Genesereth, M., and Fikes, R. 1992. Knowledge Interchange Format, Version 3.0, Reference Manual, Technical Report, Computer Science Department, Stanford University.

Genesereth, M. R., and Ketchpel, S. P. 1994. Software Agents. *Communications of the ACM* 37(7): 48–53, 147.

Kuokka, D.; McGuire, J. G.; Pelavin, R. N.; Weber, J. C.; Tenenbaum, J. M.; Gruber, T.; and Olsen, G. 1993. SHADE: Technology for Knowledge-Based Collaborative Engineering. In *AI and Collaborative Design: Papers from the 1993 AAAI Workshop,* eds. J. S. Gero and M. L. Maher, 245–262. Menlo Park, Calif.: AAAI Press.

Labrou, Y., and Finin, T. 1994. A Semantics Approach for KQML—A General-Purpose Communication Language for Software Agents. In the Third International Conference on Information and Knowledge Management. New York: Association of Computing Machinery.

Lehrer, N. 1994. The Knowledge Representation Specification Language Manual, Technical Report, ISX Corporation, Thousand Oaks, California.

MacGregor, R., and Bates, R. 1987. The LOOM Knowledge Representation Language, Technical Report ISI/RS-87-188, USC/Information Sciences Institute, Marina del Rey, California.

Mark, W., et. al. 1994. COSMOS: A System for Supporting Design Negotiation. *Journal of Concurrent Engineering: Applications and Research* 2(3).

Neches, R.; Fikes, R.; Finin, T.; Gruber, T.; Patil, R.; Senator, T.; and Swartout, W. 1991. Enabling Technology for Knowledge Sharing. *AI Magazine* 12(3): 36–56.

Newell, A. 1993. Reflections on the Knowledge Level. *Artificial Intelligence* 59:31–38.

Newell, A. 1982. The Knowledge Level. *Artificial Intelligence* 18:87–127.

Pastor, J.; McKay, D.; and Finin, T. 1992. View-Concepts: Knowledge-Based Access to Databases. In Proceedings of the First International Conference on Information and Knowledge Management. New York: Association of Computing Machinery.

Shoham, Y. 1993. Agent-Oriented Programming. *Artificial Intelligence* 60:51–92.

Tenenbaum, M.; Weber, J.; and Gruber, T. 1993. Enterprise Integration: Lessons from SHADE and PACT. In *Enterprise Integration Modeling,* ed. C. Petrie. Cambridge, Mass.: MIT Press.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* San Francisco, Calif.: Morgan Kaufmann.

Chapter 15

An Agent-Based Framework for Interoperability

Michael R. Genesereth

The software world today 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 adequate value when used in isolation, there is increasing demand for programs that can interoperate to exchange information and services with other programs, thereby solving problems that cannot be solved alone.

Unfortunately, getting programs to work together often necessitates extensive work on the part of users and developers. They must learn the characteristics of completed programs and how to negotiate communication formats and protocols for programs under development. What's more, the resulting systems are usually very rigid: components often cannot be modified or replaced without subsequent rounds of negotiation and programming.

Approaches to Software Interoperation

In order to deal with these problems, the systems community has developed various pieces of technology to transfer the burden of interoperation from the creators and users of programs to the programs themselves (figure 1). This technology includes such things as standard communication languages, subroutine libraries to assist programmers in writing interoperable software, and system services to facilitate interoperation at runtime.

Communication language standards facilitate the creation of interoperable software by decoupling implementation from interface. So long as a program abides by the details of the communication standards, it does not matter how it is implemented. Today, standards exist for a wide variety of domains. For ex-

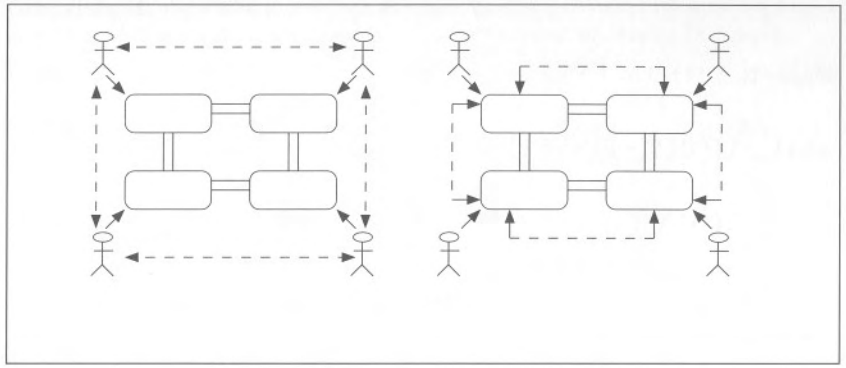

Figure 1. Programmer coordination vs. automatic coordination.

ample, electronic mail programs from different vendors manage to interoperate through the use of mail standards like SMTP. Disparate graphics programs interoperate using standard formats like GIF. Text formatting programs and printers interoperate using languages like Postscript.

There is also a wide range of systems services. Directory assistance programs help by providing a way for programs to discover which programs can handle which requests and which programs are interested in which pieces of information. Automatic brokers (such as the Publish-and-Subscribe capabilities on the Macintosh, DDE, BMS, ToolTalk, OLE, and CORBA) take the directory notion one step further: they not only compute the appropriate programs to receive messages but also forward those messages, handle any problems that arise, and, where appropriate, return the answers to the original senders.

Unfortunately, this kind of technology is too limited to support the ideal of automated coordination suggested in figure 1. To begin with, existing standards are not sufficiently expressive to allow the communication of definitions, theorems, assumptions, and the other types of information that programmers can communicate with each other and that may be needed for one system to communicate effectively with another. Current subroutine libraries provide little support for increased expressiveness. Directory assistance programs and brokers are limited by the lack of expressiveness of the languages used to document resources and their lack of inferential capability.

Furthermore, 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.

As a simple example of these inadequacies, consider two computer-assisted design programs, one running on a Macintosh, the other on a Unix workstation.

The goal is to establish communication between the two programs so that they can exchange information about the positions of chips on a printed circuit board. The problem is that they use different vocabularies for describing these positions. In the Macintosh program, each position is characterized by a single integer *(65536*column+row)*. In the Unix program, each position is characterized by two quantities, a row and a column.

Now, it is a simple matter for two programmers to deal with this discrepancy. They can either rewrite the programs so that they share a common notation for positions, or they can interject some translation code between the programs. The question of interest here, however, is whether this discrepancy can be dealt with automatically. It is really a simple matter of having each program document the coordinate system it uses. For example, the Macintosh program could emit the equation relating its point notation to the alternative row and column notation. After that, it should be able to continue to export point information, leaving it to the system software to translate between this notation and that of the Unix program. Unfortunately, none of the existing approaches to interoperation supports this sort of interaction.

The Knowledge Sharing Effort Approach to Interoperability

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

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, AppleScript, Java, 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 may typically be executed directly and efficiently.

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. Secondly, procedures are unidirectional. Much information that agents must share should be usable in both directions; for example, to compute quantity from quantity at one time and to compute quantity from quantity at another. Most significantly, scripts are difficult to merge. This is no problem so long as all communication is one-on-one. However, things become more difficult when an agent receives multiple scripts from multiple agents that must be run simultaneously and may interfere with each other. Merging procedural information is much more difficult than merging declarative specifications or mixed mode information (like 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, assumptions, and the like). To be maximally useful, a declarative language must be sufficiently expressive to communicate information of widely varying sorts (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 have defined the components of an agent communication language (ACL) that satisfies these needs.

