A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation

Michael R. Genesereth
Computer Science Department
Stanford University
genesereth@cs.stanford.edu

Narinder P. Singh
Computer Science Department
Stanford University
singh@cs.stanford.edu

Mustafa A. Syed
Center for Information Technology
Stanford University
syed@cit.stanford.edu

15^{th} November, 1994

Abstract

The support for automatic interoperation of software components can reduce cost and provide greater functionality. This paper describes a novel approach to software interoperation based on specification sharing. Software components, called agents, provide machine processable descriptions of their capabilities and needs. Agents can be realized in different programming languages, and they can run in different processes on different machines. In addition, agents can be dynamic – at run time agents can join the system or leave. The system uses the declarative agent specifications to automatically coordinate their interoperation. The architecture supports anonymous interoperation of agents, where each agent has the illusion that the capabilities of all the other agents are provided directly by the system. The distinctive feature of this approach is the expressiveness of the declarative specification language, which enables sophisticated agent interoperation, e.g., decomposing complex requests into a collection of simpler requests, and translating between the interface of a requesting agent and the interface of an agent that can service the request. The agent-based interoperation scheme relies on a shared vocabulary, and it is our thesis that more effective software interoperation is made possible by agreeing to a shared declarative vocabulary, than by agreeing to shared programming constructs (e.g., subroutine names and their argument types).

1 Introduction

Computer users operate in highly heterogeneous software environments. Programs are written by different people at different times in different languages and in different styles. Programs run on machines produced by different manufacturers and located at different sites. Taken together, these programs provide a wide variety of information and services in a variety of domains. While most 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 and thereby solve problems that cannot be solved alone.

Unfortunately, getting programs to work together often necessitates extensive work on the part of the users of those programs or their programmers – to learn the characteristics of completed programs and to negotiate communication formats and protocols for programs under development. What's more,

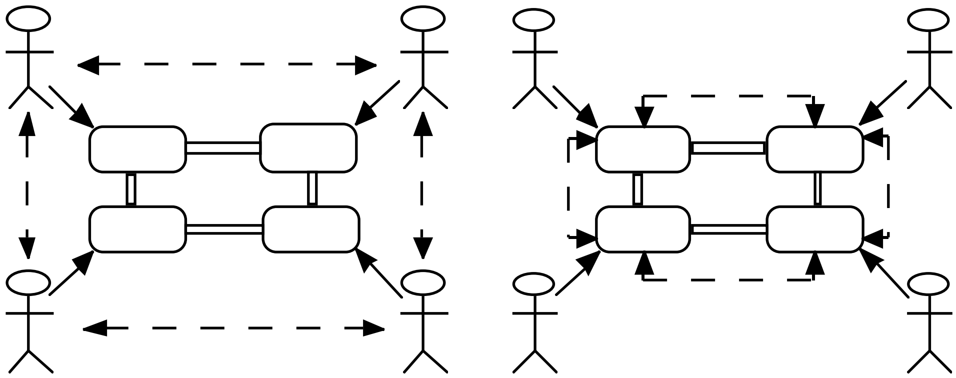

Figure 1: Manual and automated coordination.

the resulting systems are usually very rigid – components often cannot be modified or replaced without subsequent rounds of negotiation and programming.

In order to deal with these problems, the systems community has developed various pieces of technology to transfer much of the burden of interoperation from the creators and users of programs to the programs themselves, including such things as standard communication languages, subroutine libraries to assist programmers in writing interoperable software, and system services to facilitate interoperation at runtime. Unfortunately, the current technology is too limited to support the ideal of automated interoperation. Existing standards are not sufficiently expressive to allow the communication of the definitions, theorems and assumptions that are often needed for systems to interoperate. Current subroutine libraries provide little support for increased expressiveness. Directory assistance programs and brokers are limited by inexpressiveness in the languages used to document programs and by their lack of inferential capability.

