Software *Agents*

Michael R. Genesereth

The software world is one of great richness and diversity. Many thousands of software products are available to users today, providing a wide variety of information and services in a wide variety of domains. While most of these programs provide their users with significant value when used in isolation, there is increasing demand for programs that can *interoperate*—that is exchange information and services with other programs and thereby solve problems that cannot be solved alone.

Part of what makes interoperation difficult is heterogeneity. Programs are written by different people, at different times, in different languages. Consequently, they often provide different interfaces. The difficulties created by heterogeneity are exacerbated by dynamics in the software environment. Programs are frequently rewritten, new programs are added, and old programs are removed.

Agent-based software engineering was invented to facilitate the creation of software able to interoperate in such settings. In this approach to software development, application programs are written as *software agents*, i.e., software components that communicate with their peers by exchanging messages in an expressive agent *communication language*.

While agents can be as simple as subroutines, typically they are larger entities with some sort of persistent control (e.g., distinct control threads within a single address space, distinct processes on a single machine, or separate processes on different machines).

The salient feature of the language used by agents is its expressiveness. It allows for the exchange of data and logical information, individual commands and scripts (i.e., programs). Using this language, agents can communicate complex information and goals, directly or indirectly programming one another in useful ways.

Agent-based software engineering is often compared to object-oriented programming. Like an object, an agent provides a message-based interface independent of its internal data structures and algorithms. The primary difference between the two approaches lies in the language of the interface. In general object-oriented programming, the meaning of a message can vary from one object to another. In agent-based software engineering, agents use a common language with an agent-independent semantics.

The concept of agent-based software engineering raises a number of important questions:

1. What is an appropriate agent communication language?
2. How do we build agents capable of communicating in this language?
3. What communication architectures are conducive to cooperation?

(For more information on agent-based software engineering, see [5, 6]. Also see [14] for a description of a variation of agent-based software engineering known as agent-oriented programming.)

Agent Communication Language

Communication language standards facilitate the creation of interoperable software by decoupling implementation from interface. As long as programs abide by the details of the standards, it does not matter how they are implemented. Today, standards exist for a wide variety of domains. For example, email programs from different vendors manage to interoperate through the use of mail standards like SMTP. Disparate graphics programs interoperate using standard formats like GIF and JPEG. Text formatting programs and printers interoperate using languages like PostScript.

Unfortunately, problems arise when it becomes necessary for programs that use one language to interoperate with programs that use a different language. To begin with, there can be inconsistencies in the use of syntax or vocabulary. One program may use a word or expression to mean one thing while another program uses the same word or expression to mean something entirely different. At the same time, there can be incompatibilities. Different programs may use different words or expressions to say the same thing.

Agent-based software engineering attacks these problems by mandating a universal communication language, one in which inconsistencies and arbitrary notational variations are eliminated. There are two popular approaches to the design of such a language—the procedural approach and the declarative approach.

The procedural approach is based on the idea that communication can be best modeled as the exchange of procedural directives. Scripting languages, such as TCL, Apple Events, and Telescript, are based on this approach. They are both simple and powerful. They allow programs to transmit not only individual commands but entire programs, thus implementing delayed or persistent goals of various sorts. They are also (usually) directly and efficiently executable.

Unfortunately, there are disadvantages to purely procedural languages. For one, devising procedures sometimes requires information about the recipient that may not be available to the sender. Second, procedures are unidirectional. Much information that agents must share should be usable in both directions—to compute quantity *a* from quantity *b* at one time and to compute quantity *b* from quantity *a* at another. Most significantly, scripts are difficult to merge. This is no problem as long as all communication is one-on-one. However, things must be run simultaneously and may interfere with one another. Merging procedural information is much more difficult than merging declarative specifications or mixed mode information such as condition-action rules.

In contrast with this procedural approach, the declarative approach to language design is based on the idea that communication can be best modeled as the exchange of declarative statements (definitions, assumption, among others). To be maximally useful, a declarative language must be sufficiently expressive to communicate widely varying sorts of information including procedures. At the same time, the language must be reasonably compact. It must ensure that communication is possible without excessive growth over specialized languages. As an exploration of this approach to communication, researchers in the ARPA Knowledge Sharing Effort [12] have defined the components of an agent communication language (ACL) that satisfies these needs.

