AFRL-IF-RS-TR-1998-52
Final Technical Report
April 1998

INTELLIGENT AGENT INTEGRATION TECHNOLOGY

Lockheed Martin

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. A521

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-1998-52 has been reviewed and is approved for publication.

APPROVED: [illegible]

RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: [illegible]

NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** April 1998	**3. REPORT TYPE AND DATES COVERED** Final Sep 93 - Jun 97

4. TITLE AND SUBTITLE INTELLIGENT AGENT INTEGRATION TECHNOLOGY	**5. FUNDING NUMBERS** C - F30602-93-C-0177 PE - 62301E PR - A521 TA - 00 WU - 01
6. AUTHOR(S) Donald P. McKay, Tim Finn, Stuart Shapiro, and Nick Roussopoulos	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lockheed Martin 590 Lancaster Avenue P.O. Box 4001 Frazer PA 19355-1808	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714 Air Force Research Laboratory/IFTB 525 Brooks Road Rome NY 13441-4505	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-1998-52

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTB/(315) 330-3577

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; distribution unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

The major areas of research described in this report include the contribution to the Knowledge Query and Manipulation Language (KQML) specification under the DARPA-sponsored Knowledge Sharing Initiative and the developing of a scaleable and an efficient implementation of information system components for ontological translation and effective cache-based implementations. This effort focused primarily upon developing Intelligent Agent Integration Technology and demonstrating it within distributed agent systems. This effort was a joint university/industry project in which technology development was supported by several university participants. The integration technology developed is based on KQML, a language and protocol intended to support interoperability among intelligent agents in a distributed application. The technical scope is the coordination of multiple "intelligent agents" which must communicate with one another. This report presents the basic accomplishments achieved which include: software systems and associated documentation for communication among multi-agent systems using the Knowledge Query and Manipulation Language (KQML); performance metrics and instrumentation for intelligent agent communication and knowledge base access to databases which highlight performance issues, aspects and potential improvements for large-scale architectures; and study of ontological mediation issues and research.

14. SUBJECT TERMS Computers, Software, Databases, knowledge Base, Artificial Intelligence	**15. NUMBER OF PAGES** 176
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTELLIGENT AGENT INTEGRATION TECHNOLOGY

Donald P. McKay
Tim Finn
Stuart Shapiro
Nick Roussopoulos

Contractor:	Lockheed Martin
Contract Number:	F30602-93-C-0177
Effective Date of Contract:	September 1993
Contract Expiration Date:	June 1997
Program Code Number:	A5210001
Short Title of Work:	Intelligent Agent Integration Technology
Period of Work Covered:	Sep 93 - Jun 97
Principal Investigator:	Donald P. McKay
Phone:	(610) 407-3527
AFRL Project Engineer:	Raymond A. Liuzzi
Phone:	(315) 330-3577

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Raymond A. Liuzzi, AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505.

Table of Contents

1. Summary

1.1 LANGUAGE AND PROTOCOL FOR INFORMATION EXCHANGE

Large scale information networks such as that envisioned for the National Information Infrastructure (NII) require flexible, scaleable and information-oriented communication infrastructures. Most current efforts still focus on data-level communications and ignore the issues of flexibility and scalability for the large evolving information networks now being built. Lockheed Martin, along with other industrial and university collaborators, has developed the Knowledge Query and Manipulation Language, KQML, to support these and future information networks. Significantly, KQML supports an scaleable architecture for agents including software and other information services.

Lockheed Martin has demonstrated KQML supporting efficient communication within the DARPA/Rome Lab Planning Initiative (ARPI) supporting initial communication within the ARPI common prototyping environment (CPE) and subsequently in a demonstration of an intelligent information services architecture. KQML was used to integrate information agents at UCLA, USC ISI and Lockheed Martin of the internet in a demonstration of a distributed intelligent information system for military transportation logistics. In a previous use within the CPE, the Lockheed Martin KQML implementation in Common Lisp has been measured to be as efficient as standard RPC mechanisms from Common Lisp sustaining interaction rates of 50 milliseconds for round trip experiments. In addition, a compression technique for ARPI was demonstrated which resulted in a 4x reduction in space and over a 2x reduction in communication time for large messages.

We have also established and demonstrated an Intelligent Resource Agent Architecture in the DARPA Computer Aided Education and Training Initiative (CAETI) supporting an infrastructure of reusable agents. These have been used to develop demonstrations within both K-12 education and adult training domains. Further, within CAETI, Lockheed Martin established KQML as one of the "Minimal Architecture" communication standards – this was a significant step forward as other heretofore non-participants in the KQML specification and promulgation began to participate, e.g., Teknowledge developed and used a variant of KQML support the interoperation of a set of component agents for intelligent tutoring systems using KQML to satisfy a CAETI program goal of distributed and reusable tutoring components.

Lockheed Martin is also using and promulgating KQML within other DARPA programs such as the Advanced Logistics Program (ALP) in which KQML will be used to support a distributed architecture for "sentinel agents" monitoring and responding to changes in status of Long Term Agreements (a kind of supplier contract with the Defense Logistics Agency). This infrastructure is also being used as a part of a Rapid Supply demonstration that Lockheed Martin is participating in as a part of the ALP program. In a similar effort, Lockheed Martin will apply the results of this project to efforts in support of the Planning and Decision Aids Technology Integration Experiments integrating planning and scheduling systems. Preliminary use and integration has been in place with David Wilkens' planning group at SRI for over the past year.

Finally, the KQML implementation and system is being used extensively within Lockheed Martin C2 Integration Systems to support advanced information systems employing agents and agent-based mediated architectures.

1.2 Objective

This research has significantly advanced the state of the art in scaleable Intelligent Agent Integration Technology. The major areas of research contributed to this effort include the contribution to the

Knowledge Query and Manipulation Language (KQML) specification under the DARPA-sponsored Knowledge Sharing Initiative, the developing a scaleable and an efficient implementation of information system components for ontological translation and effective cache-based implementations.

1.3 Approach

This effort focused primarily upon developing Intelligent Agent Integration Technology and demonstrating it within distributed agent systems. This effort was a joint university/industry project in which technology development will be principally supported by the university participants including Tim Finin at the University of Maryland - Baltimore County (UMBC), Stuart Shapiro at the State University of New York at Buffalo (SUNY/Buffalo), and Nicholas Roussopoulos at the University of Maryland - College Park (UMD). Lockheed Martin will serve primarily as a technology provider for KQML.

The integration technology we developed is based on KQML, a language and protocol intended to support interoperability among intelligent agents in a distributed application. The technical scope is the coordination of multiple "intelligent agents" which must communicate with one another.

- Which other agents to communicate with.
- How to establish a reliable communication channel with them.
- What protocol to use in the ensuing dialogue.
- What language to use to exchange information knowledge.
- What terms within the language to use to guarantee that the other agent will interpret the expressions in the same way.
- How to handle inconsistent information and the eventual mis-matches that arise from different views and representations of the world.

1.4 Progress

Lockheed Martin developed and packaged an initial release of the KQML C and Common Lisp implementations suitable for use and for further joint collaborative development by the university participants. This release served as the baseline software and documentation for use by all participants. University participants at UMBC developed a formal semantics for KQML as well as designed and implemented a hierarchical agent name service and completed an initial security study. SUNY/Buffalo investigated the formal conditions under which it is meaningful for two software systems to exchange information which requires semantic translation, e.g., more than simple data representation translations such as unit conversions. UMD extended a model of adaptive caching for distributed systems, analyzed its performance and performed performance experiments.

1.5 Technology Transition

Key technologies exported from this project include intelligent interfaces to databases and interagent communication. This software system has been exported for use to other sites and DARPA projects such as ARPI, PDA, ALP, CAETI and other internal Lockheed Martin efforts.

Systems -

- Knowledge Query and Manipulation Language (KQML) - KQML defines a common language and protocol for intelligent agent communication and its development is supported via the ARPA

Knowledge Sharing Initiative. KQML implementations exist for Common Lisp (Lucid) and C running on Sun UNIX platforms with a minimum of 16MB of physical memory. POC: Robin McEntire, Lockheed Martin, (610) 407-3567, Robin.A.McEntire@lmco.com.

1.6 Accomplishments

In this report, we present the basic accomplishments achieved under this contract which include:

- Software systems and associated documentation for communication among multi-agent systems using the Knowledge Query and Manipulation Language (KQML).
- Performance metrics and instrumentation for intelligent agent communication and knowledge base access to databases which highlight performance issues, aspects and potential improvements for large-scale architectures..
- Study of ontological mediation issues and research.

We continued use of the Common Lisp implementation of KQML within demonstrations and experiments under the DARPA Rome Lab Planning Initiative. KQML has been used to support information mediators from UCLA, USC ISI and Lockheed Martin to interoperate over the internet. The C implementation of KQML is also used in the development of a Agent Name Server implemented in C.

1.7 Documents Produced

1.7.1 Government Documents

The following documents describe the software developed under this contract:

1. Software Design Document for the Knowledge Query and Manipulation Language (KQML)
2. Software User's Manual and for the Knowledge Query and Manipulation Language (KQML)

1.7.2 Technical Papers

The following technical papers were developed or contributed to under this contract; on-line versions can be accessed for many from the UMBC web site http://cs.umbc.edu/kqml or http://cs.umbc.edu/agents, additional site is at http://www.paoli.atm.lmco.com/

Hans Chalupsky, Stuart C. Shapiro & Alistair E. Campbell, "Ontological Mediation: An analysis" Technical Report, Department of Computer Science, SUNY/Buffalo, 1994.