Recent progress by researchers in the ARPA-sponsored Knowledge Sharing Effort suggests that it may now be possible to remedy these deficiencies through the use of knowledge sharing technology. The basis for this approach is a highly expressive communication language, called ACL (for Agent Communication Language). Programs (called agents) use ACL to supply machine-processable documentation to system programs (called facilitators), which then coordinate their activities. The agent-based approach to interoperability is based on the notion of shared abstraction. In this approach, the burden of interoperability is placed on the agents and facilitators rather than the programmers or users. Individual programmers can write their programs without knowledge of the data structures and algorithms of other programs, without knowledge of the hardware configuration in which those programs are going to be run. Computer users can avail themselves of the services of different programs by asking their systems to coordinate their interaction. The contrast between manual and automated coordination is illustrated in Figure 1.

In this paper, we examine various aspects of agent-based technology. We discuss language issues, handling legacy software and in particular we describe the implementation of a facilitator and how it is used for distributed and anonymous problem solving. We also discuss several outstanding issues and identify some areas of future research and development.

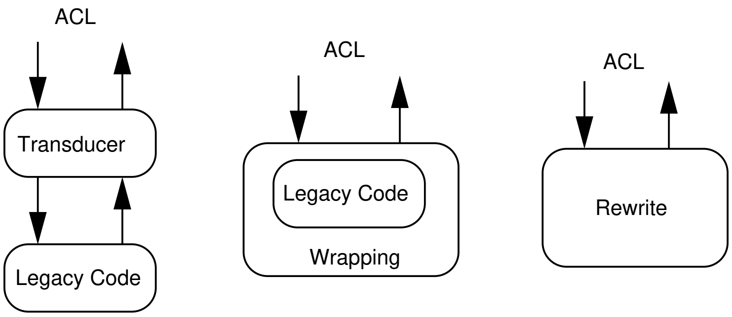

Figure 2: Three approaches to agentification.

2 Agents

In the approach to interoperation described here, application programmers write their programs as *agents*. An agent is obliged to communicate in the Agent Communication Language (ACL) (described in the next section). Upon start-up, an agent informs the system of its capabilities and needs. It then enters normal operation – it makes requests of the system when it is incapable of fulfilling them itself, and it acts to the best of its abilities to satisfy the requests the system makes of it.

The criterion for agenthood is behavioral. An entity is an agent if and only if it communicates correctly in ACL. This means that the entity must be able to read and write ACL messages, and 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, veracity (an agent must tell the truth), autonomy (an agent may not constrain another agent to perform a service unless the other agent has 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.

For our purposes here, it is sufficient to say that the use of ACL brings with it behavioral constraints. However, this leaves open 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, so 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 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. See Figure 2.

One approach 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.

A second approach to dealing with legacy software 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 that 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 where there is no interprocess communication ability in the original program. However, it requires that the code for the program be available.

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 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; its output is passed to an assembly planning program; and so forth. 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.

3 Agent Communication Language

Communication language standards facilitate the creation of interoperable software by decoupling implementation from interface. So 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, 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 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.

ACL attacks these problems by mandating a universal communication language, one in which inconsistencies and arbitrary notational variations are eliminated. ACL 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 [11] have defined the components of ACL that satisfy these needs.

ACL can best be thought of as consisting of three parts – its vocabulary, an "inner language" called KIF (short for Knowledge Interchange Format), and an "outer" language called KQML (short for 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 of 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.

Note that the existence of such a dictionary does not imply that there is only one way of describing an application area. Indeed, the dictionary can contain multiple *ontologies* for any given area. For example, it contains vocabulary for describing three-dimensional geometry in terms of polar coordinates, rectangular coordinates, cylindrical coordinates, etc. 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.

3.1 KIF

KIF [4] is a prefix version 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 sentences shown below 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 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, and so forth). The expression shown below is an example of a complex sentence in KIF. It asserts that a number`?x` raised to another `?n` is positive if `?n` is real and even.

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

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 agent 1 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 output stream; the final step is to add a carriage return to get to a new 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.

3.2 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 [3]. 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 dialog between the sender and the receiver, and KQML provides support for a wide variety of such dialog 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 dialog 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 dialog 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, as illustrated here, KQML also contains support for delayed and conditional operations, requests for bids, offers, promises, and so forth.