In the approach to interoperation described here, programs (called *agents*) use ACL to supply machine-processable documentation to system programs (called *facilitators*), which then coordinate their activities. Since agents and facilitators assume the burden of interoperation, application programmers are relieved of this responsibility and can construct their programs without having to learn the details of other programs in the runtime environment.

Facilitators and the agents they manage are typically organized 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 each other. Instead, they communicate only with their local facilitators, and facilitators, in turn, communicate with each other. In effect, the agents form a "federation" in which they surrender their autonomy to the facilitators.

As with most other brokering approaches, messages from servers to facilitators are undirected; i.e., they have content but no addresses. It is the responsibility of the facilitators to route such messages to agents able to handle them. There can be an arbitrary number of facilitators, on one or more machines, and the network of facilitators can be connected arbitrarily.

The federation architecture provides assisted coordination of other agents based on a *specification-sharing* approach to interoperation. Agents can dynamically connect or disconnect from a facilitator. Upon connecting to a facilitator, an agent supplies a specification of its *capabilities* and *needs* in ACL. In addition to this meta-level information, agents also send application-level information and requests to their facilitators and accept application-level information and requests in return. Facilitators used the documentation provided by these agents to transform these application-level messages and route them to the appropriate agents. The agents agree to service the requests sent by the facilitators, and in return, the facilitators manage the requests posted by the agents.

A major difference between the knowledge-sharing approach to software interoperation and previous approaches lies in the sophistication of the processing done by these facilitators. In some cases, facilitators may have to translate the messages from the sender's form into a form acceptable to the recipient. In some cases, they may have to decompose the message into several messages, sent to different agents. In some cases, they may combine multiple messages. In some

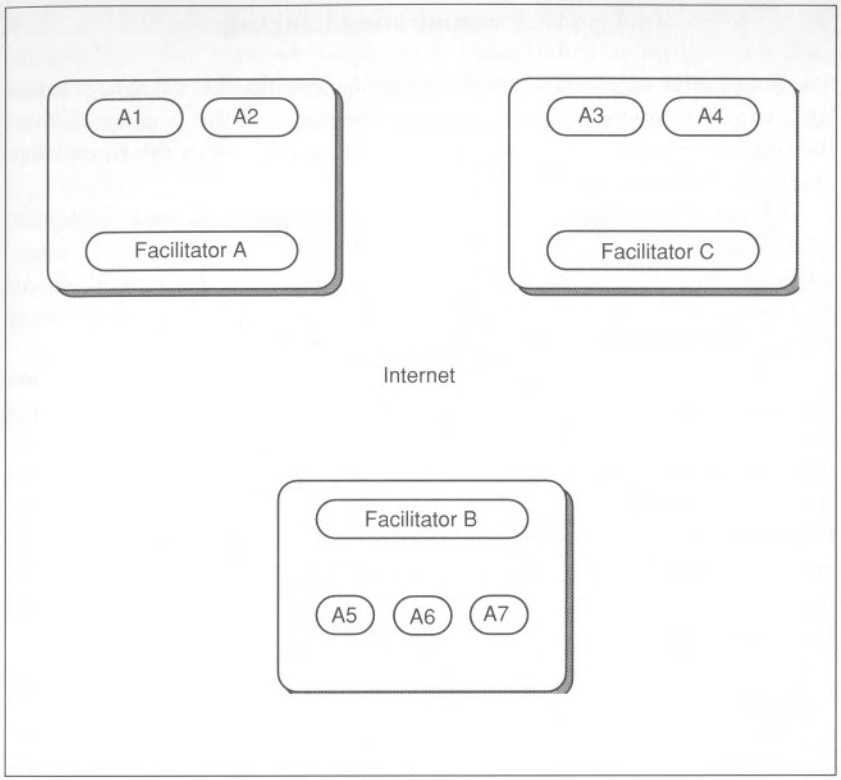

Figure 2. Federated system.

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 between individual agents and then stepping out of the picture.

The knowledge-sharing approach to software interoperation has been developed into a practical technology that has been put to use in a variety of applications necessitating interoperation.

In the next section, I introduce the ACL language and describe various approaches to building agents capable of communicating in ACL. In the Software Agents section, I show how ACL can be used by agents in communicating their specifications to other agents. I then describe the implementation of facilitators (Facilitators section) and give a detailed example of the federation architecture (Example section). In the Applications section, I describe three applications of the technology, and conclude in the final section with a discussion of remaining issues and a summary of key points.

Agent Communication Language

The basis for the approach to interoperation described here is a language called ACL (Agent Communication Language). The design of this language follows the recommendations of the various committees involved in the Knowledge Sharing Effort sponsored by ARPA.

ACL can best be thought of as consisting of three parts: an "inner" language called KIF (Knowledge Interchange Format), its vocabulary, 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 example below illustrates the use of ACL in an exchange of information between an agent named Joe and an agent named Bill. Joe starts off by sending a message to Bill asking him to store the fact that either p or q is true of every object in the world. He then sends a second message asking Bill to store the fact that p is not true of the object named a. In the third message Joe asks Bill for some object ?x for which (q ?x) is true. Since either p or q is true of every object and p is not true of a, then q must be true of a. In this example, it is assumed that Bill is able to draw such conclusions, with the result that he responds to Joe's request with the answer a.

```
Joe to Bill:
(request :sender joe :receiver bill
:content (stash '(or (p ?x) (q ?x))))
Joe to Bill:

(request :sender joe :receiver bill
:content (stash '(not (p a))))

Joe to Bill:
(request :sender joe :receiver bill :reply-with msg1
:content (ask-one '?x '(q ?x))))

Bill to Joe:
(respond :sender bill :receiver joe :in-reply-to msg1
:content a)
```

Knowledge Interchange Format (KIF)

As mentioned above, "arguments" in an ACL message are expressions in a formal language called KIF (Knowledge Interchange Format). KIF is a prefix version of the language of first order predicate calculus, with various extensions to enhance its expressiveness.

The basic vocabulary of the language includes variables (the first column in the examples below), operators (the second column), object constants (the third column), function constants (the fourth column), and relation constants (the fifth column). Individual variables are distinguished by the presence of ? as initial character, and sequence variables are distinguished by the presence of an initial @. There is a fixed set of operators. All others words are constants.