ACL can best be thought of as consisting of three parts—its vocabulary, an inner language called KIF (Knowledge Interchange Format), and an outer language called KQML (Knowledge Query and Manipulation Language). An ACL message is a KQML expression in which the "arguments" are terms or sentences in KIF formed from words in the ACL vocabulary.

The vocabulary of ACL is listed in a large and open-ended dictionary or words appropriate to common application areas [9]. Each word in the dictionary has an English description for use by humans in understanding the meaning of the word; and each word has formal annotations (written in KIF) for use by programs. The dictionary is open-ended to allow for the addition of new words within existing areas and in new application areas.

It should be noted that the existence of such a dictionary does not imply there is only one way to describe an application area. Indeed, the dictionary can contain multiple ontologies for any given area. For example, it contains vocabulary for describing 3D geometry in terms of polar coordinates, rectangular coordinates, cylindrical coordinates, and so forth. A program can use whichever ontology is most convenient. The formal definitions of the words associated with any one of these ontologies can then be used by system programs in translating messages using one ontology into messages using other ontologies.

KIF is a prefix version of first order predicate calculus, with various extensions to enhance its expressiveness. It provides for the encoding of simple data, constraints, negations, disjunctions, rules, quantified expressions, metalevel information, and so forth. (See "Knowledge Interchange Format" for a brief summary.)

While it is possible to design an entire communication framework in which all messages take the form of KIF sentences, this would be inefficient. Because of the contextual independence of KIF's semantics, each message would have to include any implicit information about the sender, the receiver, the time of the message history, and so forth. The efficiency of communication can be enhanced by providing a linguistic layer in which context is taken into account. This is the function of KQML. (See "Knowledge Query and Manipulation Language" for a brief summary.)

ACL has been used in several large-scale demonstrations of software interoperation, and the results are promising. Full specifications are available, and parts of the language are making their way through various standards organizations. Several start-up companies are proposing to offer commercial products for processing ACL, and a number of established computer system vendors are looking at ACL as a possible language for communication among heterogeneous systems.

As of this writing, it is not clear which of these two approaches will succeed. The declarative approach seems inevitable in the long run. However, scripting languages are likely to be popular in the short run because of their familiarity. Therefore, the ultimate agent communication language may end up looking more like a scripting language than ACL.

Agents

The criterion for agenthood is a behavioral one. An entity is a software agent if and only if it communicates correctly in an agent communication language such as ACL. This means that the entity must be able to read and write ACL messages, and it means that the entity must abide by the behavioral constraints implicit in the meanings of those messages.

The specific constraints associated with a message derive from the content of that message and general principles of agent behavior. For example, there is veracity (an agent must tell the truth), autonomy (an agent may not constrain another agent to perform a service unless the other agent had advertised its willingness to accept such a request), commitment (if an agent advertises a willingness to perform a service, then it is obliged to perform that service when asked to do so), and so forth.

From a theoretical perspective, it is interesting to note that all of these principles can be derived from the single principle of veracity. In other words, if all agents are constrained to tell the truth, then qualities such as autonomy and commitment will follow. To many people the principle of veracity sounds too strong, but it is not difficult to achieve. An agent can always state its own inputs, outputs, and definitions with confidence, and it can nest conjectures inside of statements about its *beliefs*. Unfortunately, a full account of this issue is beyond the scope of this article, and interesting as it may be theoretically, it has only indirect practical value.

For our purposes here, it is sufficient to say the use of ACL brings with it behavioral constraints. However, this leaves a wide range of possibilities. At one extreme, we can imagine perfect agents that retain all of the information they receive and act in accordance with the logical consequences of this information. At the other extreme, we can imagine simple agents, like calculators, that answer arithmetic problems and ignore everything else. More powerful agents utilize a larger portion of ACL. Less powerful agents use a smaller subset. All are agents, as long as they use the language correctly.

Given a clear statement of the language and the behavioral principles that agents must satisfy, it is straightforward to write programs that behave correctly. But what about all of the programs that have already

Knowledge Interchange Format

KIF [7] is a prefix version of the language of first-order predicate calculus with various extensions to enhance its expressiveness. First and foremost, KIF provides for the expression of simple data. For example, the following sentences encode 3 tuples in a personnel database. The first argument in each is the Social Security number of an individual, the second argument is the department within which the individual works, and the third argument is the individual's salary.

```
(salary 015-46-3946 widgets 72000)
(salary 026-40-9152 grommets 36000)
(salary 415-32-4707 fidgets 42000)
```