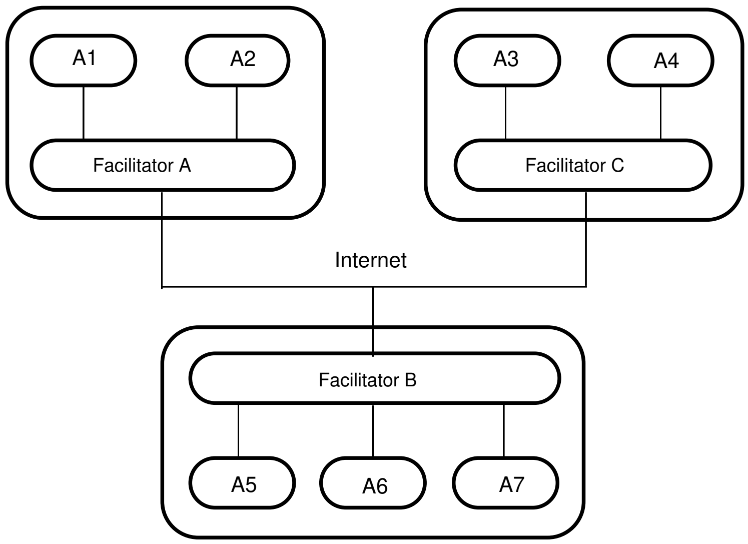

Figure 3: Federated System.

4 Federation Architecture

The previous sections defined the requirements for an agent, including the restriction that all agents communicate in ACL. These requirements place no restrictions on the possible connectivity of agents, and agents are free to coordinate their activities by communicating with each other directly. Instead of leaving the burden of interoperation to the agents, the Federation Architecture provides a collection of services to support these activities. An important feature of the Federation Architecture is the support for anonymous interaction between agents. In return for these services the agents must relinquish some of their autonomy to the system and agree to abide by additional constraints. To each agent it appears that there is a single system agent that handles all requests directly. This system agent, called a facilitator,[1] realizes a virtual agent with the capabilities of all the other agents.

Figure 3 shows a picture of a Federation Architecture in the simple case where there are three machines, with one facilitator per machine, one machine with three agents, and the remaining with two agents each. Agents are restricted to communicate directly with facilitators. 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 use 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.

[1]The concept of a facilitator is a generalization of *mediators*, as described in [16].

4.1 Facilitators

Facilitators are the system-provided agents that coordinate the activities of the other agents in the Federation Architecture. The network of facilitators keep each other informed of the agents connected to them and the facts communicated by the agents.

Facilitators provide a collection of services including:[2]

- white pages– finding the identity of agents by name, e.g., "What agents are connected?," or "Is agent x connected?."
- yellow pages– finding the identity of agents capable of performing a task, e.g., "What agents are capable of answering the query x?"
- direct communication– sending a message to a specific agent.
- Content-Based Routing (CBR)– 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.
- 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. Similar to CBR, the facilitator makes use of the specifications and application-specific information provided by the agents to accomplish this.
- 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.
- 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 interoperation of agents in a system is independent of their implementation. This is similar to the abstraction capability provided in traditional object-oriented programming languages. 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.

Specifying Agent Capabilities

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, e.g., `'(ask-one ,?variables (dealer ,?dealer ,?vendor))` in the first example. This specification is similar to the object interface specifications in IDL (as used in CORBA).