?x	not	chip1	sin	prime
?res	and	theta	cos	left
@lor	123	+	>	

From words, we can build KIF terms to refer to objects in the universe of discourse. In addition to variables and constants, the language includes operators to build complex terms, conditional terms, set descriptors, and quoted expressions (which refer to expressions).

```
(size chip1)
(+ (sin theta) (cos theta))

(if (> theta 0) theta (- theta))
(setofall ?x (above ?x chip1))
(quote (above chip1 chip2))
```

From terms, we can build sentences. These include simple sentences, as well as logical sentences involving Boolean operators like "and," "or" and "=>" (the implication operator).

```
(prime 565762761)
(> (sin theta) (cos phi))

(not (> sin theta) 0))
(or (< 0 (log h)) (< (log h) 1))
(=> (> ?x 0) (positive ?x))
```

Finally, there are rules and definitions. Rules differ from implications (sentences involving the =>) in that they can be used in only one direction. The presence of the operator "consis" in the first part of a rule signals that the rule can applied so long as the enclosed sentence is consistent with the database. Without the "consis," the sentence must be contained in the database in order for the rule to be applied. New object constants, function constants, and relation constants can be defined using the appropriate definitional operators. Such definitions are not essential in order for new constants to be used; however, they provide a convenient grouping for the sentences defining those concepts and allow one to distinguish facts that are true by definition from assertions about the world.

```
(=>> (consis (not (conn ?x ?y))) (not (conn ?x ?y))) (defrelation leq (?x ?y) := (not
(> ?x ?y)))
```

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 "quote," "if," and "setof"), 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 funda-

mental characteristics of first-order logic, including compactness and the semidecidability of logical entailment.

The semantics of rules and definitions is not first order and leads to potential noncompactness and potential nonsemidecidability. However, the extension is upwardly compatible, so that these properties are guaranteed for any databases without rules or definitions.

Vocabulary

In order for programs to communicate about an application area, they must use words that refer to the objects, functions, and relations appropriate to that application area, and they must use these words consistently. One way to promote this consistency is to create an open-ended dictionary of words appropriate to common application areas. Each word in the dictionary would have an English description for use by humans in understanding the meaning of the word, and each word would have KIF annotations for use by facilitators in mediating disagreements of terminology.

Note that, in proposing such a dictionary I am not proposing that there be one standard way of encoding information in each application area. Indeed, the dictionary would probably contain multiple *ontologies* for many areas. For example, we would expect it to contain vocabulary for describing three dimensional geometry in terms of polar coordinates, rectangular coordinates, cylindrical coordinates, and so on. Each program could use whichever ontology is most convenient. The formal definitions of the words associated with any one of these ontologies could then be used by the facilitator in translating messages using one ontology into messages using the other ontology. These issues are discussed in more detail in the section on translation.

Knowledge Query and Manipulation Language (KQML)

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, 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.

As used in ACL, each KQML *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 below is the simplest possible KQML dialog. 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 below, 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, KQML also contains support for delayed and conditional operations, requests for bids, offers, promises, and so forth.

Software Agents

In the approach to interoperation described here, application programmers write their programs as software agents. Like other agents, a software agent is obliged to communicate in ACL, but it does so in a particularly stylized way:

1. On start up, it initiates an ACL connection to the local facilitator.
2. It supplies the facilitator with a description of its capabilities.
3. It then enters normal operation: it sends the facilitator requests when it is incapable of fulfilling its own needs, and it acts to the best of its abilities to satisfy the facilitator's requests.

A software agent is a special kind of agent in that it surrenders its autonomy to the facilitator. A general agent is not compelled to satisfy the requests of other

agents. It can accept them or decline them, or it can negotiate for payment. A software agent does not have this freedom. After registering with its local facilitator and supplying its specification, the software agent is obliged to satisfy the facilitator's requests whenever it can. Of course, this is a good deal in many cases, since the agent gets the facilitator's services in return.

The following subsection describes how software agents specify their capabilities and needs; and the Agent Implementation Strategies subsection discusses various strategies for building them, from writing new programs to dealing with legacy software.

Specifying Agent Capabilities and Needs

In order to provide services to other agents, an agent must communicate its capabilities to the facilitator in ACL. An agent specifies its capabilities by transmitting "handles" facts to its facilitator. For example, an agent capable of answering questions about the dealer of a vendor may transmit the following specification to its facilitator:

```
(handles business-agent '(ask-one ,?variables (dealer ,?dealer ,?vendor)))
(handles business-agent '(ask-all ,?variables (dealer ,?dealer ,?vendor)))
```

These facts state that agent business-agent is capable of answering queries about a single dealer for a vendor, or all the dealers for a vendor. The actual capability is a quoted KQML expression, such as '(ask-one ,?variables (dealer ,?dealer ,?vendor)) in the first example. This specification is similar to the object interface specifications in CORBA's IDL.

If some other agent A_1 wants to know the dealers of NEC, it may communicate the following request to the facilitator:

```
(ask-all ?x (dealer ?x nec))
```

The facilitator examines its knowledge base and determines that the business-agent can handle the request. The facilitator sends the request to the business-agent, gets the answer, and passes it to A_1. Agent A_1 is completely unaware of the sequence of steps performed in servicing its request.

Capabilities can be more complicated, as in the following conditional specification:

```
(<= (handles business-agent '(ask-all ,?variables (dealer ,?dealer ,?vendor))) (=
?vendor 'ibm))
```

This states that the business-agent can answer only queries about the dealers of ibm. The specifications can have arbitrarily complicated preconditions.

An agent specifies its needs by transmitting "interested" facts to its facilitator. For example, the following states that the agent cs-manager is interested in all facts regarding the release of PC-compatible computers.

```
(interested cs-manager '(tell (released ,?manufacturer PC ,?model)))
```

Similar to "handles" statements, "interested" statements can be conditional:

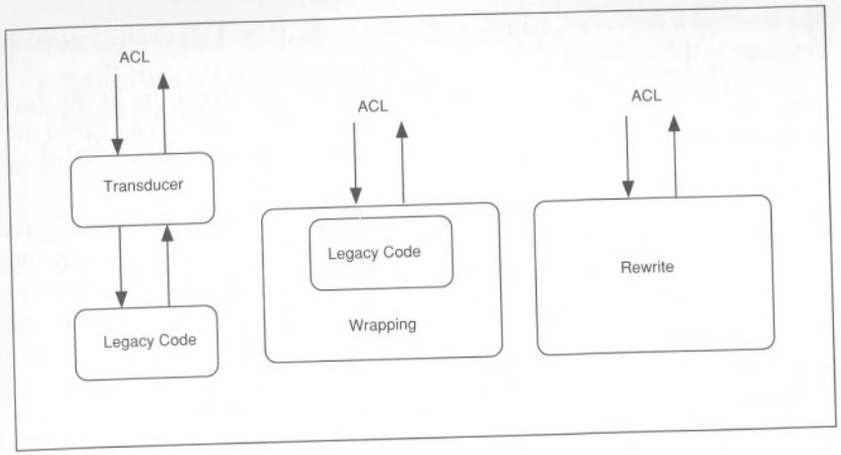

Figure 3. Three approaches to agent implementation.

```
(<= (interested cs-manager '(tell (released ,?manufacturer PC ,?model))) (member
?manufacturer '(ibm toshiba nec micro-international)))
```

This states that the cs-manager agent is interested only in the release of PC-compatible computers from IBM, Toshiba, NEC, and Micro-International. If another agent transmits the following fact to the facilitator:

```
(tell (released micro-international PC 6500D))
```

then the facilitator will examine its knowledge base and find that the agent cs-manager is interested in expressions of this form, and it will send the same KQML expression to the cs-manager. We will discuss the specification of agent needs and capabilities from the facilitator's point of view in the Content-Based Routing subsection that follows.

Agent Implementation Strategies

The criterion for agenthood is a behavioral one. A process is a software agent if and only if it acts like one. Any implementation that achieves this behavior is acceptable. Nevertheless, it is natural to ask whether there are any standard strategies for converting legacy programs into software agents. In my work thus far, I have taken different approaches in different cases (figure 3).