More complicated pieces of information can be expressed through the use of complex terms. For example, the following sentence states that one chip is larger than another.

```
(> (* (width chip1) (length chip1)) (* (width chip2) (length chip2)))
```

KIF includes a variety of logical operators to assist in the encoding of logical information (such as negations, disjunctions, rules, quantified formulas). The following expression is an example of a complex sentence in KIF. It asserts that the number obtained by raising any real-number ?x **to an even power** ?n **is positive.**

```
(=> (and (real-number ?x) (even-number ?n)) (> (expt ?x ?n) 0))
```

One of the distinctive features of KIF is its ability to encode knowledge about knowledge, using the ' and , operators and related vocabulary. For example, the following sentence asserts that agent Joe is interested in receiving triples in the salary relation. The use of commas signals that the variables should not be taken literally. Without the commas, this sentence would say that Joe is interested in the sentence (salary ?x ?y ?z) **instead of its instances.**

```
(interested joe '(salary ,?x ,?y ,?z))
```

KIF can also be used to describe procedures, i.e., to write programs or scripts for agents to follow. Given the prefix syntax of KIF, such programs resemble Lisp or Scheme. The following is an example of a three-step procedure written in KIF. The first step ensures that there is a fresh line on the standard output stream; the second step is to print Hello! **to the standard out stream; the final step is to add a carriage return to get to a new fresh line.**

```
(progn (fresh-line t) (print "Hello!") (fresh-line t))
```

The semantics of the KIF core (KIF without rules and definitions) is similar to that of first-order logic. There is an extension to handle nonstandard operators (like ' and , **), and there is a restriction to models that satisfy various axiom schemata (to give meaning to the basic vocabulary in the format). Despite these extensions and restrictions, the core language retains the fundamental characteristics of first-order logic, including compactness and the semidecidability of logical entailment.**

been written, our so-called legacy software? Are there any standard techniques for converting such programs into software agents? In work thus far, a number of different approaches have been taken as shown in Figure 1.

One approach (the leftmost diagram in Figure 1) is to implement a *transducer* that mediates between an existing program and other agents. The transducer accepts messages from other agents, translates them into the program's native communication protocol, and passes those messages to the program. It accepts the program's responses, translates into ACL, and sends the resulting messages on to other agents.

This approach has the advantage that it requires no knowledge of the program other than its communication behavior. It is, therefore, especially useful for situations in which the code for the program is unavailable or too delicate to modify.

This approach also works for other types of resources, such as files and people. It is a simple matter to write a program to read or modify an existing file with a specialized format, thereby providing access to that file via ACL. Similarly, it is possible to provide a graphical user interface (GUI) for a person, allowing one to interact with the system in a specialized graphical language, which is then converted into ACL, and vice versa.

A second approach to dealing with legacy software (the middle diagram in Figure 1) is to implement a *wrapper*, i.e., inject code into a program to allow it to communicate in ACL. The wrapper can directly examine the data structures of the program and can modify those data structures. Furthermore, it may be possible to inject calls out of the program so it can take advantage of externally available information and services.

This approach has the advantage of greater efficiency than the transduction approach, since there is less serial communication. It also works for cases having no interprocess communication ability in the original program. However, it requires the code for the program be available.