C. Chen and N. Roussopoulos, "Adaptive selectivity estimation using query feedback", Technical Report, Department of Computer Science, University of Maryland College Park, May 1994.

A. Delis and N. Roussopoulos, "Management of updates in the enhanced client-server DBMS", Technical Report, Department of Computer Science, University of Maryland College Park, June 1994

Donald McKay, Jon Pastor, Robin McEntire and Tim Finin, An architecture for information agents, in "Advanced Planning Technology". (ed. Tate,A.), The AAAI Press, Menlo Park, CA., USA, May 1996, ISBN 0-929280-98-0.

Chelliah Thirunavukkarasu, Tim Finin and James Mayfield, Secret Agents -- A Security Architecture for the KQML agent communication language, proceedings of the ACM CIKM Intelligent Information Agents Workshop, Baltimore, December 1995.

Tim Finin, Chelliah Thirunavukkarasu, Anupama Potluri, Donald McKay, and Robin McEntire, On Agent Domains, Agent Names and Proxy Agents, proceedings of the ACM CIKM Intelligent Information Agents Workshop, Baltimore, December 1995.

James Mayfield, Yannis Labrou, and Tim Finin, Evaluation of KQML as an Agent Communication Language, in Intelligent Agents Volume II -- Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages. M. Wooldridge, J. P. Muller and M. Tambe (eds). Lecture Notes in Artificial Intelligence, Springer-Verlag, 1996.

James Mayfield, Yannis Labrou and Tim Finin. Evaluation of KQML as an Agent Communication Language, IJCAI-95 Workshop on Agent Theories, Architectures, and Languages, Montreal, Quebec, 19-20 August 1995.

Tim Finin, Yannis Labrou, and James Mayfield, KQML as an agent communication language. In Jeff Bradshaw (Ed.), ``Software Agents", MIT Press, Cambridge, to appear 1997.

James Mayfield, Yannis Labrou, and Tim Finin, Desiderata for Agent Communication Languages , Proceedings of the AAAI Symposium on Information Gathering from Heterogeneous, Distributed Environments, AAAI-95 Spring Symposium, Stanford University, Stanford, CA. March 27-29, 1995.

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire, "The Knowledge Query and Manipulation Language for Information and Knowledge Exchange", In the *Proceedings Third International Conference on Information and Knowledge Management (CIKM'94),* November 1994.

Yannis Labrou and Tim Finin, "A semantics approach for KQML—a general purpose communication language for software agents", In the *Proceedings Third International Conference on Information and Knowledge Management (CIKM'94),* November 1994.

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire, "KQML: An Information and Knowledge Exchange Protocol" , in Kazuhiro Fuchi and Toshio Yokoi (Ed.), "Knowledge Building and Knowledge Sharing", Ohmsha and IOS Press, 1994.

Tim Finin, Don McKay, Rich Fritzson and Robin McEntire, KQML - A Language and Protocol for Knowledge and Information Exchange. *Proceedings of the 13th International Distributed Artificial Intelligence Workshop*. July, 1994.

Specification of the KQML Agent-Communication Language. The ARPA Knowledge Sharing Initiative External Interfaces Working Group

1.8 Major Software Systems

We have developed a major release of the KQML. The release notes and documentation described the basic features of the software.

2. KQML – AN AGENT COMMUNICATION LANGUAGE

Intelligent Agent Integration Technology

This section appeared in The Proceedings of the Third International Conference on Information Management (CIKM), ACM Press, November 1994. It details project background and architectural underpinnings for the KQML system developed as a part of this contract.

KQML as an Agent Communication Language *

Tim Finin and Richard Fritzson
Computer Science Department
University of Maryland Baltimore County
Baltimore MD USA
finin@cs.umbc.edu
fritzson@cs.umbc.edu

Don McKay and Robin McEntire
Valley Forge Laboratory
Unisys Corporation
Paoli PA USA
mckay@vfl.paramax.com
robin@vfl.paramax.com

Abstract

This paper describes the design of and experimentation with the Knowledge Query and Manipulation Language (KQML), a new language and protocol for exchanging information and knowledge. This work is part of a larger effort, the ARPA Knowledge Sharing Effort which is aimed at developing techniques and methodology for building large-scale knowledge bases which are sharable and reusable. KQML is both a message format and a message-handling protocol to support run-time knowledge sharing among agents. KQML focuses on an extensible set of *performatives*, which defines the permissible "speech acts" agents may use and comprise a substrate on which to develop higher-level models of interagent interaction such as contract nets and negotiation. In addition, KQML provides a basic architecture for knowledge sharing through a special class of agent called *communication facilitators* which coordinate the interactions of other agents The ideas which underlie the evolving design of KQML are currently being explored through experimental prototype systems which are being used to support several testbeds in such areas as concurrent engineering, intelligent design and intelligent planning and scheduling.

1 Introduction

The computational environment which is emerging in such programs as the National Information Infrastructure (NII) is characterized by being highly distributed, heterogeneous, extremely dynamic, and comprising a large number of autonomous nodes. An information system operating in such an environment must handle several emerging problems:

- The predominant architecture on the Internet, the client-server model, is too restrictive. It is difficult for current Internet information services to take the initiative in bringing new, critical material to a user's attention. Some nodes will want to act as both clients and servers, depending on who they are interacting with.
- Several forms of heterogeneity need to be handled, e.g. different platforms, different data formats, the capabilities of different information services, and the implementation technologies employed.
- Many software technologies such as event simulation, applied natural language processing, knowledge–based reasoning, advanced information retrieval, speech processing, etc. have matured to the point of being ready to participate in and contribute to an NII type environment. However, there is a lack of tools and techniques for constructing intelligent clients and servers or for building agent–based software in general.

A community of *intelligent agents* can address each of the problems mentioned above. When we describe these agents as intelligent, we refer to their ability to: communicate with each other using an expressive communication language; work together cooperatively to accomplish complex goals; act on their own initiative; and use local information and knowledge to manage local resources and handle requests from peer agents.

Knowledge Query and Manipulation Language (KQML) is a language that is designed to support interactions among intelligent software agents. It was developed by the ARPA supported Knowledge Sharing Effort [24, 27] and separately implemented by several research groups. It has been successfully used to implement a variety of information systems using different software architectures.

The Knowledge Sharing Effort

The ARPA Knowledge Sharing Effort (KSE) is a consortium to develop conventions facilitating sharing and reuse of knowledge bases and knowledge based systems. Its goal is to define, develop, and test infrastructure and supporting technology to enable participants to build much bigger and more broadly functional systems than could be achieved working alone. The KSE is organized around four working groups each of which addresses a complementary problem identified in current knowledge representation technology: *Interlingua*, *KRSS*, *SRKB*, and *External Interfaces*.

The *Interlingua Group* is developing a common language for expressing the content of a knowledge-base. This group has published a specification document describing the *Knowledge Interchange Formalism* or *KIF* [15] which is based on first order logic with some extensions to support non-monotonic reason and definitions. KIF includes both a specifica-

*This work was supported in part by the Air Force Office of Scientific Research under contract F49620-92-J-0174, and the Advanced Research Projects Agency monitored under USAF contracts F30602-93-C-0177 and F30602-93-C-0028 by Rome Laboratory.

To appear in *The Proceedings of the Third International Conference on Information and Knowledge Management* (CIKM'94), ACM Press, November 1994.

tion of a syntax for the language as well as a specification for the semantics. KIF can be used to support the translation from one content language to another or as a common content language between two agents which use different native representation languages. Information of KIF and associated tools and is available from `http://www.cs.umbc.edu/kse/kif/` .

The *KRSS Group* (Knowledge Representation System Specification) is focussed on defining common constructs within families of representation languages. It has recently finished a common specification for terminological representations in the KL-ONE family. This document and other information on the KRSS group is available as `http://www.cs.umbc.edu/kse/krss/` .

The *SRKB Group* (Shared, Reusable Knowledge Bases) is concerned with facilitating consensus on contents of sharable knowledge bases, with sub-interests in shared knowledge for particular topic areas and in topic-independent development tools and methodologies. It has established a repository for sharable ontologies and tools which is available over the Internet as `http://www.cs.umbc.edu/kse/srkb/`

The scope of the *External Interfaces Group* is the run-time interactions between knowledge based systems and other modules in a run-time environment. Special attention has been given to two important cases – communication between two knowledge-based systems and communication between a knowledge-based system and a conventional database management system [26]. The KQML language is one of the main results which have come out of the external interfaces group of the KSE. General information is available from `http://www.cs.umbc.edu/kqml`.

2 KQML and Intelligent Information Integration

We could address many of the difficulties of communication between intelligent agents described in the Introduction by giving them a common language. In linguistic terms, this means that they would share a common syntax, semantics and pragmatics.

Getting information processes, especially AI processes, to share a common syntax is a major problem. There is no universally accepted language in which to represent information and queries. Languages such as KIF [15], extended SQL, and LOOM [22] have their supporters, but there is also a strong position that it is too early to standardize on any representation language [19]. As a result, it is currently necessary to say that two agents can communicate with each other if they have a common representation language or use languages that are inter-translatable.