One approach is to implement a transducer that mediates between an existing program and the local facilitator. The transducer initiates communication with the local facilitator. It supplies the program's specification to the facilitator. It accepts requests from the facilitator, translates them into the program's native communication language, and passes those messages to the program. It accepts

the program's responses, translates them into ACL, and sends the resulting messages to the facilitator.

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 tamper with.

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 and thereby provide access to that file via ACL. Similarly, it is possible to provide a graphical user interface for a person that allows that person 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 is to implement a wrapper that, in essence, injects 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 into the code so that the program 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 where there is no interprocess communication ability in the original program.

The third and most drastic approach to dealing with legacy software 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 traditional engineering programs are designed to work to completion before communicating with other programs. Recent work in concurrent engineering suggests that there is much advantage to be gained by writing programs that communicate partial results in the course of their activity and that 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. The expressiveness of ACL allows programs to express partial information. However, many existing programs are unable to take advantage of this expressiveness and the concurrent engineering strategy. In these cases, rewriting the programs is the only alternative.

Facilitators

Facilitators are the system-provided agents that coordinate the activities of the other agents in the federation architecture. Each facilitator keeps the other facilitators in the network informed of which agents are connected to it and what facts have been communicated by them.

Facilitators provide a collection of services, including:

- *White Pages:* finding the identity of agents by name, for example, "What agents are connected?" or "Is agent x connected?"
- *Yellow Pages:* finding the identity of agents capable of performing a task. For example, "What agents are capable of answering the query x?"
- *Direct Communication:* sending a message to a specific agent.
- *Content-based Routing:* the facilitator is given the responsibility of handling a request. It makes use of the specifications and other information provided by the agents to do this, thereby giving the illusion that it is the sole provider of all services.
- *Translation:* agents may use different vocabulary. In order to interoperate, the facilitator may have to translate the vocabulary of one agent into the vocabulary of another.
- *Problem Decomposition:* handling a complex request may require breaking it into sub-problems, getting the answers to the sub-problems, and then combining these answers to obtain the answer to the original request. As in content-based routing, the facilitator makes use of the specifications and application-specific information provided by the agents to accomplish this.
- *Monitoring:* when an agent informs the facilitator of a need, the facilitator monitors its knowledge to determine if the need can be satisfied. For example, an agent may specify the need "I am interested in facts about the position of chips in design x."

The responsibility of the facilitator on each machine is to assist the agents running on that machine to collaborate with each other and, indirectly, with the agents running on other machines. In this section, I describe the communication services a facilitator needs to provide, and I discuss ways in which the facilitator can help agents set up direct communication amongst themselves in order to eliminate the overhead inherent in communicating through the facilitator. I give an overview of the reasoning program in the next subsection, followed by illustrations of how it is used in content-based routing, translation, synthesis, buffering, matchmaking, and connectivity.

Overview of the Implementation

The top-level program of the facilitator is a loop that accepts messages from the agents and facilitators to which it is connected. On receipt of a message, it passes the message to its message handler and goes back to listening for additional messages.

In handling a message, the message handler uses an automated reasoning program on its knowledge base of specification information. Our reasoning program is a variation on the method used in Prolog. There are two primary dif-

ferences. First of all, it handles KIF syntax rather than Prolog syntax. Secondly, unlike Prolog, it is sound and complete for full first-order predicate calculus: it is based on the model elimination rule of inference, the unification algorithm does an occurcheck, the restriction to Horn clauses is removed, and the search is done in iterative deepening fashion.

A full description of the program is beyond the scope of this paper. However, a simple example should convey sufficient detail for readers to follow the examples in the following subsections.

Consider a database with the sentences shown below. The predicate p holds of three pairs of objects—a and a, a and b, and b and c. The predicate q is also true of three pairs of objects—a and b, b and c, and c and d. The predicate r is defined to be true of two objects if there is an intermediate object such that p is true of the former object and this intermediate object and q is true of the intermediate object and the latter object.

```
(p a a)
(p a b)
(p b c)

(q a b)
(q b c)
(q c d)

(<= (r ?x ?z)
(p ?x ?y)
(q ?y ?z))
```

Suppose now, we wanted to know whether r was true of a and c. The trace shown below shows how the reasoning program derives this result.

```
Call: (r a c)?
Call: (and (p a ?y) (q ?y c))
Call: (p a ?y)
Exit: (p a a)
Call: (q a c)
Fail: (q a c)
Call: (p a ?y)
Exit: (p a b)
Call: (q b c)
Exit: (q b c)
Call: (and (p a b) (q b c))
Exit:(r a c)
```

The desired conclusion (r a c) unifies with the conclusion of the last sentence in the knowledge base with the variable ?x bound to a and the variable ?z bound to c. The program thus reduces the original question to the subquestion on the second line—in effect the question of whether there is a binding for the variable ?y for which the conjunction is true. In order for this to be true, there must be a binding for ?y for which (p a ?y) is true. The program first finds (p a a) and binds ?y to a. It then tries to prove (q a c). Unfortunately, this fails. So, the program backs up and tries to find another way to satisfy (p a ?y). In so doing, it discovers the fact (p a b) and binds ?y to b. Again it tries to prove (q b c) and in this case succeeds. Since both conjuncts are proved, the conjunction is proved; and, since the conjunction is proved, the original sentence is proved.

This program is both sound and complete. In other words, if the program manages to prove a result, then that result must logically follow from the sentences in the database; and if a conclusion logically follows from the database, the method will prove it.

Unfortunately, as with all sound and complete reasoning methods for the full first-order predicate calculus, the method does not necessarily terminate. If a conclusion does not follow from the database, the method may spend forever trying to prove it. While this situation does not often arise, it is a real danger for a piece of system code.

In order to deal with this difficulty, the facilitator uses a preprocessor to screen sentences before they are added to the facilitator's database. The facilitator adds a sentence if and only if it can prove that doing so will not cause an infinite loop.

Note that the problem of making this determination is itself undecidable; so it is not possible to know in all cases whether a sentence will cause an infinite loop. Our facilitator circumvents this difficulty by taking a conservative approach to proving the "safeness" of a set of sentences: it uses a variety of tests to determine whether an inference will terminate. If a database passes the tests, termination is assured. If not, the database may or may not be safe. Fortunately, the tests are cheap, and they cover a very large fraction of the kinds of sentences that programmers writing communication specifications will need.

One example of a test is the requirement that a database be function-free. If there are no function constants, then the database reduces, in effect, to propositional calculus, for which there is a known decision procedure. The sentence shown below would not pass this test, because of the embedded function f, and it is easy to see how the method described above would enter an infinite loop in using this sentence to prove a simple conclusion like (p a).

```
(<= (p ?x) (p (f ?x)))
```

Even if a database contains functions, it is possible to show termination provided that the database contains no recursion. For example, the first sentence below would pass this test, whereas the second would fail, since p is dependent on p

```
+ (<= (r ?x ?z) (p ?x ?y) (q ?y ?z))
x (<= (p ?x ?z) (p ?x ?y) (p ?y ?z))
```

Even if the database has recursion and functions, it is possible to show termination provided that every recursive call diminishes the complexity of its

arguments. For example, the following database passes this test, because the recursive calls to r all concern a subpart of the original expression.

```
(r 0)
(<= (r (listof ?x ?y)) (r ?x) (r ?y))
```

Of course, other tests are possible. The problem of provable termination has been studied extensively in the database and automated reasoning communities. With additional work, more tests can be employed, thus enlarging the set of sentences the facilitator can handle.