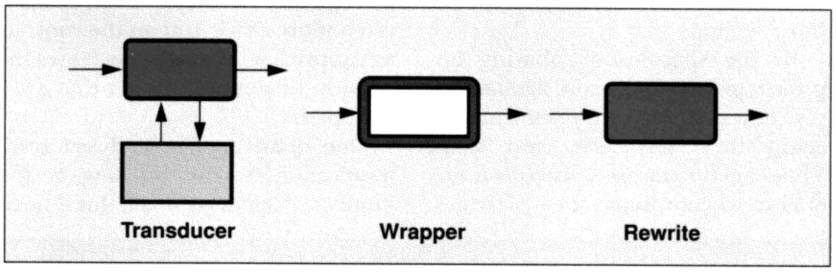


Figure 1. Three approaches to agentification

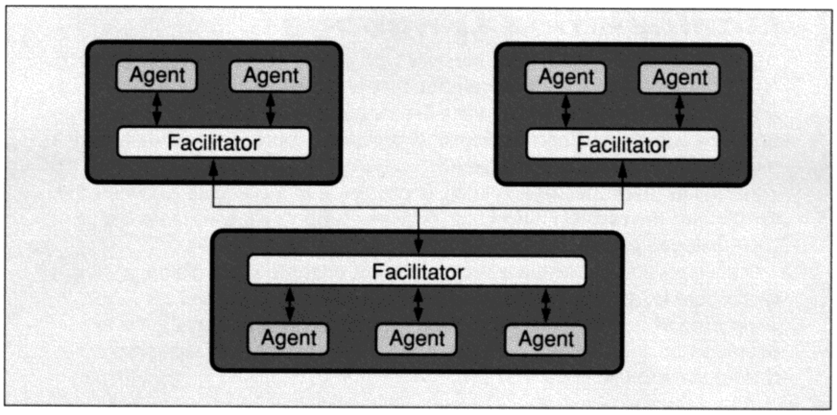


Figure 2. Federated system

The third and most drastic approach to dealing with legacy software (the rightmost diagram in Figure 1) is to rewrite the original program. The advantage of this approach is that it may be possible to enhance its efficiency or capability beyond what would be possible in either the transduction or wrapping approaches.

The best examples of this approach come from the engineering domain. Many automated design programs work to completion before communicating with other programs. For example, the output of a logic synthesis program is passed as input to a printed circuit board layout and routing program, whose output is in turn passed to an assembly planning program. This process is repeated down the line. Recent work in concurrent engineering suggests there is much advantage to be gained by writing programs that communicate partial results in the course of their activity and accept partial results and feedback from other programs. By communicating a partial result and getting early feedback, a program can save work on what may turn out to be an unworkable alternative [11].

Architecture of Multiagent Systems

Once we have a language and the ability to build agents, there remains the question of how these agents should be organized to enhance collaboration. Two very different approaches have been explored: direct communication, in which agents handle their own coordination and assisted coordination, in which agents rely on special system programs to achieve coordination.

The advantage of direct communication is that it does not rely on the existence, capabilities, or biases of any other programs. Two popular architectures for direct communication are the contract-net approach and specification sharing.

In the contract net approach to interoperation [2], agents in need of services distribute requests for proposals to other agents. The recipients of these messages evaluate those requests and submit bids to the originating agents. The originators use these bids to decide which agents to task and then award contracts to those agents.

In the specification sharing approach to interoperation, agents supply other agents with information about their *capabilities* and *needs*. These agents can then use this information to coordinate their activities. The specification sharing approach is often more efficient than the contract net approach because it decreases the amount of communication that must take place.

One disadvantage of direct communication is cost. So long as the number of agents is small, this is not a problem. But, in a setting like the Internet, with millions of programs, the cost of broadcasting bids or specifications and the consequential processing of those messages is prohibitive. In this case, the only alternative is to organize the agents in a way that avoids such broadcasts.

Another disadvantage is implementational complexity. In the direct communication schemes, each agent is responsible for negotiating with other agents and must contain all of the code necessary to support this negotiation. If only these capabilities could be provided by the system, this would lessen the complexity of application programs.

A popular alternative to direct communication that eliminates both of these disadvantages is to organize agents into what is often called a *federated system*. Figure 2 illustrates the structure of such a system in the simple case in which there are just three machines, one with three agents and two with two agents apiece. As suggested by the diagram, agents do not communicate directly with one another. Instead, they communicate only with system programs called *facilitators*, and facilitators communicate with one another. (The concept of a facilitator [6] derives from and generalizes the concept of a *mediator* [16].)

In a federated system, agents use ACL (in practice, a restricted subset of ACL) to document their needs and abilities for their local facilitators. In addition to this metalevel information, they also send application-level information and requests to their facilitators and accept application-level information and requests in return. Facilitators use the documentation provided by these agents to transform these application-level messages and route them to the appropriate places. In effect, the agents form a federation in which they surrender their autonomy to their facilitators and the facilitators take the responsibility for fulfilling their needs.

The concepts of system services in support of software interoperation are not new here. For example, directory assistance programs facilitate software interoperation by providing a way for programs to discover which

Knowledge Query and Manipulation Language

As used in ACL, KQML *messages* are similar to KIF expressions. Each message is a list of components enclosed in matching parentheses. The first word in the list indicates the type of communication. The subsequent entries are KIF expressions appropriate to that communication, in effect the *arguments*.

Intuitively, each message in KQML is one piece of a dialogue between the sender and the receiver, and KQML provides support for a wide variety of such dialogue types.