Assuming a common or translatable language, it is still necessary for communicating agents to share a framework of knowledge in order to interpret message they exchange. This is not really a shared semantics, but a shared ontology. There is not likely to be one shared ontology, but many. Shared ontologies are under development in many important application domains such as planning and scheduling, biology and medicine.

Pragmatics among computer processes includes 1) knowing who to talk with and how to find them and 2) knowing how to initiate and maintain an exchange. KQML is concerned primarily with pragmatics (and secondarily with semantics). It is a language and a set of protocols that support computer programs in identifying, connecting with and exchanging information with other programs.

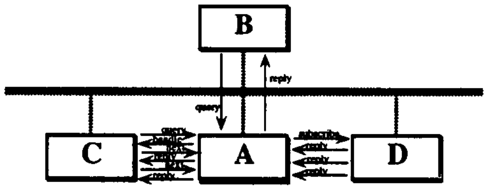


Figure 1: **Several basic communication protocols are supported in KQML.**

Agent Communication Protocols

There are a variety of interprocess information exchange protocols. In the simplest, one agent acts as a client and sends a query to another agent acting as a server and then waits for a reply, as is shown between agents A and B in Figure 1. The server's reply might consist of a single answer or a collection or set of answers. In another common case, shown between agents A and C, the server's reply is not the complete answer but a handle which allows the client to ask for the components of the reply, one at a time. A common example of this exchange occurs when a client queries a relational database or a reasoner which produces a sequence of instantiations in response. Although this exchange requires that the server maintain some internal state, the individual transactions are as before – involving a *synchronous* communication between the agents. A somewhat different case occurs when the client subscribes to a server's output and an indefinite number of *asynchronous* replies arrive at irregular intervals, as between agents A and D in Figure 1. The client does not know when each reply message will be arriving and may be busy performing some other task when they do.

There are other variations of these protocols. Messages might not be addressed to specific hosts, but broadcast to a number of them. The replies, arriving synchronously or asynchronously have to be collated and, optionally, associated with the query that they are replying to.

Facilitators and Mediators

One of the design criteria for KQML was to produce a language that could support a wide variety of interesting agent architectures. Our approach to this is to introduce a small number of KQML performatives which are used by agents to describe the meta-data specifying the information requirements and capabilities and then to introduce a special class of agents called *communication facilitators* [16]. A facilitator is an agent that performs various useful communication services, e.g. maintaining a registry of service names, forwarding messages to named services, routing messages based on content, providing "matchmaking" between information providers and clients, and providing mediation and translation services.

As an example, consider a case where an agent A would like to know the truth of a sentence X, and agent B may have X in its knowledge-base, and a facilitator agent F is available. If A is aware that it is appropriate to send a query about X to B, then it can use a simple *point to point* protocol and send the query directly to B, as in Figure 2. If, however, A is not aware of what agents are available, or which may have X in their knowledge-bases, or how to contact those of whom it is aware, then a variety of approaches can be used. Figure 3 shows an example in which A uses the *subscribe* performative to request that facilitator F monitor for the truth of X. If B subsequently informs F that it believes X to be true, then F can in turn inform A.

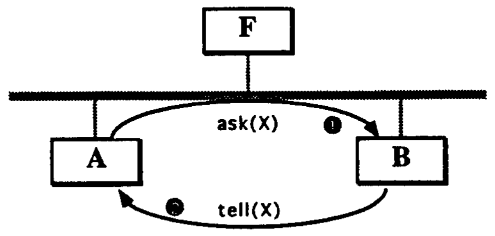


Figure 2: **When A is aware of B and of the appropriateness of querying B about X, a simple point-to-point protocol can be used.**

Figure 4 shows a slightly different situation. A asks F to find an agent that can process an *ask(X)* performative. B independently informs F that it is willing to accept performatives matching *ask(X)*. Once F has both of these messages, it sends B the query, gets a response and forwards it to A.

In Figure 5, A uses a slightly different performative to inform F of its interest in knowing the truth of X. The recruit performative asks the recipient to find an agent that is willing to receive and process an embedded performative. That agent's response is then to be directly sent to the initiating agent. Although the difference between the examples used in Figures 3 and 5 are small for a simple ask query, consider what would happen if the embedded performative was *subscribe(ask-all(X))*.

As a final example, consider the exchange in Figure 6 in which A asks F to "recommend" an agent to whom it would be appropriate to send the performative *ask(X))*. Once F learns that B is willing to accept *ask(X)* performatives, it replies to A with the name of agent B. A is then free to initiate a dialog with B to answer this and similar queries.

From these examples, we can see that one of the main functions of facilitator agents is to help other agents find appropriate clients and servers. The problem of how agents find facilitators in the first place is not strictly an issue for KQML and has a variety of possible solutions.

Current KQML-based applications have used one of two simple techniques. In the PACT project [7], for example, all agents used a central, common facilitator whose location was a parameter initialized when the agents were launched. In the ARPI applications [5], finding and establishing contact with a local facilitator is one of the functions of the KQML API. When each agent starts up, its KQML router module announces itself to the local facilitator so that it is registered in the local database. When the application exits, the router sends another KQML message to the facilitator, removing the application from the facilitator's database. By

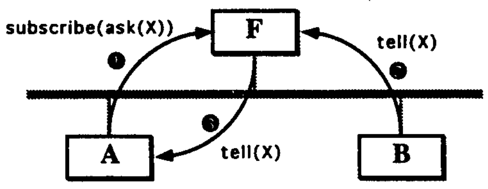


Figure 3: **Agent A can ask facilitator agent F to monitor for changes in its knowledge-base. Facilitators are agents that deal in knowledge about the information services and requirements of other agents and offer such services as forwarding, brokering, recruiting and content-based routing.**

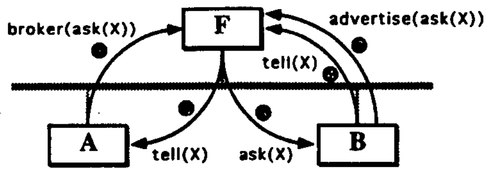


Figure 4: **The broker performative is used to ask a facilitator agent to find another agent which can process a given performative and to receive and forward the reply.**

convention, a facilitator agent should be running on a host machine with the symbolic address *facilitator.domain* and listening to the standard KQML port.

Scaling up to a national-scale information enterprise will require the incorporation of new techniques. The current Internet *Domain Name Servers* (DNS) use a very simple, yet robust technique for mapping symbolic names into internet IP addresses. Similar techniques can be used to map symbolic agent "names" into specific agent references that can be used to contact the agent. What will be involved is the development of a hierarchical "ontology" for organizing information that is orthogonal to the hierarchical scheme used to organize the Internet. Figure 7 shows such an agent which could function as such facilitator-agent-server.

The role of KQML

As a communication language for intelligent information agents, KQML draws on work in both *distributed systems* and *distributed AI* and offers a level of abstraction that should be useful to both.

With respect to distributed software systems in general, KQML provides an abstraction of a process as an information agent as a knowledge-based system (KBS). The KBS model easily subsumes a broad range of commonly used information agent models in use today, including database management systems, hypertext systems, server-oriented software (e.g. finger demons, mail servers, HTML servers, etc), simulations, etc. Such systems can usually be modeled as having two virtual knowledge bases – one representing the agent's information store (i.e., beliefs) and the other representing its intentions (i.e., goals). We hope that future standards for interchange and interoperability languages and protocols will be based on this very powerful and rich model. This will avoid the built-in limitations of more constrained models (e.g., that of a simple remote procedure call or relational database query) and also make it easier to integrate truly intelligent agents with simpler and more mundane information clients and servers. Current KQML implementations have used standard communication and messaging protocols as a transport layer, including TCP/IP, email, Linda, HTTP, and CORBA. As standards in this area evolve and

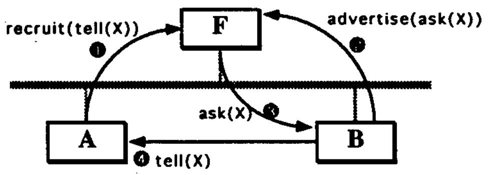


Figure 5: **The recruit performative is used to ask a facilitator agent to find an appropriate agent to which an embedded performative can be forwarded. Any reply is returned directly to the original agent.**

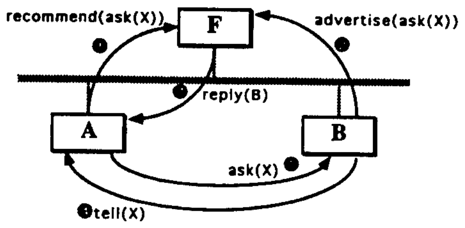


Figure 6: **The recommend performative is used to ask a facilitator agent to respond with the "name" of another agent which is appropriate for sending a particular performative.**

new standards are introduced, we expect that KQML implementations will use them.

The contribution that KQML makes to Distributed AI research is to offer a standard language and protocol that intelligent agents can use to communicate among themselves as well as with other information servers and clients. The independence of the communication and content languages affords a flexibility which is quite useful. In designing KQML, our goal is to build in the primitives necessary to support all of the interesting agent architectures currently in use. If we have been successful, then KQML should serve to be a good tool for DAI research, and, if used widely, should enable greater research collaboration among DAI researchers.