Content-Based Routing

From an application programmer's point of view, communication in a federation architecture is undirected: application programs are free to send messages without specifying destinations for those messages. It is the job of the facilitator to determine appropriate recipients for undirected messages and to forward the messages accordingly. In so doing, the facilitator functions as a broker for the services provided by the servers in its community.

In order to see how this is done, consider how the facilitator handles the message shown below. It is being told via one particular encoding that the object named chip1 is indeed a computer chip.

```
(tell '(member chip1 chips))
```

The facilitator is connected to three agents, named layout, domain-editor, and board-editor. These agents have given the facilitator the specification information shown below.

```
(interested layout `(tell (position ,?x ,?r ,?c)))

(<= (interested domain-editor `(tell (member ,?x ,?y)))
(symbol ?x)
(symbol ?y))

(<= (interested board-editor `(tell (= (,?f ,?x) ,?y)))
(member ?f (setof 'row 'col))
(symbol ?x)
(natural-number ?y))
```

In order to determine which agents are interested in this message, the facilitator forms the query (interested ?a '(tell (member chip1 chips))) and uses its reasoning program to find a binding for variable ?a. In this case, there is just one, the domain-editor. Consequently, the facilitator sends the message to this agent.

```
(tell '(member chip1 chips))
```

Note that in making the determination that the domain-editor agent is interested, the facilitator must not only match the pattern in the first line of its specification but also verify properties of the bindings of the variables, in particular that they are both symbols.

Translation

Agents in a system may interoperate even when they are not created using the same programming language or development framework. Like programming "objects," agents define message-based interfaces that are independent of their respective internal data structures and algorithms. The translation capability of facilitators extends this significantly by making interoperation independent of the agent interface (the KQML expressions the agent can handle). An agent can be replaced with a more capable implementation with a different interface. By providing translation rules to map the old interface to the new, the agent can provide its old functionality in addition to the new and improved one.

There are two parts to the translation process: *vocabulary* and *logical*. The need for *vocabulary translation* arises because of differences between the abstractions inherent in the implementations of different agents. For example, one agent may work with rectangular coordinates, while another works with polar coordinates.

The need for *logical translation* arises because of limits imposed by agents on the logical structure of messages in which they are interested. Some agents are capable of accepting any message in ACL. Other agents are more selective. For example, a Prolog agent might restrict its interest to Horn Clauses; a relational database might restrict its interests to ground atomic formulas.

As an example of translation, consider a situation in which the facilitator receives the message shown below. As before, it is being told via one particular encoding that the position of a particular chip on a printed circuit board is located in the tenth row and sixteenth column.

```
(tell '(= (pos chip1) (point 10 16)))
```

The facilitator's agent catalog mentions that an agent named layout is interested in receiving messages of the form (position **), where ** and ** are natural numbers.

```
(<= (interested kb (tell '(position ?x ?m ?n))
(natural-number ?m)
(natural-number ?n))
```

Since the incoming sentence does not have the form specified in this interest, content-based routing alone would not cause any message to be sent to layout. However, let us suppose that the facilitator also has information relating pos and position, as in the following sentence:

```
(<=> (= (pos ?x) (point ?row ?col))
(position ?x ?row ?col))
```

Using this sentence together with the sentence (= (pos chip1) (point 10 16)), the facilitator is able to deduce the sentence (position chip1 10 16). In other words, it can translate from one form to the other. It then checks whether any agent is interested in this information, finds layout, and sends the message shown below.

```
(tell '(position chip1 10 16))
```

An important issue in translation is knowing when to make the effort. When the facilitator receives a message, how does it know that translation will lead to a new message that is of interest to one of its agents without doing the translation? Randomly generating conclusions from the information in the message is impractically expensive. A better alternative is derivation of conclusions after filtering with the results of some connectivity analysis. Given a set of interests and a data base of axioms relating differing vocabularies, it is possible to distinguish lines of reasoning that lead to potentially interesting conclusions from those lines that cannot possibly lead to interesting conclusions. This analysis is efficient and needs to be done just once, before the facilitator receives any messages.

Synthesis

The example of translation in the preceding subsection is particularly simple. One incoming message leads to one outgoing message. In some cases, an incoming message can be handled only by sending multiple messages to multiple agents. In order to handle such messages, the facilitator must be able to synthesize a multi-step communication plan to handle the incoming message.

As an example of this type of message handling, consider the message shown below. As in the last example, the facilitator is being told the position of a particular chip.

```
(tell '(= (pos chip1) (point 10 16)))
```

One difference in this example is that the facilitator's agent catalog contains the information shown below, documenting an agent interested in row information and col information but not pos information.

```
(<= (interested board-editor (tell `(= (,?f ,?x) ,?y)))
(member ?f (setof 'row 'col))
(symbol ?x)
(natural-number ?y))
```

As before, let us assume that the facilitator's library contains a sentence relating the two vocabularies.

```
(<=> (= (pos ?x) (point ?row ?col))
(and (= (row ?x) ?row) (= (col ?x) ?col)))
```

In this case, there are two conclusions from the original sentence. The facilitator discovers these two conclusions and sends them on to the board-editor agent.

```
(tell '(= (row chip1) 100))
(tell '(= (col chip1) 160))
```

Note that if the incoming message had been an ask-if message, the facilitator would have been able to reduce this to two questions: one about the row of the chip and another about the column. In this case, it would first send the row question to board-editor; then, on getting an answer, it would send in the col question; and, on getting that answer, would be able to answer the original question.

Buffering

Another important issue in translation is buffering. In some cases, it may not be possible to transform a message into a form that is acceptable to any agent, yet it is possible to merge the information from two or more messages to form an acceptable result.

As an example of how the facilitator handles these cases, consider the incoming message shown below. In this case, the incoming information involves the row of a chip. No col information is provided.

```
(tell '(= (row chip1) 10))
```

The facilitator's agent catalog contains information about an agent layout that is interested in pos information.

```
(interested frame-editor `(tell (= (pos ,?x) ,?y))
```

The facilitator's library contains information relating row and col to pos, just as before.

```
(<=> (= (pos ?x) (point ?row ?col))
(and (= (row ?x) ?row) (= (col ?x) ?col))
```

Unfortunately, in this case, it is unable to complete its translation since there is no col information. But all is not lost. From looking at its agent catalog and library, the facilitator knows that sentences involving row can lead to sentences involving pos; and it knows that there are agents interested in pos sentences. Also from an examination of its agent catalog, it can conclude that there are no agents that handle requests to store row sentences. Given these two conditions, the facilitator decides to save the incoming fact in its information buffer for potential further use.

```
(= (row chip1) 10)
```

Now suppose that, at some point after this, the facilitator receives the missing col information in a message like the following:

```
(tell '(= (col chip1) 16))
```

Putting this information together with the preceding fact, it can conclude a pos sentence; and it sends the derived sentence onto the layout agent.

```
(tell '(= (pos chip1) (point 10 16)))
```

Note the importance here of knowing when to buffer information and when to discard. If the facilitator were to save every piece of information it receives, it would quickly run out of space. By saving only those pieces of information that are of potential use and that are not being stored elsewhere, the amount of information that must be saved remains manageable.