The expression shown here is the simplest possible KQML dialogue. In this case, there is just one message—a simple notification. The sender is conveying the enclosed sentence to the receiver. In general, there is no expectation on the sender's part about what use the receiver will make of this information.

```
A to B: (tell (> 3 2))
```

The following dialogue is a little more interesting. In this case, the first message is a request for the receiver to execute the operation of printing a string to its standard I/O stream. The second message tells the sender that the request has been satisfied.

```
A to B: (perform (print "Hello!" t))
B to A: (reply done)
```

In the dialogue shown next, the sender is asking the receiver a question in an ask-if message. The receiver then sends the answer to the original sender in a reply message.

```
A to B: (ask-if (> (size chip1) (size chip2)))
B to A: (reply true)
```

In the following case, the sender asks the receiver to send it a notification whenever it receives information about the position of an object. The receiver sends it three such sentences, after which the original sender cancels the service.

```
A to B: (subscribe (position ?x ?r ?c))
B to A: (tell (position chip1 8 10))
B to A: (tell (position chip2 8 46))
B to A: (tell (position chip3 8 64))
A to B: (unsubscribe (position ?x ?r ?c))
```

In addition to simple notifications, commands, questions, and subscriptions, as illustrated here, KQML also contains support for delayed and conditional operations, requests for bids, offers, promises, and so forth.

(For those who have seen a little of KQML and wonder where the packages went, it is worth noting that in addition to the communication layer described here KQML includes yet another linguistic layer to support the transmission of messages among agents operating in different processes. This layer characterizes the additional information that must be conveyed in communication protocols between distributed systems, such as email and TCP connections. The details of this package layer are irrelevant to the discussion in this article. See the KQML document for more information.)

programs can handle which requests and which programs are interested in which pieces of information. Distributed object managers such as CORBA, OLE, DSOM provide location transparency for object-oriented systems, routing messages to objects without requiring senders to know the locations of those objects. Automatic brokers such as the Publish and Subscribe capabilities on the Macintosh, DDE, BMS, and Tooltalk, combine these capabilities—they not only compute the appropriate programs to receive messages but forward those messages, handle any problems that arise, and, where appropriate, return the answers to the original senders.

The primary difference between these approaches to software interoperation and agent-based software engineering lies in the sophistication of the processing done by facilitators. Using ACL, agents can express their needs and capabilities more accurately than in pattern-based metalanguages, and facilitators can use this added information to be more discriminating in routing messages. To deal with notational incompatibilities, facilitators can translate messages from one vocabulary to another using definitions supplied by agents or retrieved from the ACL dictionary. In so doing, they can decompose messages into submessages and send them to different agents. When necessary, they can combine multiple messages. In some cases this assistance can be rendered interpretively with messages going through the facilitators. In other cases, it can be done in one-shot fashion with the facilitators setting up specialized links among individual agents and then stepping out of the picture.

To provide these capabilities, current implementations of facilitators take advantage of automated reasoning technology developed in the artificial intelligence (AI) and database communities. Powerful search control techniques are used to enhance normal message-processing performance, and automatic generation of message-routing programs and pairwise translators is used for cases requiring greater efficiency.

Even with these enhancements, these implementations consume more time in the worst case than simpler processing techniques (like the pattern-matching method used in BMS). This is sometimes acceptable, especially when the alternative is no interoperation at all. However, in time-critical applications such as machine control the extra cost can be prohibitive.

Summary

The agent-based approach to software interoperation described here has been developed into a practical technology, which has been put to use in a variety of applications necessitating interoperation (e.g., concurrent engineering [1], database integration) and is being used at multiple institutions in the construction of software for the national information infrastructure.

In order to concentrate on the central issues in agent-based software engineering, we have ignored many key problems in our presentation, such as synchronization, security, payment for services, crash recovery, and inconsistencies in program specifications. Although partial solutions to these problems exist, further work is needed.

In our treatment so far, we have assumed there is sufficient common interest among the agents that they will frequently volunteer to help one another and receive no direct reward for their labor. As the Internet becomes increasingly commercialized, we envision a world where agents act on behalf of their creators to make a profit. Agents will seek payment for services provided and may negotiate with one another to maximize their expected utility, which might be measured in a form of electronic currency.