3 The KQML Language

Communication takes place on several levels. The content of the message is only a part of the communication. Being able to locate and engage the attention of someone you want to communicate with is a part of the process. Packaging your message in a way which makes your purpose in communicating clear is another.

When using KQML, a software agent transmits content messages, composed in a language of its own choice, wrapped inside of a KQML message. The content message can be expressed in any representation language and written in either ASCII strings or one of many binary notations (e.g. network independent XDR representations). All KQML implementations ignore the content portion of the message except to the extent that they need to recognize where it begins and ends.

The syntax of KQML is based on a balanced parenthesis list. The initial element of the list is the performative and the remaining elements are the performative's arguments as keyword/value pairs. Because the language is relatively simple, the actual syntax is not significant and can be changed if necessary in the future. The syntax reveals the roots of the initial implementations, which were done in Common Lisp, but has turned out to be quite flexible.

KQML is expected to be supported by an software substrate which makes it possible for agents to locate one another in a distributed environment. Most current implementations come with custom environments of this type; these are commonly based on helper programs called *routers* or *facilitators*. These environments are not a specified part of KQML. They are not standardized and most of the current KQML environments will evolve to use some of the emerging commercial frameworks, such as OMG's CORBA or Microsoft's OLE2, as they become more widely used.

The KQML language supports these implementations by allowing the KQML messages to carry information which is

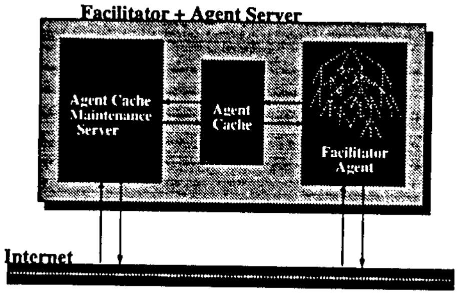


Figure 7: **Some facilitator agents will specialize in knowing how to contact other agents (among other things) and can thus act as "agent-servers".**

useful to them, such as the names and addresses of the sending and receiving agents, a unique message identifier, and notations by any intervening agents. There are also optional features of the KQML language which contain descriptions of the content: its language, the ontology it assumes, and some type of more general description, such as a descriptor naming a topic within the ontology. These optional features make it possible for the supporting environments to analyze, route and deliver messages based on their content, *even though the content itself is inaccessible.*

The forms of these parts of the KQML message may vary, depending on the transport mechanism used to carry the KQML messages. In implementations which use TCP streams as the transport mechanism, they appear as fields in the body of the message. In an earlier version of KQML, these fields were kept in *reserved* locations, in an outer wrapper of the message, to emphasize the difference from other fields. In other transport mechanisms the syntax and content of these message may be different. For example, in the E-mail implementation of KQML, these fields are embedded in KQML mail headers.

The set of performatives forms the core of the language. It determines the kinds of interactions one can have with a KQML-speaking agent. The primary function of the performatives is to identify the protocol to be used to deliver the message and to supply a *speech act* which the sender attaches to the content. The performative signifies that the content is an *assertion*, a *query*, a *command*, or any other mutually agreed upon speech act. It also describes how the sender would like any reply to be delivered, that is, what protocol will be followed.

Conceptually, a KQML message consists of a performative, its associated arguments which include the real content of the message, and a set of optional arguments *transport* which describe the content and perhaps the sender and receiver. For example, a message representing a query about the price of a share of IBM stock might be encoded as:

```
(ask-one
  :content (PRICE IBM ?price)
  :receiver stock-server
  :language LPROLOG
  :ontology NYSE-TICKS)
```

In this message, the KQML performative is *ask-one*, the content is *(price ibm ?price)*, the ontology assumed by the query is identified by the token *nyse-ticks*, the receiver of the message is to be a server identified as *stock-server* and the query is written in a language called *LPROLOG*. A similar query could be conveyed using standard Prolog as the content language in a form that requests the set of all answers as:

```
(ask-all
  :content "price(IBM, [?price, ?time])"
  :receiver stock-server
  :language standard_prolog
  :ontology NYSE-TICKS)
```

The first message asks for a single reply; the second asks for a set as a reply. If we had posed a query which had a large number of replies, would could ask that they each be sent separately, instead of as a single large collection by changing the performative. (To save space, we will no longer repeat fields which are the same as in the above examples.)

```
(stream-all
  ;;?VL is a large set of symbols
  :content (PRICE ?VL ?price))
```

The *stream-all* performative asks that a set of answers be turned into a set of replies. To exert control of this set of reply messages we can wrap another performative around the preceding message.

```
(standby
   :content (stream-all
               :content (PRICE ?VL ?price)))
```

The *standby* performative expects a KQML language content and it requests that the agent receiving the request take the stream of messages and hold them and release them one at a time, each time the sending agent transmits a message with the *next* performative. The exchange of next/reply messages can continue until the stream is depleted or until the sending agent sends either a *discard* message (i.e. discard all remaining replies) or a *rest* message (i.e. send all of the remaining replies now). This combination is so useful that it can be abbreviated:

```
(generate
    :content (PRICE ?VL ?price)))
```

A different set of answers to the same query can be obtained (from a suitable server) with the query:

```
(subscribe
   :content (stream-all
              :content (PRICE IBM ?price)))
```

This performative requests all future changes to the answer to the query, i.e. it is a stream of messages which are generated as the trading price of IBM stock changes. An abbreviation for subscribe/stream combination is known a *monitor*.

```
(monitor
    :content (PRICE IBM ?price)))
```

Though there is a predefined set of reserved performatives, it is neither a minimal required set nor a closed one. A KQML agent may choose to handle only a few (perhaps one or two) performatives. The set is extensible; a community of agents may choose to use additional performatives if they agree on their interpretation and the protocol associated with each. However, an implementation that chooses to implement one of the reserved performatives must implement it in the standard way.

Basic query performatives:

- evaluate, ask-if, ask-in, ask-one, ask-all, ...

Multi-response query performatives:

- stream-in, stream-all, ...

Response performatives:

- reply, sorry, ...

Generic informational performatives:

- tell, achieve, cancel, untell, unachieve, ...

Generator performatives:

- standby, ready, next, rest, discard, generator, ...

Capability-definition performatives:

- advertise, subscribe, monitor, import, export, ...

Networking performatives:

- register, unregister, forward, broadcast, route, ...

Figure 8: **There are about two dozen reserved performative names which fall into seven basic categories.**

Some of the reserved performatives are shown in Figure 8. In addition to standard communication performatives such as ask, tell, deny, delete, and more protocol oriented performatives such as *subscribe*, KQML contains performatives related to the non-protocol aspects of pragmatics, such as *advertise* - which allows an agent to announce what kinds of asynchronous messages it is willing to handle; and *recruit* - which can be used to find suitable agents for particular types of messages. For example, the server in the above example might have earlier announced:

```
(advertise
    :ontology NYSE-TICKS
    :language LPROLOG
    :content (monitor
                :content (PRICE ?x ?y)))
```

Which is roughly equivalent to announcing that it is a stock ticker and inviting monitor requests concerning stock prices. This *advertise* message is what justifies the subscriber's sending the *monitor* message.

4 KQML Software Architectures

KQML was not defined by a single research group for a particular project. It was created by a committee of representatives from different projects, all of which were concerned with managing distributed implementations of systems. One was a distributed collaboration of expert systems in the planning and scheduling domain. Another was concerned with problem decomposition and distribution in the CAD/CAM domain. A common concern was the management of a collection of cooperating processes and the simplification of the programming requirements for implementing a system of this type. However, the groups did not share a common communication architecture. As a result, KQML does not dictate a particular system architecture, and several different systems have evolved.

Our group has two implementations of KQML. One is written in Common Lisp, the other in C. Both are fully interoperable and are frequently used together. The design of these implementations was motivated by the need to integrate a variety of preexisting expert systems into a collaborating group of processes. Most of the systems involved were never designed to operate in a communication oriented

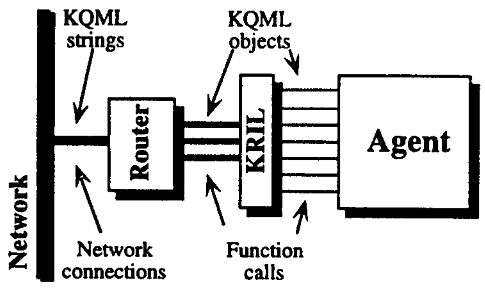


Figure 9: **A router gives an application a single interface to the network, providing both client and server capabilities, managing multiple simultaneous connections, and handling some KQML interactions autonomously. The KRIL is the interface to the agent application and provides internal access points to which the router delivers incoming messages, analyzes outgoing messages for appropriate domain tagging and routing, and provides application specific interface and procedures for communication access.**

environment. The design is built around two specialized programs, a *router* and a *facilitator*, and a library of interface routines, called a *KRIL.*

KQML Routers

Routers are content independent message routers. Each KQML speaking software agent is associated with its own separate router process. All routers are identical; each is just an executing copy of the same program. A router handles all KQML messages going to and from its associated agent. Because each program has an associated router process, it is not necessary to make extensive changes to each program's internal organization to allow it to asynchronously receive messages from a variety of independent sources. The router provides this service for the agent and provides the agent with a single point of contact for the rest of the network. It provides both client and server functions for the application and manages multiple simultaneous connections with other agents.