Matchmaking

As described above, content-based routing, translation, and buffering are all performed by the facilitator. This is necessary for maximum flexibility, thus allowing for agent substitutions, changing interest lists, system reconfiguration, and so forth. However, performing these services at runtime can be needlessly costly from a computational point of view.

As an example of this, consider the case of two agents running on a single machine, one a database program and the other a database editor. Whenever the user of the editor makes a change, the results must be propagated to the database program. The editor must format an appropriate message; the facilitator must route and translate the message; and the database program must unformat the message in order to make the appropriate changes to the database.

Fortunately, under certain circumstances, it is possible to eliminate this overhead by moving portions of the computation to server initialization time. Suppose, in the example, that we know that a database program and editor are the only two agents interested in the information they are exchanging (at least for the time being). In this case, the facilitator can request that the agents implement their communication via remote procedure call, giving each the address of the other and taking itself out of the picture.

Of course, in order to preserve flexibility, it is essential that the facilitator retain the ability to request that the agents terminate or modify such connections in the event that the environment changes in an incompatible way. Although this check carries with it a certain amount of overhead, the advantage of this approach is that it eliminates the overhead associated with the transmission of messages; instead, the overhead is paid only when there are changes in system configuration, agents' interests, and so forth.

Connectivity

Our final facilitator-related topic concerns the issue of connectivity between facilitators and agents and between facilitators and other facilitators.

Since remote communication is more expensive than local communication, there is good reason for having at least one facilitator on each machine. Otherwise, in order for a program to communicate with another program on the *same* machine, it would have to send a message to a *remote* machine!

On the other hand, there is really no reason to have more than one facilitator per machine. Anything that can be handled by two facilitators can be handled by one facilitator. There can be no computational advantage, unless the two facilitators are running on different processors with the same machine.

What about the connection of agents to facilitators? While it is possible to consider a situation in which every agent is connected to every facilitator, this is impractical in settings, like the Internet, where there are likely to be many agents and many facilitators. For this reason, in federation architecture, I assume that every agent is connected to one and only one facilitator.

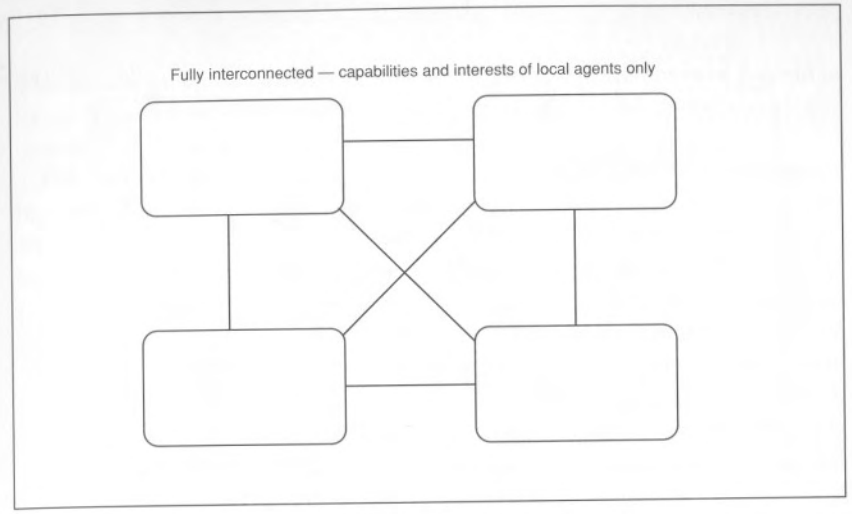

Figure 4. Full interconnection architecture.

Finally, there is the issue of inter-facilitator connectivity. Here, there are multiple choices, each with advantages and disadvantages.

The simplest sort of architecture is full interconnection, as suggested by figure 4. In this architecture, every facilitator is connected to every other facilitator. Since these connections are logical connections and not physical wires, this sort of architecture is feasible, though not necessarily desirable.

The disadvantage of this approach is the cost of interconnectivity. On a large network, like the Internet, the number of facilitators could be very large, and under this scheme every one would have to know about every other one.

An alternative that alleviates this difficulty is a spanning tree architecture, as suggested in figure 5. In this approach, facilitators are connected in such a way that there is a path from every facilitator to every other facilitator but there are no loops.

This approach is good because it allows connectivity without the cost of numerous connections. It has the disadvantage of being susceptible to failure when one of the nodes goes down, as this can break the network into disconnected components.

Finally, there is the general connectivity architecture. In this architecture, every facilitator is connected at least indirectly with every other facilitator, as in the spanning tree architecture, but there is no restriction that the connectivity be loop free.

Like the spanning tree architecture, a general connectivity architecture can be more economical of connections than a full interconnection architecture. It

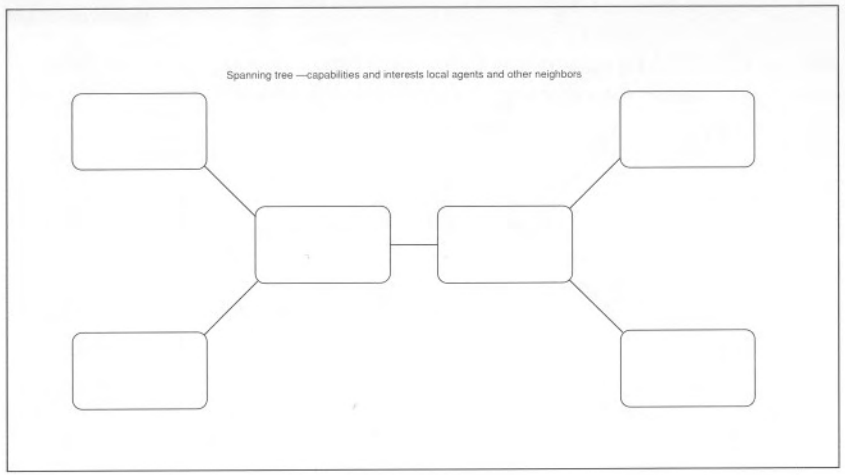

Figure 5. Spanning tree architecture.

has the added advantage that a failure of a node or connection does not necessarily disconnect different segments of the network.

Unfortunately, it has the disadvantage of possible loops. If one facilitator sends a message to a second and the second passes it on to a third and the third passes it on again, it might end up back where it started.

Fortunately, loops of this sort can be caught by adding sender information to each message (as in many mail protocols, for example) and checking for this information when a message is received. It can also be handled by having each facilitator save information about which messages it has sent. Either way the loops can be broken. The programming cost is a little higher, but the efficiency and reliability of the approach recommend it highly.

Another complexity in the spanning tree and general connectivity architectures stems from the need of facilitators to merge the interests of other facilitators in with those of their own agents in complicated ways. In a full connectivity architecture, each facilitator simply aggregates the interests of its local agents and passes those interests on to all other facilitators. Each facilitator uses this information to handle incoming requests. In the other two architectures, the interests passed on to neighbors are more complicated. A facilitator connected to two other facilitators must blend the interests of its first neighbor into the interests of its local agents in the specification it sends to its second neighbor; and it must blend the interests of its second neighbor into the interests of its local agents in the specification it sends to its first neighbor.