These problems mark the intersection of economics and distributed AI (DAI). A number of researchers in DAI are using tools developed in economics and game theory to evaluate multiagent interactions [8, 10, 13, 17]. Depending on the prevailing conditions of the situation, any one of a number of protocols might be applicable. In the simplest case, the agent requesting a service offers a specific reward for the completion of a task. The agent that performs the task receives the payment. In more complex scenarios a task may be completed by a set of agents, who need to negotiate how to divide the reward. Dividing the total amount equally might not be fair if the agents made different contributions. If there are many agents (or sets of agents) that may complete the task, the requester might try to minimize its cost by seeking multiple bids or holding an auction. There are a number of alternatives (e.g., English Ascending Auction, Dutch Descending Auction, Sealed-Bid, Vickery's Second Price) that have different properties and may be applicable or preferred in different situations. The WALRAS system [15] is an example of market mechanics being used to coordinate agents.

A further goal of DAI research is to obviate the need for the truth-telling assumption. If the selected protocols are truth dominant, agents tell the truth out of self-interest, rather than by fiat. This makes the system as a whole more resistant to a scheming agent that might try to exploit other agents by lying. The next step in this research thread is to create protocols that are resistant to the efforts of groups of agents that attempt to manipulate the system for their own benefit.

In this article we have taken a brief look at how agent technology can be used to promote software interoperation. Our long-range vision is one in which any system—software or hardware—can interoperate with any other system without the intervention of human users or their programmers. Although many problems remain to be solved, we believe the introduction of agent technology will be an important step toward achieving this vision. C

References

1. Cutkosky, M. et al. PACT: An experiment in integrated engineering systems. *Computer 26*, 1 (1993), 28–37.
2. Davis, R. and Smith, R.G. Negotiation as a metaphor for distributed problem

CONTINUED ON PAGE 147

CONTINUED FROM PAGE 53

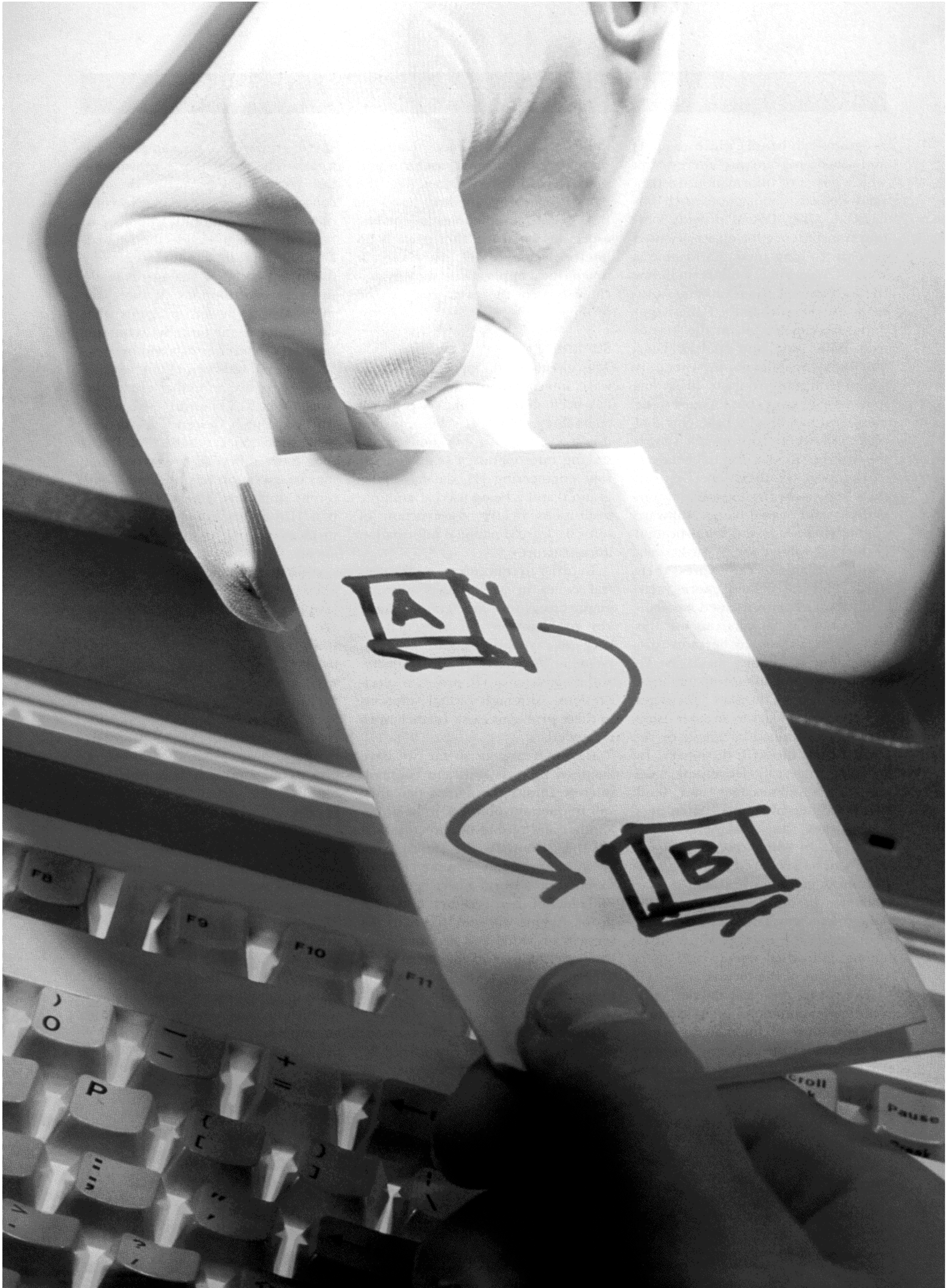
solving. *Artif. Intell. 20,* 1 (1983), 63–109.