The router never looks at the content fields of the messages it handles. It relies on the KQML performatives and its arguments. If an outgoing KQML message specifies a particular Internet address, the router directs the message to it. If the message specifies a particular service, the router will attempt to find an Internet address for that service and deliver the message to it. If the message only provides a description of the content (e.g. query, :ontology "geo-domain-3", :language "Prolog", etc.) the router may attempt to find a server which can deal with the message and it will deliver it there, or it may choose to forward it to a smarter communication agent which may be willing to route it. Routers can be implemented with varying degrees of sophistication – they can not guarantee to deliver all messages.

KQML Facilitators

To deliver messages that are incompletely addressed, routers rely on *facilitators.* A facilitator is a network application which provides useful network services. It maintains a registry of service names; it will forward messages on request to named services. It may provide matchmaking services between information providers and consumers. Facilitators are actual network software agents which have their own KQML routers to handle their traffic and deal exclusively in KQML messages. There is typically one facilitator for each local group of agents. This can translate into one facilitator per local site or one per project; there may be multiple local facilitators to provide redundancy. When each application starts up, its router announces itself to the local facilitator so that it is registered in the local database. When the application exits, the router sends another KQML message to the facilitator, removing the application from the facilitator's database. In this way applications can find each other without there having to be a hand maintained list of local services.

KQML KRILs

Since the router is a separate process from the application, it is necessary to have a programming interface between the application and the router. This application program interface (API) is called a KRIL (KQML Router Interface Library). While the router is a separate process, with no understanding of the content field of the KQML message, the KRIL API is embedded in the application and has access to the application's tools for analyzing the content. While there is only one piece of router code, which is instantiated for each process, there can be various KRILs, one for each application type and one for each application language. The general goal of the KRIL is to make access to the router as simple as possible for the programmer.

To this end, a KRIL can be as tightly embedded in the application, or even the application's programming language, as is desirable. For example, an early implementation of KQML featured a KRIL for the Prolog language which had only a simple declarative interface for the programmer. During the operation of the Prolog interpreter, whenever the Prolog database was searched for predicates, the KRIL would intercept the search; determine if the desired predicates were actually being supplied by a remote agent; formulate and pose an appropriate KQML query; and return the replies to the Prolog interpreter as though they were recovered from the internal database. The Prolog program itself contained no mention of the distributed processing going on except for the declaration of which predicates were to be treated as remote predicates.

It is not necessary to completely embed the KRIL in the application's programming language. A simple KRIL generally provides two programmatic entries. For initiating a transaction there is a **send-kqml-message** function. This accepts a message content and as much information about the message and its destination as can be provided and returns either the remote agent's reply (if the message transmission is synchronous and the process blocks until a reply is received) or a simple code signifying the message was sent. For handling incoming asynchronous messages, there is usually a **declare-message-handler** function. This allows the application programmer to declare which functions should be invoked when messages arrive. Depending on the KRILs capabilities, the incoming messages can be sorted according to *performative*, or *topic*, or other features, and routed to different message handling functions.

In addition to these programming interfaces, KRILs accept different types of declarations which allow them to register their application with local facilitators and contact remote agents to advise them that they are interested in receiving data from them. Our group has implemented a variety of experimental KRILs, for Common Lisp, C, Prolog, Mosaic, SQL, and other tools.

5 Experience with KQML

The KQML language and implementations of the protocol have been used in several prototype and demonstration systems. The applications have ranged from concurrent design and engineering of hardware and software systems, military transportation logistics planning and scheduling, flexible architectures for large-scale heterogeneous information systems, agent-based software integration and cooperative information access planning and retrieval. KQML has the potential to significantly enhance the capabilities and functionality of large-scale integration and interoperability efforts now underway in communication and information technology such as the national information infrastructure and OMG's CORBA, as well as in application areas electronic commerce, health information systems and virtual enterprise integration. The content languages used have included languages intended for knowledge exchange including the Knowledge Interchange Format (KIF) and the Knowledge Representation Specification Language (KRSL) [21] as well as other more traditional languages such as SQL. Early experimentations with KQML began in 1990. The following is a representative selection of applications and experiments developed using KQML.

The design and engineering of complex computer systems, whether exclusively hardware or software systems or both, today involves multiple design and engineering disciplines. Many such systems are developed in fast cycle or concurrent processes which involve the immediate and continual consideration of end-product constraints, e.g., marketability, manufacturing planning, etc. Further, the design, engineering and manufacturing components are also likely to be distributed across organizational and company boundaries. KQML has proved highly effective in the integration of diverse tools and systems enabling new tool interactions and supporting a high-level communication infrastructure reducing integration cost as well as flexible communication across multiple networking systems. The use of KQML in these demonstrations has allowed the integrators to focus on what the integration of design and engineering tools can accomplish and appropriately deemphasized how the tools communicate [17, 23, 8, 10].

We have used KQML as the communication language in several technology integration experiments in the ARPA Rome Lab Planning Initiative. One of these experiments supported an integrated planning and scheduling system for military transportation logistics linking a planning agent (in SIPE [30, 4]), with a scheduler (in Common Lisp), a knowledge base (in LOOM [22]), and a case based reasoning tool (in Common Lisp). All of the components integrated were preexisting systems which were not designed to work in a cooperative distributed environment.

In a second experiment, we developed a information agent consisting of CoBASE [6], a cooperative front-end, SIMS [1, 2], an information mediator for planning information access, and LIM [26], an information mediator for translating relational data into knowledge structures. CoBASE processes a query, and, if no responses are found relaxes the query based upon approximation operators and domain semantics and executes the query again. CoBASE generates a single knowledge-based query for SIMS which using knowledge of different information sources selects which of several information sources to access, partitions the query and optimizes access. Each of the resulting queries in this experiment is sent to a LIM knowledge server which answers the query by creating objects from tuples in a relational database. A LIM server front-ends each different database. This experiment was run over the internet involving three, geographically dispersed sites.

Agent-Base Software Integration [18] is an effort underway at Stanford University which applying KQML as an integrating framework for general software systems. Using KQML, a federated architecture incorporating a highly sophisticated facilitator is developed which supports an agent-based view of software integration and interoperation [16]. The facilitator in this architecture is an intelligent agent used to process and reason about the content of KQML messages supporting tighter integration of disparate software systems.

We have also successfully used KQML in other smaller demonstrations integrating distributed clients (in C) with mediators which were retrieving data from distributed databases. Mediators are not just limited distributed database access. In another demonstration, we experimented with a KQML URL for the World Wide Web. The static nature of links within such hypermedia structures lends itself to be extended with virtual and dynamic links to arbitrary information sources as can be supported easily with KQML.

6 Conclusion

This paper has described KQML – a language and associated protocol by which intelligent software agents can communicate to share information and knowledge. We believe that KQML, or something very much like it, will be important in building the distributed agent-oriented information systems of the future.

The design of KQML has continued to evolve as the ideas are explored and feedback is received from the prototypes and the attempts to use them in real testbed situations. Furthermore, new standards for sharing persistent object-oriented structures are being developed and promulgated, such as OMG's CORBA specification and Microsoft's OLE 2.0. Should any of these become widely used, it will be worthwhile to evolve KQML so that its key ideas the collection of reserved performatives, the support for a variety of information exchange protocols, the need for an information based directory service can enhance these new information exchange languages.

Additional information on KQML, including papers, language specifications, access to APIs, information on email discussion lists, etc, can be obtained via the world wide web as `http://www.cs.umbc.edu/kqml/` and via ftp from `ftp.cs.umbc.edu` in pub/kqml/.

References

[1] Yigal Arens. Planning and reformulating queries for semantically-modeled multidatabase systems. In *First International Conference on Information and Knowledge Management*, October 1992.

[2] Yigal Arens, Chin Chee, Chun-Nan Hsu, Hoh In, and Craig A. Knoblock. Query processing in an information mediator. In *Proceedings of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop*, February 1994.

[3] External Interfaces Working Group ARPA Knowledge Sharing Initiative. Specification of the KQML agent-communication language. Working paper. Available as `http://www.cs.umbc.edu/kqml/papers/kqml-spec.ps`, December 1992.

[4] Marie Bienkowski, Marie desJardins, and Roberto Desimone. SOCAP: system for operations crisis action planning. In *Proceedings of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop*, February 1994.

[5] Mark Burstein, editor. *Proceedings of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop.* Morgan Kuafmann Publishers, Inc., February 1994.

[6] Wes Chu and Hua Yang. Cobase: A cooperative query answering system for database systems. In *Proceedings of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop*, February 1994.

[7] M. Cutkosky, E. Engelmore, R. Fikes, T. Gruber, M. Genesereth, and W. Mark. PACT: An experiment in integrating concurrent engineering systems. *IEEE Computer*, pages 28–38, January 1993.

[8] D. Kuokka et. al. Shade: Technology for knowledge-based collaborative. In *AAAI Workshop on AI in Collaborative Design*, 1993.

[9] J. McGuire et. al. Shade: Technology for knowledge-based collaborative engineering. *Journal of Concurrent Engineering: Applications and Research (CERA)*, 1(2), September 1993.

[10] William Mark et. al. Cosmos: A system for supporting design negotiation. *Journal of Concurrent Engineering: Applications and Research (CERA)*, 2(3), 1994.