Example

This section presents a simple example of the federation architecture. Instead of focusing on the details, I present a broad picture of the types of software interoperation made possible.

First, a brief overview of the scenario. There is a computer systems manager in a publishing company who wants to upgrade the computers used by the sales staff to portable Pentium-based machines. The computer systems manager informs the facilitator of his interest in Pentium laptops. Sometime later, the computer product agent notifies the facilitator of the availability of a Pentium laptop, and this information is passed on to the computer systems manager by the facilitator. The computer systems manager asks the meeting scheduling agent to set up a joint meeting with the managers of the sales and finance departments to discuss the purchase of the new machines. The meeting scheduling agent gets the available times from the calendar agents for the sales and finance managers to schedule a meeting. We fill in some of the details below.

The computer systems manager sits at his terminal with a graphical user interface (GUI) and tells the facilitator that he is interested in being told of the availability of PC-compatible Pentium laptops. The GUI commands are translated into the following KIF fact, which is transmitted to the facilitator:

```
(<= interested cs-manager '(tell (available ,?manufacturer ,?model-name)))
   (= (denotation ?model-name) ?model) ; the model from its name
   (computer-family ?model PC)
   (laptop ?model))
```

There is a product agent that can answer queries about the computer family a product belongs to (e.g., PC, Apple) and which computers are laptops. It has specified its capabilities by transmitting the following facts to the facilitator:

```
(handles product-agent
   '(ask-one ,?variables (computer-family ,?computer ,?family)))
(handles product-agent '(ask-if (laptop ,?computer)))
```

Whenever a new piece of information is added to the product agent's knowledge base it notifies the facilitator of the fact. A new Micro-International 3600D computer is announced, and information about it is added to the knowledge base of the product agent. The product agent communicates the following KQML message to the facilitator:

```
(tell (available Micro-International 3600D))
```

The facilitator performs inference to see if any agent is interested in this fact. It finds that the cs-manager agent is interested, but only if the computer family of the 3600D is PC, and if the 3600D is a laptop. The facilitator cannot answer these questions locally. However, it forwards the queries to the product-agent, who can answer them. The product-agent responds positively to both queries, and the cs-manager is notified of the previous availability of the 3600D. A mes-

sage indicating this pops up on the GUI of the computer systems manager.

The computer systems manager uses his GUI to ask the facilitator to schedule a one hour meeting with the managers of the sales and finance groups during the week of December 12th to 16th. The GUI transmits the following KQML message to the facilitator:

```
(schedule-meeting (listof sales-manager finance-manager)
        (interval 12-12-94 12-16-94)
        60)
```

There is a scheduling agent that can schedule meetings. It previously transmitted the following fact to the facilitator:

```
(handles scheduler '(schedule-meeting ,?people ,?interval ,?meeting-duration))
```

The original meeting request is passed on to the scheduler agent by the facilitator. The scheduler is not able to schedule a meeting directly, since it does not have access to the calendars of the sales and finance managers. Therefore, the scheduling agent passes on the following query to the facilitator:

```
(ask-one ?x (calendar sales-manager (interval 12-12-94 12-16-94) ?x))
```

There is a datebook agent for the sales manager that records his calendar. It had previously notified the facilitator of its capability with the following fact:

```
(handles sales-manager-datebook
   '(ask-one ,?x (calendar sales-manager ,?interval ,?x)))
```

Similarly, there is a synchronize agent that can answer queries regarding the calendar of the finance manager.

The facilitator passes on the two queries of the scheduler to the sales-manager-datebook agent and the finance-manager-synchronize agent. The calendars returned by these agents are sent to the scheduling agent, who schedules the earliest possible meeting. The first available meeting time is transmitted to the facilitator, who finally forwards the results to the cs-manager.

This example illustrates a collection of points: anonymous interaction of agents through the use of a facilitator, interoperation of a variety of program types, different types of agent implementations incorporating legacy code, and the dual nature of agents as both clients and servers.

Some programs in the example are based on legacy code, such as the product agent which uses an SQL database for recording data about computers, the datebook calendar program, and the synchronize calendar program. Other programs are written from scratch, such as the scheduling agent that computes the intersection of available times for a group of meeting participants.

Different techniques are used to incorporate legacy code. The product agent uses an SQL database for recording facts, and it is "agentified" by providing a transducer to convert ACL into SQL commands and vice versa. The datebook calendar program is agentified by a wrapper: the source code is modified to support ACL communication. The meeting scheduling component of the datebook

and synchronize programs was rewritten in the scheduling agent to support a more general notion of time.

Finally, the example also illustrates the dual nature of agents as both providers and consumers of services. For example, the meeting scheduling agent can handle a request to schedule a meeting. However, in order to service this request, the scheduling agent must ask the facilitator for the calendars of the participants.

Applications

In this section, I describe some experiments designed to assess the power and limitations of the knowledge-sharing approach to software interoperation. While knowledge-sharing technology has potential value in many different application areas, I chose to concentrate my experiments on two particular application areas: computer-aided engineering and heterogeneous distributed information access.

Designworld

Designworld is the result of the first integration effort. In its current form, Designworld is an automated prototyping system for small scale electronic circuits built from standard parts (TTL chips and connectors on prototyping boards). The design for a product is entered into the system via a multimedia design workstation; the product is built by a dedicated robotic cell—in effect, a microfactory. If necessary, the product, once built, can be returned to the system for diagnosis and repair.

The Designworld system consists of eighteen processes on six different machines (two Macintoshes and four HP workstations). Each of the eighteen programs is implemented as a distinct agent that communicates with its peers via messages in ACL. Any one of these programs can be replaced by an ACL-equivalent program without changing the functionality of the system as a whole. Any agent can be moved to a different machine (with equivalent capabilities). Any agent can be deleted and the system will continue to run correctly, albeit with reduced functionality.

In the development of the system, there was virtually no communication between programmers, except at the very end; the discussion, when it occurred, was limited to negotiation on message vocabulary; and no reimplementation took place as a result of this negotiation (since the mediation of all disagreements was handled by the system's facilitator). The Designworld system is a good example of software interoperation through knowledge-sharing technology. However, as an experiment in interoperation, it is somewhat suspect since all of the software was developed by a single team.

PACT

Our second application of the technology deals with this experimental weakness. PACT (the Palo Alto Collaborative Testbed) is a multi-institutional testbed for research in the integration of engineering tools. PACT differs from Designworld in its emphasis on the integration of previously developed tools and tools developed by multiple teams without the benefit of institutionally enforced coordination.

The system in its current form incorporates four previously existing systems (i.e., Designworld, NVisage, NextCut, and DME) and includes several commercial products (e.g., Mathematica). Overall, there are thirty processes on eighteen different machines.

We have demonstrated the interoperation of the components of PACT by applying the system to the design of a particular electromechanical device, viz. an electronically controlled robotic manipulator. The various parts of the manipulator are modeled by the different systems participating in PACT. There is a simulation of the manipulator system as a whole based on coordinated piecemeal simulations performed by the participating systems. There is an example of a design change and effective communication of this change among the various systems.