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

[2] The examples are in English for clarity.

```
(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 this to A_1. Agent A_1 is completely unaware of the sequence of steps performed in servicing its request.

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

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

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

Specifying Agent Needs

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:

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

This states that the `cs-manager` agent is only interested 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`.

Processing Messages

Each facilitator is connected to a set of local agents, and some number of external facilitators. It is the responsibility of facilitators to process the information communicated on its connections. There are two types of KQML messages: requests and announcements.

Facilitators use their knowledge base to service requests, which may involve one or more of: identifying a single agent that can handle the request, translating the request into the vocabulary understood by the receiving agent, decomposing a request into simpler sub-requests, etc. For announcements, the facilitator examines its knowledge base to find out which agents are interested in the announcement. This may require translating the vocabulary of the original announcement to a form understood by the interested recipient.

Facilitators use automated inference to reason about agent specifications and application specific facts. The inference procedure, based on model-elimination, is an extension of the familiar backward chaining inference rule used in Prolog [7, 15]. The extensions permit the inference procedure to be complete for first-order logic.

For a request, the facilitator uses backward inference to find an answer. For example, if the request is `(ask-one ?x (dealer ?x Apple))`, then the facilitator finds a single proof for the query `(dealer ?x Apple)`, and returns the binding of the variable `?x` for a successful proof (if one can be found). It uses the facts in its local knowledge base and the specifications of the capabilities of the other agents in trying to find a proof. For this example, the facilitator does not have any facts about Apple Computer dealers in its local knowledge base. The facilitator next examines the agent specifications to see if an agent can handle the quoted query `'(ask-one ?x (dealer ?x Apple))`. The `business-agent` can handle this request, and the facilitator forwards the request to it.

This example illustrated a simple scenario where a fact was either available in the facilitator or from a remote agent. In general, servicing a request can require a complicated sequence of inference steps going back and forth between agents. For example, if the query is to find a local dealer, then the facilitator will first find a dealer and then confirm that the dealer is located in Santa Clara county. It is the responsibility of the facilitator to give the illusion of a virtual theory, which includes the facts in the facilitator knowledge base and the facts the agents can provide. Instead of recording all the facts of the agents locally, the facilitator makes requests of the needed facts from the agents at run time in the process of finding a proof.

A similar sequence of inference steps is followed in handling an announcement.

4.2 Scalability

An important concern in the design of the Federation Architecture is its scalability. 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. For example, two agents should not use the relation `apple` to mean different things–one using it to mean fruits, while the other using it to mean a computer manufacturer. In a small setting, with only a few agents, it is possible to agree to a shared vocabulary by direct communication between the agent writers. In a large system, however, this is impractical, especially if the agent writers are from different communities which have specialized languages with conflicting vocabularies, e.g., architects and electrical engineers disagreeing on the meaning of the word "column". The Federation Architecture provides a mechanism for supporting multiple ontologies. A collection of agents can define a new ontology for their use. Each ontology provides a dictionary of terms and their definitions, which all agents must be consistent with. The definitions include an English description for humans, and a formal KIF specification for facilitators and agents.

In order not to require ontologies to be defined from scratch, it is possible to build ontologies using other existing ontologies. The ontologies are related in a directed graph, where each ontology can incorporate some or all the terms/definitions of its parent ontologies and it can override those that it wishes to 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 due to 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 (conjunct and/or disjunct ordering), and avoids infinite loops (with identical-ancestor pruning and/or iterative-deepening) [14, 10].

It is possible to guarantee desirable performance properties by placing restrictions on the rules a facilitator accepts. For example, if all facts are ground atomics (similar to IDL in CORBA), then inference is simplified to database lookup, and the cost is logarithmic in the number of facts. If the facts are stratified (no recursive definitions), then it is possible to compute time bounds on inference. It is important to note that inference is only expensive with complex rules, and it possible to enforce a policy of only accepting 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 only record ground atomic facts, or it may only record facts in a given ontology. There may be a limit to the number of facts that a facilitator records – it may flush some facts when a space limit is reached. Throwing away information may lead to incompleteness, but this may be unavoidable due to storage limitations.

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/interests of its directly connected agents, and passes on this summarization as its capability/interests to the neighboring facilitators. The summarization reduces the number of facts and may involve generalization, e.g., if one directly connected agent can answer questions about the dealers of Apple computers and another directly connected agent can answer questions about the dealers of IBM, then the facilitator may summarize this 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, or a larger number of more precise specifications. It is acceptable for an agent to handle a request by indicating that it cannot answer it (e.g., if its specifications are too general), however, this has the disadvantage of wasting effort.