[11] Tim Finin, Rich Fritzson, and Don McKay. A high-level language and protocol to support intelligent agent interoperability. In *Workshop on Enabling Technologies for Concurrent Engineering*, April 1992.

[12] Tim Finin, Rich Fritzson, and Don McKay. A knowledge query and manipulation language for intelligent agent interoperability. In *Fourth National Symposium on Concurrent Engineering, CE & CALS Conference*, June 1–4 1992. Available as `http://www.cs.umbc.edu/kqml/papers/cecals.ps`.

[13] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In *International Conference on Building and Sharing of Very Large-Scale Knowledge Bases*, December 1993. A version of this paper will appear in Kazuhiro Fuchi and Toshio Yokoi (Ed.), "Knowledge Building and Knowledge Sharing", Ohmsha and IOS Press, 1994. Available as `http://www.cs.umbc.edu/kqml/papers/kbks.ps`.

[14] Tim Finin, Charles Nicholas, and Yelena Yesha, editors. *Information and Knowledge Management, Expanding the Definition of Database.* Lecture Notes in Computer Science 752. Springer-Verlag, 1993. (ISBN 3-540-57419-0).

[15] M. Genesereth and R. Fikes et. al. Knowledge interchange format, version 3.0 reference manual. Technical report, Computer Science Department, Stanford University, 1992.

[16] Michael R. Genesereth and Steven P. Katchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 147, 1994.

[17] Mike Genesereth. Designworld. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 2,785–2,788. IEEE CS Press.

[18] Mike Genesereth. An agent-based approach to software interoperability. Technical Report Logic-91-6, Logic Group, CSD, Stanford University, February 1993.

[19] Matt Ginsberg. Knowledge interchange format: The KIF of death. *AI Magazine*, 1991.

[20] Yannis Labrou and Tim Finin. A semantics approach for KQML – a general purpose communication language for software agents. In *Third International Conference on Information and Knowledge Management*, November 1994. Available as `http://www.cs.umbc.edu/kqml/-papers/kqml-semantics.ps`.

[21] Nancy Lehrer. The knowledge representation specification language manual. Technical report, ISX Corporation, Thousand Oaks, California, 1994.

[22] Robert MacGregor and Raymond Bates. The LOOM knowledge representation language. Technical Report ISI/RS-87-188, USC/ISI, 1987. Also appears in *Proceedings of the Knowledge-Based Systems Workshop* held in St. Louis, Missouri, April 21–23, 1987.

[23] M.Tenenbaum, J. Weber, and T. Gruber. Enterprise integration: Lessons from shade and pact. In C. Petrie, editor, *Enterprise Integration Modeling.* MIT Press, 1993.

[24] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[25] Jeff Y-C Pan and Jay M. Tenenbaum. An intelligent agent framework for enterprise integration. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6), December 1991. (Special Issue on Distributed AI).

[26] Jon Pastor, Don McKay, and Tim Finin. View-concepts: Knowledge-based access to databases. In *First International Conference on Information and Knowledge Management*, October 1992.

[27] R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA knowledge sharing effort: Progress report. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, November 1992. Available as `http://www.cs.umbc.edu/kqml/-papers/kr92.ps`.

[28] R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA knowledge sharing effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, San Mateo, CA, November 1992. Morgan Kaufmann.

[29] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 33(11):89–99, November 1992.

[30] David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc., San Mateo, CA., 1988.

3. PROPOSED NEW KQML SPECIFICATION

Intelligent Agent Integration Technology

Prepared by:
Yannis Labrou
and
Tim Finin
Co-Principal Investigator
University of Maryland - Baltimore County
finin@cs.umbc.edu, (410)455-3522

This section is also available as Technical Report TR CS-97-03 from UMBC.

This document constitutes a proposal for a revision of the current KQML specification document ([1]). Although the differences regarding the syntax of KQML messages and the reserved performative parameters are minimal, there are significant changes regarding the set of reserved performatives, their meaning and intended use. Parts of Sections 1 and 2 appear in the current KQML specification document ([1]) and are included here for reasons of completeness of this presentation.

1 KQML transport assumptions

This chapter presumes a model of message transport. So for these purposes, we define the following abstraction of the transport level:

- Agents are connected by unidirectional communication links that carry discrete messages.
- These links may have a non–zero message transport delay associated with them.
- When an agent receives a message, it knows from which incoming link the message arrived.
- When an agent sends a message it may direct the message to a particular outgoing link.
- Messages to a single destination arrive in the order they were sent.
- Message delivery is reliable.

  NOTE: The latter property is less useful than it may appear, unless there is a guarantee of *agent reliability* as well. Such a guarantee is a policy issue, and may vary among systems but it is important (as an assumption) for the semantic description presented in [3]

This abstraction may be implemented in many ways. For example, the links could be TCP/IP connections over the Internet, which may only actually exist during the transmission of a single message or groups of messages. The links could be email paths used by mail–enabled programs. The links could be UNIX IPC connections among processes running on an ether–networked LAN. Or, the links could be high–speed switches in a multiprocessor machine like the Hypercube, accessed via Object Request Broker software. Regardless of how communication is actually carried out, KQML assumes that at the level of agents, the communication appears to be point–to–point message passing.

The point of this point–to–point message transport abstraction is to provide a simple, uniform model of communication for the outer layers of agent–based programs. This should make agent–based programs and APIs easier to design and build.

2 KQML string syntax

A KQML message is also called a *performative.* A performative is expressed as an ASCII string using the syntax defined in this section. This syntax is a restriction on the ASCII representation of Common Lisp Polish-prefix notation. The ASCII-string LISP list notation has the advantages of being readable by humans, simple for programs to parse (particularly for many knowledge–based programs), and transportable by many inter–application messaging platforms. However, no choice of message syntax will be both convenient and efficient for all messaging APIs.

Unlike Lisp function invocations, parameters in performatives are indexed by keywords and are therefore order independent. These keywords, called *parameter names*, must begin with a colon (:) and must precede the corresponding *parameter value.* Performative parameters are identified by keywords rather than by their position due to a large number of optional parameters to performatives. Several examples of the syntax appear throughout this document.

The KQML string syntax in BNF is shown in Figure 1. The BNF assumes definitions for **`<ascii>`**, **`<alphabetic>`**, **`<numeric>`**, **`<double-quote>`**, **`<backslash>`**, and **`<whitespace>`**. "*" means any number of occurrences, and "-" indicates set difference. Note that **`<performative>`** is a specialization of **`<expression>`**. In length–delimited strings, *e.g.*, "#3"abc", the whole number before the double–quote specifies the length of the string after the double–quote.

```
<performative>::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)
<expression>  ::= <word> | <quotation> | <string> |
                   (<word> {<whitespace> <expression>}*)
<word>        ::= <character><character>*
<character>   ::= <alphabetic> | <numeric> | <special>
<special>     ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
                  @ | $ | % | : | . | ! | ?
<quotation>   ::= '<expr> | `<comma-expr>
<comma-expr>  ::= <word> | <quotation> | <string> | ,<comma-expr> |
                   (<word> {<whitespace> <comma-expr>}*)
<string>      ::= "<stringchar>*" | #<digit><digit>*"<ascii>*
<stringchar>  ::= \<ascii> | <ascii>-\-<double-quote>
```

Figure 1: KQML string syntax in BNF

3 Reserved performative parameters

As described in Section 2, performatives take parameters identified by keywords. This section defines the meaning of some common performative parameters, by coining their keywords and describing the meaning of the accompanying values. This will facilitate brevity in the performative definitions presented in Section 4, since those parameters are used heavily.

The following parameters are *reserved* in the sense that any performative's use of parameters with those keywords must be consistent with the definitions below. These keywords and information parameter meanings are summarized in Table 1. The specification of reserved parameter keywords is useful in at least two ways: 1) to mandate some degree of uniformity on the semantics of common parameters, and thereby reduce programmer confusion, and 2) to support some level of understanding, by programs, of performatives with unknown names but with known parameter keywords.

```
:sender <word>
:receiver <word>
```

These parameters convey the actual sender and receiver of a performative, as opposed to the virtual sender and receiver in the `:from` and `:to` parameters of a *forward* performative (see Section 4.3).

```
:reply-with  <word>
:in-reply-to <word>
```

The sender knows that the *reply* (meaning the *response* or *follow-up*, in a more general sense, that is "related" or "linked" to the message), if any, will have a `:in-reply-to` parameter with a value identical to the `<word>` of the `:reply-with` parameter of the message to which it is responding.

```
:language <word>
:ontology <word>
:content <expression>
```

Keyword	*Meaning*
`:sender`	the actual sender of the performative
`:receiver`	the actual receiver of the performative
`:from`	the origin of the performative in `:content` when *forward* is used
`:to`	the final destination of the performative in `:content` when *forward* is used
`:in-reply-to`	the expected label in a *response* to a previous message (same as the `:reply-with` value of the previous message)
`:reply-with`	the expected label in a *response* to the current message
`:language`	the name of the representation language of the `:content`
`:ontology`	the name of the ontology (*e.g.*, set of term definitions) assumed in the `:content` parameter
`:content`	the information about which the performative expresses an attitude

Table 1: Summary of reserved parameter keywords and their meanings.