Infomaster

Infomaster is a virtual information system that allows users to access a variety of heterogeneous distributed information sources from multiple perspectives. Infomaster accesses information stored in databases or knowledge bases using ACL, and uses facilitators to decompose, route, and translate requests, and assemble, route and translate responses.

The first information available through Infomaster concerns rental housing in the San Francisco Bay area. Every morning, an agent extracts the latest classified advertisements for rental housing from the World Wide Web sites of several newspapers. These are then separated into individual advertisements, parsed into a structured format, and loaded into a KIF knowledge base. This knowledge base has advertised that it can handle queries for rental housing. Once users have specified their query, they may determine how many rental advertisements satisfy their requirements and further constrain query in an iterative fashion. In the first day of availability for new Stanford students, Infomaster handled 3,000 queries. Support for additional sources and kinds of information are planned.

Issues and Summary

In order to provide adequate power and scalability for agent-based capabilities, current implementations of facilitators take advantage of automated reasoning technology developed by the artificial intelligence and database research 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.

Scalability is an important concern in the design of the federation architecture. There are three important issues: consistent vocabulary, inference cost, and knowledge base size. Interoperation in the federation architecture relies on the assumption that all agents agree to a shared ontology. However in a large system, multiple overlapping ontologies must be supported. Ideally, specialized ontologies can be built using existing ontologies. The ontologies are related in a directed graph, where each ontology can incorporate some or all of the terms and definitions of its parent ontologies, while overriding those that it must define differently.

The second scalability issue concerns inference cost. As the number of agents increases, the number of facts about agent capabilities, needs, and application-specific facts increases. However, the performance of the system should not degrade because of irrelevant facts. Ontologies help address some of the complexity. All requests are relative to an ontology, and the graph structure of the ontologies partitions the knowledge into smaller relevant sets. In addition, the facilitator controls the inference process by selecting the cheapest agent to handle a request and by avoiding infinite loops. Additionally, it is possible to guarantee desirable performance properties by placing restrictions on the rules a facilitator may accept. For example, if all facts are ground atomics (as in CORBA's IDL specification), then inference is reduced to database lookup, and the cost is logarithmic in the number of facts. If the facts are stratified (i.e., no recursive definitions), then it is possible to compute time bounds on inference. It is important to note that inference is expensive only with complex rules, and it is possible to enforce a policy of accepting only simple rules.

The third scalability issue deals with managing the size of the knowledge base. There are two aspects to this: application-specific facts, and meta-level specifications. Facilitators run continuously, and it is not possible to put a bound on the number of application-specific facts it may be told. The maintainer of each facilitator can enforce a policy for deciding what information to record. For example, a facilitator may follow the policy of recording only ground atomic facts, or it may record only facts in a given ontology. There may be a limit to the number of facts that a facilitator records, and it may discard some facts when a space limit is reached. Similarly, it is not possible to put a bound on the total number of agents in the system. A system can have a network of facilitators, with different agents connected to different facilitators. Each facilitator

must be capable of transmitting a request to any agent that can handle it, independent of its location. To minimize the number of capability and interest specification facts, each facilitator summarizes the capabilities and interests of its directly connected agents, and passes on this summary to its neighboring facilitators. The summary reduces the number of facts and may involve generalization. For example, if one directly-connected agent can answer questions about the dealers of Apple computers and another directly-connected agent can answer questions about IBM dealers, then the facilitator may summarize the answers by informing its neighboring facilitators that it can answer questions about the dealers of all personal computers. There is a space-time tradeoff here: fewer less-precise specifications vs. a larger number of more precise specifications. It is acceptable for an agent to handle a request by indicating that it cannot answer it, for example if its specifications are too general.

In my treatment so far, I have assumed that there is sufficient common interest among the agents that they will frequently volunteer to help each other and receive no direct reward for their labor. As the Internet becomes increasingly commercialized, I envision a world where agents act on behalf of their creators to make a profit. Agents will seek payments for services provided and may negotiate with each other to maximize their expected utility, which might be measured in a form of electronic currency. These problems mark the intersection of economics and distributed artificial intelligence (DAI). Several researchers in DAI are using tools developed in economics and game theory to evaluate multiagent interactions. I am currently examining extensions to the federation architecture to incorporate some of these capabilities.

I have ignored several other key problems in my presentation, such as security, crash recovery, inconsistencies in program specifications, and so forth. Although I have partial solutions to these problems, further work is needed.

There are many applications of knowledge-sharing technology in offline software integration that I have not discussed, such as in software documentation, retrieval of components from software libraries based on this documentation, and software verification.

In this chapter, I have taken a brief look at how knowledge-sharing technology can be used to promote software interoperation. My 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, I believe that the introduction of knowledge-sharing technology will be an important step toward achieving this vision.

Acknowledgments

Many people contributed to the ideas presented in this chapter. The most significant contributions came from participants in the ARPA Knowledge Sharing Initiative, the Designworld Project, the PACT Project, and the ABSE Project, most notably Mark Cutkosky, Richard Fikes, Rich Fritzson, Mark Gisi, Tom Gruber, Jon Gustafson, Pierre Huyn, Marta Kosarchyn, Reed Letsinger, Don MacKay, Ofer Matan, Bill Mark, Greg Olsen, Vishal Sikka, Marty Tenenbaum, and Jay Weber. This chapter is as much theirs as it is mine. Support for the authors' work was provided by Hewlett-Packard under grant number 172S338 and by the Office of Naval Research under contract number N00014-90-J-1533.

Bibliography

Cutkosky, M.; Englemore, R.; Fikes, R.; Gruber, T. R.; Genesereth, M.; Mark, W.; Tenenbaum, J. M.; and Weber, J. 1993. PACT: An Experiment in Integrating Concurrent Engineering Systems. *IEEE Computer* 26(1): 28–37.

Finin, T.; Weber, J.; Wiederhold, G.; Genesereth, M.; Fritzson, R.; McGuire, J.; McKay, D.; Shapiro, S.; Pelavin, R.; and Beck, C. 1992. Specification of the KQML Agent Communication Language (Official Document of the DARPA Knowledge Sharing Initiative's External Interfaces Working Group), Technical Report 92-04, Enterprise Integration Technologies, Inc., Menlo Park, California.

Genesereth, M. R. 1991a. An Agent-Based Approach to Software Interoperation, Logic-91-6, Department of Computer Science, Stanford University.

Genesereth, M. R. 1991b. DESIGNWORLD. In Proceedings of the IEEE Conference on Robotics and Automation, 2-785–2-788. Washington, D.C.: IEEE Computer Society.

Genesereth, M. R., and Fikes, R. 1992. Knowledge Interchange Format Version 3.0 Reference Manual, Logic Group Report, Logic-92-1, Department of Computer Science, Stanford University.

Gruber, T. R. 1993. A Translation Approach to Portable Ontology Specification. *Knowledge Acquisition* 5(2): 199–220.

Gruber, T. R. 1992. ONTOLINGUA: A Mechanism to Support Portable Ontologies, Version 3.0, Stanford Knowledge Systems Laboratory Technical Report, KSL 91-66, Department of Computer Science, Stanford University.

Neches, R.; Fikes, R.; Finin, T.; Gruber, T.; Patil, R.; Senator, T.; and Swartout, W. R. 1991. Enabling Technology for Knowledge Sharing. *AI Magazine* 12(3): 36–56.