5 Example

This section presents a simple example of the Federation Architecture. Instead of focusing on the details we 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. Some time 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 department 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 of a product (PC, Apple, etc.), 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 agents knowledge base it notifies the facilitator of it. 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 message 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 sales and finance during the week of December 12^{th} to 16^{th}. 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 (all agents communicate directly with a facilitator), interoperation of a variety of program types, different types of agentification of legacy code, and the dual nature of agents as both clients and servers.

Some programs in the example are legacy code, e.g., the product agent which utilizes an SQL database for recording computer information, the Datebook calendar program, and the Synchronize calendar program. Other programs are custom written, for example the scheduling agent which computes the intersection of the available times for a group of participants.

The example is realized using different techniques for agentifying legacy code. The product agent utilizes an SQL data base for recording facts, and it is agentified by providing a transducer to converting 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 of the meeting.

6 Related Work

The agent-based approach to software interoperation 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 interoperation, agents use a common language with an agent-independent semantics.

The concepts of system services in support of software interoperation is not new here. For example, directory assistance programs facilitate software interoperation by providing a way for programs to discover which programs can handle which requests and which programs are interested in which pieces of information. Distributed object managers (like CORBA, OLE, DSOM, OpenDoc) provide location transparency for object-oriented systems, routing messages to objects without requiring senders to know the locations of those objects. Automatic brokers (like the Publish and Subscribe capabilities on the Macintosh, DDE, BMS, Tooltalk, etc.) 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 interoperation 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. In order 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 between individual agents and then stepping out of the picture).

In our treatment so far, we 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, 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 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). A number of researchers in DAI are using tools developed in economics and game theory to evaluate multi-agent interactions [17, 8]. We are currently examining extensions to the Federation Architecture to incorporate some of these capabilities.

7 Conclusion

The agent-based approach to software interoperation described here has been developed into a practical technology and has been put to use in a variety of applications necessitating interoperation (e.g. concurrent engineering [2], database integration, and so forth) and is being used at multiple institutions in the construction of software for the national information infrastructure.[3]

In order to provide these capabilities, current implementations of facilitators take advantage of automated reasoning technology developed in the Artificial Intelligence 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.

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

In this paper, 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 that the introduction of agent technology will be an important step toward achieving this vision.

References

[1] The Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Revision 1.1 91.12.1, December, 1991.

[2] Cutkosky, M. et al. "PACT: An Experiment in Integrated Engineering Systems," *Computer 26*, 1(1993), 28-37.

[3] Additional information about the Federation Architecture, including software, can be obtained from Mosaic using the URL `http://logic.stanford.edu/sharing.html`.

[3] Finin, T., and Wiederhold, G. "An Overview of KQML: A Knowledge Query and Manipulation Language," available through the Stanford University Computer Science Department, 1991.

[4] Genesereth, M. R., Fikes, R. E. et al. " Knowledge Interchange Format Version 3 Reference Manual," Logic-92-1, Stanford University Logic Group, 1992.

[5] Genesereth, M. R. and Ketchpel, S. "Software Agents," *Communications of the ACM,* Vol 37, no. 7, July 1994, pp. 48-53.

[6] Genesereth, M. R. and Singh, N. "A Knowledge Sharing Approach to Software Interoperation," Logic Group, Computer Science Department, Stanford University, 1993.

[7] Genesereth, M. R., and Singh, N. "Epilog 1.0 for Lisp," available from Mosaic with the URL `http://Logic.Stanford.edu/sharing/programs/epilog/documentation/`.

[8] Gymtrasiewicz, 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] Korf, R. E. "Linear-space Best-first Search: Summary of Results," in *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92),* pages 533-538, Menlo Park, California, 1992. AAAI Press.

[11] Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. "Enabling Technology for Knowledge Sharing," *AI Magazine 12*, 3(1991), 36-56.

[12] "Object Linking and Embedding," *Microsoft Technical Backgrounder OLE 2.0.*

[13] "OpenDoc Technical Summary," The OpenDoc Design Team, Apple Computers, Cupertino, California, October, 1993.

[14] Smith, D. E., Genesereth, M. R., and Ginsberg, M. L. "Controlling Recursive Inference," *Artificial Intelligence*, 30 (3): 343-389, December 1986.

[15] Stickel, M. E. "A Prolog Technology Theorem Prover: A New Exposition and Implementation in Prolog," Technical Note 464, SRI International, June 1989.

[16] Wiederhold, G. "The Architecture of Future Information Systems, Stanford University Computer Science Department," 1989.

[17] Zlotkin, G. "Mechanisms for Automated Negotiation among Autonomous Agents," Ph.D. Dissertation. Hebrew University. February 1994.