The `:content` parameter indicates the "direct object" (in the linguistic sense) of the performative. For example, if the performative name is `tell` then the `:content` will be the sentence being "told". The `<expression>` in the `:content` parameter must be a valid expression in the representation language specified by the `:language` parameter (or KQML in some cases). Figure 1 suggests that expressions in the `:content`, that have parentheses (like the Prolog expressions that appear in the examples throughout this chapter) should be enclosed in `<double-quote>`s (" "). Furthermore, the constants used in the `<expression>` must be a subset of those defined by the ontology named by the `:ontology` parameter.

> NOTE: The BNF suggests that both `:language` and `:ontology` are restricted to only take `<word>`s as values, and therefore complex terms, *e.g.*, denoting unions of ontologies, are not allowed. The definitions for `<quotation>` and `<comma-expr>` in Figure 1, are intended to accommodate expressions in KIF that use special operators.

4 The reserved performatives

We provide descriptions of the **reserved** performatives and examples that show their proper use. We use the following notation:

- When referred to in text, performative names are written in italics, *e.g.*, *ask-all*, *tell*, *etc.*
- In text, we use the names of reserved performative parameters to refer to their values, so `:sender` refers to the particular sender of a performative, `:content` refers to the content and so on.
- Occasionally, we use $\texttt{parameter}_{performative}$ to refer to the value of a particular performative parameter, *i.e.*, $\texttt{sender}_{advertise}$ to refer to the sender of an advertise in a particular case.
- We use `<performative>` to refer to a particular instance of a performative.

The performatives examined in this document are organized in three (3) categories and their meaning and some properties of interest can be found in Table 2 (page 7), Table 3 (page 8), Table 4 (page 9) and Table 5 (page 10). The parameters presented with the *performatives'* specifications are mandatory and define the minimum for proper use of the *performative.* Parameters preceded by an asterisk (*) are not always mandatory. For example, the `:in-reply-to` for *ask-if* is mandatory if the *ask-if* follows a relevant *advertise*, but not in other cases. The asterisk itself is not part of the KQML syntax; we only use it as a meta–syntactic marker. Finally, although often some of the values of the parameters can be inferred, we choose completeness over economy.

4.1 Discourse performatives

This is the category of performatives that may be considered as close as possible to speech acts in the linguistic sense. Of course the idea of explicitly stating the format of the response (as in *stream-all* or *ask-one*) is unusual from a speech act theory perspective, but they may still be considered as speech acts in the pure sense. These are the performatives to be used in the context of an information and knowledge exchange kind of discourse between two agents.

```
(ask-if
        :sender        <word>
        :receiver      <word>
*       :in-reply-to   <word>
        :reply-with    <word>
        :language      <word>
        :ontology      <word>
        :content       <expression>)
```

Agent A sends the following performative to agent B. The `:in-reply-to` suggests that the *ask-all* follows a relevant *advertise* message.

```
(ask-all
                :sender         A
                :receiver       B
                :in-reply-to    id0
                :reply-with     id1
                :language       Prolog
                :ontology       foo
                :content        "bar(X,Y)")
```

and agent B replies with the following KQML message

```
(tell           :sender         B
                :receiver       A
                :in-reply-to    id1
                :reply-with     id2
                :language       Prolog
                :ontology       foo
                :content        "[bar(a,b),bar(c,d)]")
```

Figure 2: An *ask-all* performative and the appropriate response.

The `:sender` wishes to know if the `:content` is true of the receiver. *True* of the `:receiver` is taken to mean that either the `<expression>` matches a sentence in the receiver's Knowledge Base (KB) or is provable of the `:receiver`, *i.e.*, matches a sentence in the receiver's Virtual Knowledge Base (VKB).[1]

```
(ask-all
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

The `:sender` wishes to know all *instantiations* of the `:content` that are true of the `:receiver`; `<expression>` has free variables that are bound to values in the *instantiations* of the *response*. Those instantiations will be delivered in the form of a collection provided by `:language`. Of course, the notion of the collection is language dependent. In the example in Figure 2 (`:language` is *Prolog*) such a collection is just a *list*.

```
(ask-one
```

[1]From now on we will use "VKB" to refer to either "exists in the KB" or "provable."

Name	*Page*	Meaning
ask-if	6	S wants to know if the **:content** is in R's VKB
ask-all	6	S wants all of R's instantiations of the **:content** that are true of R
ask-one	11	S wants one of R's instantiations of the **:content** that is true of R
stream-all	11	multiple-response version of **ask-all**
eos	11	the end-of-stream marker to a multiple-response (**stream-all**)
tell	13	the sentence is in S's VKB
untell	13	the sentence is **not** in S's VKB
deny	13	the **negation** of the sentence is in S's VKB
insert	14	S asks R to add the **:content** to its VKB
uninsert	14	S wants R to reverse the act of a previous **insert**
delete-one	16	S wants R to remove one matching sentence from its VKB
delete-all	16	S wants R to remove all matching sentences from its VKB
undelete	16	S wants R to reverse the act of a previous **delete**
achieve	17	S wants R to do make something true of its physical environment
unachieve	17	S wants R to reverse the act of a previous **achieve**
advertise	19	S wants R to know that S can and will process a message like the one in **:content**
unadvertise	21	S wants R to know that S cancels a previous **advertise** and will not process any more messages like the one in the **:content**
subscribe	21	S wants updates to R's response to a performative
error	22	S considers R's earlier message to be mal-formed
sorry	24	S understands R's message but cannot provide a more informative response
standby	24	S wants R to announce its readiness to provide a response to the message in **:content**
ready	25	S is ready to respond to a message previously received from R
next	25	S wants R's next response to a message previously sent by S
rest	25	S wants R's remaining responses to a message previously sent by S
discard	29	S does not want R's remaining responses to a previous (multi-response) message
register	30	S announces to R its presence and symbolic name
unregister	30	S wants R to reverse the act of a previous **register**
forward	31	S wants R to forward the message to the **:to** agent (R might be that agent)
broadcast	32	S wants R to send a message to all agents that R knows of
transport-address	30	S associates its symbolic name with a new transport address
broker-one	35	S wants R to find one response to a **<performative>** (some agent other than R is going to provide that response)
broker-all	35	S wants R to find all responses to a **<performative>** (some agent other than R is going to provide that response)
recommend-one	37	S wants to learn of an agent who may respond to a **<performative>**
recommend-all	37	S wants to learn of all agents who may respond to a **<performative>**
recruit-one	37	S wants R to get one suitable agent to respond to a **<performative>**
recruit-all	39	S wants R to get all suitable agents to respond to a **<performative>**

Table 2: Summary of reserved performatives for `:sender` S and `:receiver` R.

Category	*Name*	*Response Required*	*Response Only*	*No Response*	`:content`
Discourse	ask-if	X			`<expression>`
	ask-all	X			`<expression>`
	ask-one	X			`<expression>`
	stream-all	X			`<expression>`
	eos		X		empty
	tell		X		`<expression>`
	untell		X		`<expression>`
	deny		X		`<expression>`
	insert			X	`<expression>`
	uninsert			X	`<expression>`
	delete-one			X	`<expression>`
	delete-all			X	`<expression>`
	undelete			X	`<expression>`
	achieve			X	`<expression>`
	unachieve			X	`<expression>`
	advertise			X	`<performative>`
	unadvertise			X	`<performative>`
	subscribe	X			`<performative>`
Intervention and Mechanics	error		X		empty
	sorry		X		empty
	standby	n/a	n/a	n/a	`<performative>`
	ready	n/a	n/a	n/a	empty
	next	n/a	n/a	n/a	empty
	rest	n/a	n/a	n/a	empty
	discard	n/a	n/a	n/a	empty
Facilitation and Networking	register			X	`<expression>`
	unregister			X	empty
	forward			`:content`	`<performative>`
	broadcast			`:content`	`<performative>`
	transport-address			X	`<expression>`
	broker-one			`:content`	`<performative>`
	broker-all			`:content`	`<performative>`
	recommend-one	X			`<performative>`
	recommend-all	X			`<performative>`
	recruit-one			`:content`	`<performative>`
	recruit-all			`:content`	`<performative>`

Table 3: This is the set of performatives discussed in this document and their properties when used in conversations. The properties have the following meaning: "response required" means that the `:receiver` processes the performative and generates the response on its own; "response only" means that the performative can only be used in the context of responding to some other performative; "no response" means that those performatives **do not** require (but might allow) a response (there is also the possibility of a follow-up message); and `:content` refers to the type of the `:content` ("n/a" stands for not applicable; see Section 4.2 for an explanation). *Forward*, *broadcast*, *broker-one*, *broker-all*, *recruit-one* and *recruit-all*, do not require a response *by default*. Whether there is a response or a follow-up to them, depends solely on the `:content`, *i.e.*, on the `<performative>` that appears in the `:content` and its properties in conversations.

Category	*Name*	*advertise*	*subscribe*	*standby*	*forward broadcast*	Facilitation *performatives*
Discourse	ask-if	X	X		X	X
	ask-all	X	X		X	X
	ask-one	X	X	X	X	X
	stream-all	X	X	X	X	X
	eos			X	X	
	tell			X	X	
	untell				X	
	deny				X	
	insert	X			X	X
	uninsert				X	
	delete-one	X			X	X
	delete-all	X			X	X
	undelete				X	
	achieve	X			X	X
	unachieve				X	
	advertise				X	
	unadvertise				X	
	subscribe	X		X	X	X
Intervention and Mechanics	error				X	
	sorry				X	
	standby				X	
	ready				X	
	next				X	
	rest				X	
	discard				X	
Facilitation and Networking	register					
	unregister					
	forward					
	broadcast					
	transport-address					
	broker-one				X	
	broker-all				X	
	recommend-one		X	X	X	
	recommend-all		X	X	X	
	recruit-one				X	
	recruit-all				X	

Table 4: *Advertise, subscribe, standby, forward, broadcast* and the *facilitation performatives* are the only performatives that may have a `<performative>`, *i.e.*, a KQML message, as `:content` ("facilitation *performatives*" refers to *broker-one, broker-all, recruit-one, recruit-all, recommend-one* and *recommend-all*). Note that the facilitation performatives allow exactly the same performatives as *advertise*, which makes sense since the processing of the facilitation performatives depends on advertisements. The facilitation performatives may appear in the `:content` of *advertise* messages if and only if a non-facilitator is the `:sender` of the *advertise.*

Category	*Name*	*All agents*	*Facilitators only*	*Only if advertised*
Discourse	ask-if	X		
	ask-all	X		
	ask-one	X		
	stream-all	X		
	eos	X		
	tell	X		
	untell	X		
	deny	X		
	insert	X		
	uninsert	X		
	delete-one	X		
	delete-all	X		
	undelete	X		
	achieve	X		
	unachieve	X		
	advertise	X		
	unadvertise	X		
	subscribe	X		
Intervention and Mechanics	error	X		
	sorry	X		
	standby	X		
	ready	X		
	next	X		
	rest	X		
	discard	X		
Facilitation and Networking	register		X	
	unregister		X	
	forward	X		
	broadcast	X		
	transport-address		X	
	broker-one		X	X
	broker-all		X	X
	recommend-one		X	X
	recommend-all		X	X
	recruit-one		X	X
	recruit-all		X	X

Table 5: This table lists the performatives that various kinds of agents may process. We distinguish between agents that are *facilitators* and agents that are not *facilitators*. The categories have the following meaning: "all agents" refers to all agents, whether they serve as facilitators on not; "facilitators only" only applies to agents that are facilitators; and "only if advertised" refers to non-facilitator agents that have to *advertise* for the specific `<performative>`. A subtle distinction has to be drawn between an agent's ability to process a *performative* in principle and to process a `<performative>`, *i.e.*, a KQML message of that *performative* with a particular `:content`. So, for example, although all agents can process *ask-if*, *i.e.*, they have *handler functions* for that performative, they still have to *advertise* their ability to process an *ask-if* with a particular `:content`.