3. Ephrati, E. and Rosenschein, J.S. The Clarke Tax as a consensus mechanism among automated agents. In *Proceedings of the Ninth National Conference on Artificial Intelligence* (Anaheim, Calif., 1991). AAAI Press, Menlo Park, Calif., pp. 173–178.
4. Finin, T. and Wiederhold, G. An overview of KQML: A knowledge query and manipulation language. Available through the Stanford University Computer Science Dept., 1991.
5. Genesereth, M.R. A proposal for research on informable agents, Logic-89-9. Stanford University Logic Group, June 1989.
6. Genesereth, M.R. An agent-based approach to software interoperability. In *Proceedings of the DARPA Software Technology Conference,* 1992.
7. Genesereth, M.R., Fikes, R.E. et al. Knowledge Interchange Format Version 3 Reference Manual, Logic-92-1. Stanford University Logic Group, 1992.
8. Gmytrasiewicz, P.J., Durfee, E.H., and Wehe, D.K. A decision-theoretic approach to coordinating multiagent interactions. In *Proceedings of the Twelfth International Joint Conference On Artificial Intelligence* (Sydney, Australia, 1991). International Joint Conferences on Artificial Intelligence, Inc. pp. 62–68.
9. Gruber, T. Ontolingua: A mechanism to support portable ontologies, KSL-91-66, Stanford Knowledge Systems Laboratory, 1991.
10. Kraus, S. Agents contracting tasks in noncollaborative environments. In *Proceedings of the Eleventh National Conference on Artificial Intelligence* (Washington, D.C., 1993). AAAI Press, Menlo Park, Calif., pp. 243–248.
11. Lander, S.E. and Lesser, V.R. Understanding the role of negotiation in distributed search among heterogeneous agents. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence* (Chambery, France, 1993). International Joint Conferences on Artificial Intelligence, Inc. pp. 438–444.
12. Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. Enabling technology for knowledge sharing. *AI Mag. 12,* 3 (1991), 36–56.
13. Rosenschein, J.S. and Genesereth, M.R. Deals among rational agents. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (Los Angeles, Calif., 1985). AAAI Press, Menlo Park, Calif., pp. 91–99.
14. Shoham, Y. Agent-oriented programming. *Artif. Intell. 60,* 1 (1993), 51–92.
15. Wellman, M.P. A market-oriented programming environment and its application to distributed multicommodity flow problems. *J. Artif. Intell. Res. 1* (1993), 1–23.
16. Wiederhold, G. The architecture of future information systems. Stanford University Computer Science Dept., 1989.
17. Zlotkin, G. Mechanisms for automated negotiation among autonomous agents. Ph.D. dissertation. Hebrew University. Feb. 1994.

About the Authors:
MICHAEL R. GENESERETH is an associate professor in the Computer Science Department of Stanford University.

STEVEN P. KETCHPEL is a doctoral student in the Computer Science Department of Stanford University. His research interests include distributed AI and the application of game theory and other economic techniques to DAI problems.

Authors' Present Address: Computer Science Department, Margaret Jacks Hall, Building 460, Stanford University, Stanford, CA 94305.


A
B