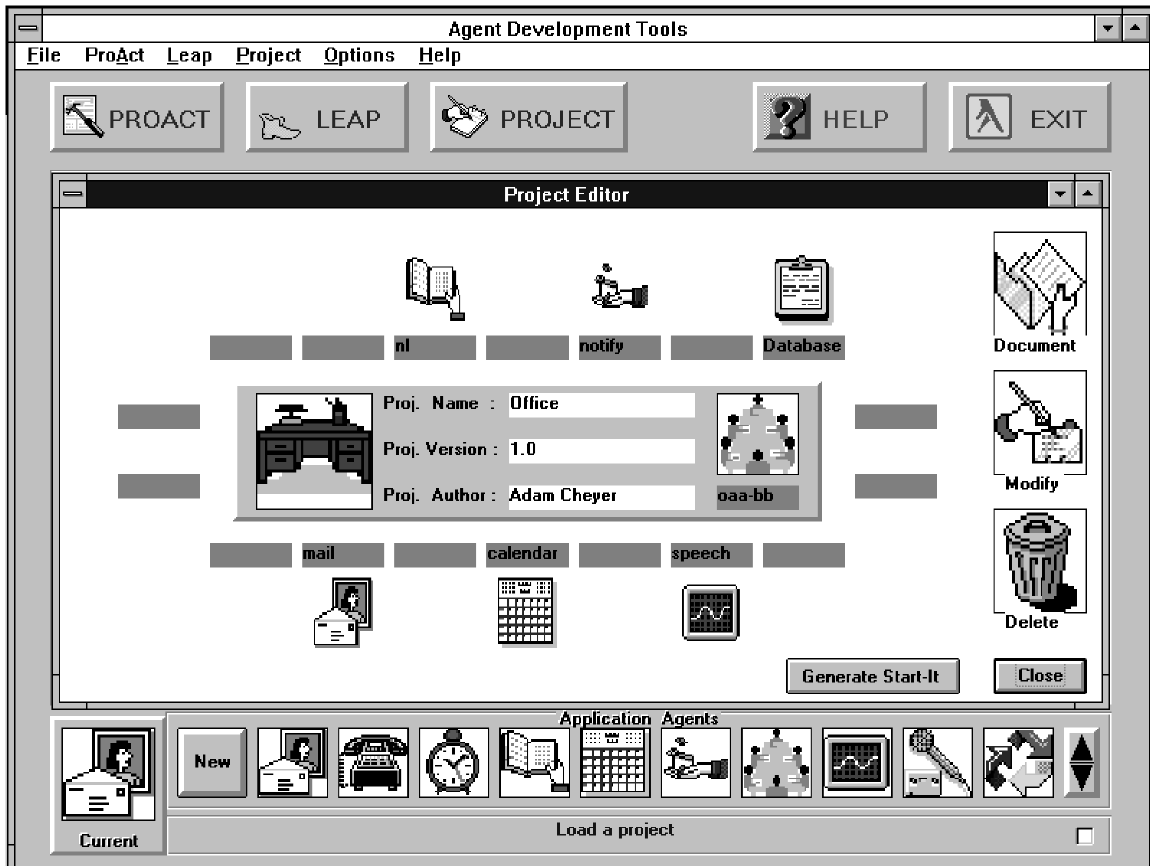

Figure 3: Using PROJECT to define an operable configuration of agents.

Thus the Natural Language and Speech Recognition agents could be used to provide a multimodal interface for the ADT, just as they have for some of our application domains. More importantly, implementation within the OAA means that the results of many development decisions can be tested immediately and demonstrated to the developer within their context of use. For example, when introducing new vocabulary for an agent using LEAP, it should be possible to immediately try out a sentence containing that vocabulary and observe, first, whether the Natural Language agent produces the correct formal representations, and second, whether these representations result in the desired set of agent interactions.

One important area that has not been addressed is debugging tools. Because of the complexity associated with interactions of multiple autonomous agents and the overhead associated with deployment on distributed sites, the ability to simulate a community of agents will have great value. We see this ability as something that will be tightly integrated with the execution environment (which again, will be facilitated by the implementation of the ADT within the OAA). For any selected configuration of agents, it should be possible to initiate a simulated set of interactions without requiring any additional setup effort. The simulation will allow for global and local views of agent activities, with the ability to inspect data, trace, set breakpoints, and step through execution.

Finally, there is important work to be done in reasoning about agent capabilities specifications. So far we have only made use of each agent's specification of the services it *provides*, but it is interesting to consider what could be done if additional information were provided by each agent as to what services it *uses*. We would like to explore to what extent, given these additional specifications, the development environment can automatically determine whether a given configuration of agents can supply a given set of services, and if not, find and select existing reusable agents that supply the missing capabilities.

References

[1] Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. In *Proceedings of the Second International World Wide Web Conference*, pages 763–771, Chicago, Illinois, October 1994.

[2] A. Cheyer and L. Julia. Multimodal maps: An agent-based approach. In *Proceedings of the International Conference on Cooperative Multimodal Communication*, Eindhoven, The Netherlands, May 1995.

[3] Michael Cohen, Ze'ev Rivlin, and Harry Bratt. Speech recognition in the ATIS domain using multiple knowledge sources. In *Proceedings of the ARPA Spoken Language Systems Technology Workshop*, Austin, Texas, January 1995.

[4] P. R. Cohen, A. Cheyer, M. Wang, and S. C. Baeg. An open agent architecture. In O. Etzioni, editor, *Proceedings of the AAAI Spring Symposium Series on Software Agents*, pages 1–8, Stanford, California, March 1994. American Association for Artificial Intelligence.

[5] J. Dowding, J. M. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, and D. Moran. Gemini: A natural language system for spoken-language understanding. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pages 54–61, Columbus, Ohio, June 1993.

[6] Barbara J. Grosz, Douglas E. Appelt, Paul Martin, and Fernando Pereira. TEAM: An experiment in the design of transportable natural-language interfaces. Technical Note 356R, Artificial Intelligence Center, SRI International, Menlo Park, California, 1987.

[7] 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.