```
          :sender        <word>
          :receiver      <word>
*         :in-reply-to   <word>
          :reply-with    <word>
          :language      <word>
          :ontology      <word>
          :content       <expression>)
```

This performative is the same as *ask-all* but only one expression is sought as a response. Any of the *tell* performatives of Figure 3 would constitute the appropriate response to an *ask-one* message similar to the *ask-all* message of Figure 2.

> NOTE: The `:sender` of an *ask-one* has no control over which of the possible responses might be delivered to it (first, last, random, *etc.*)

```
(stream-all
          :sender        <word>
          :receiver      <word>
          :in-reply-to   <word>
          :reply-with    <word>
          :language      <word>
          :ontology      <word>
          :content       <expression>)
```

This performative's meaning is identical to that of *ask-all*, except for the format of the delivery of the response. Instead of delivering the collection of matches in a single performative, a series of performatives, one for each member of the collection, should be sent. This only holds of course, if the response to the corresponding *ask-all* would have been a *tell*. See Figure 3 for an example of an exchange that involves the *stream-all* performative and note that the collective response is equivalent to that of Figure 2.

```
(eos
          :sender        <word>
          :receiver      <word>
          :in-reply-to   <word>
          :reply-with    <word>)
```

This performative only serves the purpose of marking the end–of–stream of the multi–response to a *stream-all* (see Figure 3).

Agent A sends a message to agent B

```
(stream-all
                :sender       A
                :receiver     B
                :in-reply-to  id0
                :reply-with   id1
                :language     Prolog
                :ontology     foo
                :content      "bar(X,Y)")
```

and agent B replies with the following KQML message

```
(tell           :sender       B
                :receiver     A
                :in-reply-to  id1
                :reply-with   id2
                :language     Prolog
                :ontology     foo
                :content      "bar(a,b)")
```

and later agent B sends

```
(tell           :sender       B
                :receiver     A
                :in-reply-to  id1
                :reply-with   id3
                :language     Prolog
                :ontology     foo
                :content      "bar(c,d)")
```

and finally concludes the response with

```
(eos            :sender       B
                :receiver     A
                :in-reply-to  id1
                :reply-with   id4)
```

Note that B's response is equivalent to B's single performative response to the similar *ask-all* of Figure 2.

Figure 3: A *stream-all* performative and the appropriate responses.

```
(tell
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

This performative indicates that the `:content` expression is true of the `:sender`, *i.e.*, that `:expression` is in its VKB.

```
(untell
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

This performative indicates that the `:content` expression in not true of the sender, *i.e.*, it is not part of the sender's VKB. This does not necessarily mean that the expression's negation is true of the sender. In other words, $\mathit{untell}_{\texttt{<expression>}}$ is *not* the same as $\mathit{tell}_{\neg\texttt{<expression>}}$.

```
(deny
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

This performative indicates that the **negation** of the `:content` is true of the sender, *i.e.*, it is in the sender's VKB. In other words, $\mathit{deny}_{\texttt{<expression>}}$ is the same as $\mathit{tell}_{\neg\texttt{<expression>}}$.

> NOTE: The reason for having such a performative is that a system might not provide for *logical negation* in `:language` but still operate under a Closed World Assumption (CWA), *i.e.*, non-provability of an `<expression>` is equivalent to provability of its negation.

```
(insert
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

The `:sender` requests the `:receiver` to add the `:content` to its KB (see Figure 4).

```
(uninsert
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

This performative is a request to reverse an *insert* that took place in the past by deleting the inserted expression.

> NOTE: Performatives like *insert* and *delete* can only be used when an agent has *advertised* that is going to accept them. Such an *advertisement* implies the acceptance of the corresponding *uninsert* and *undelete* messages. Although it is tempting to view *insert* and *delete* as complementary and use *delete* in the place of *uninsert*, and *insert* instead of *undelete*, we choose having performatives of the *un-* variety, because: (a) an agent might *advertise* only an *insert* or only a *delete* for a particular `:content`, and (b) to emphasize that the intent of the *un-* performative is to reverse an action that has taken place rather than negate its effects. An *uninsert* can only be used after a corresponding *insert*.

An example that involves *insert* and *uninsert* can be seen in Figure 4.

```
(delete-one
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

Agent A sends the following performative to agent B

```
(advertise
              :sender       A
              :receiver     B
              :reply-with   id1
              :language     KQML
              :ontology     kqml-ontology
              :content      (insert
                                        :sender       B
                                        :receiver     A
                                        :in-reply-to  id1
                                        :language     Prolog
                                        :ontology     foo
                                        :content      "bar(X,Y)" ))
```

Later B sends the following message to A, making use of the *advertise*

```
(insert
              :sender       B
              :receiver     A
              :in-reply-to  id1
              :reply-with   id2
              :language     Prolog
              :ontology     foo
              :content      "bar(a,b)" )
```

and some time later B sends the following message to A

```
(uninsert
              :sender       B
              :receiver     A
              :in-reply-to  id1
              :reply-with   id3
              :language     Prolog
              :ontology     foo
              :content      "bar(a,b)" )
```

which is followed a bit later by

```
(insert
              :sender       B
              :receiver     A
              :in-reply-to  id1
              :reply-with   id4
              :language     Prolog
              :ontology     foo
              :content      "bar(c,d)" )
```

Figure 4: An *insert* performative following a related *advertise*, and an example of a proper *uninsert*. Note that $\texttt{reply}-\texttt{with}_{insert}$ is not preset by the `:sender` of the *advertise*.

This performative is a request to delete one sentence from the receiver's KB. The sentence to be deleted is the one that would have been the `:content` of the response if an identical *ask-one* KQML message had been sent and a *tell* performative had been used in the response.

> NOTE: Had the response to the corresponding *ask-one* been anything other than a *tell*, a *sorry* should be the response to a *delete-one*. The idea is that in such a case, *e.g.*, had a *deny* been the response to the *ask-all*, the `:content` of the deny would not appear in the KB, and thus cannot be removed from it.

```
(delete-all
        :sender        <word>
        :receiver      <word>
*       :in-reply-to   <word>
        :reply-with    <word>
        :language      <word>
        :ontology      <word>
        :content       <expression>)
```

This performative is a request to delete all sentences from the receiver's KB that would have constituted the response if an identical *ask-all* KQML message had been sent and a *tell* performative had been used for the response.

```
(undelete
        :sender        <word>
        :receiver      <word>
        :in-reply-to   <word>
        :reply-with    <word>
        :language      <word>
        :ontology      <word>
        :content       <expression>)
```

This performative is a request to reverse a *delete* that took place in the past by inserting the deleted expression(s).

> NOTE: An *undelete* can only be used after a corresponding *delete-one* or *delete-all*. In either case, it *undeletes* whatever was *deleted* in the first place, assuming of course that the original *delete* action was executed successfully (no *error* or *sorry* was sent as a response).

```
(achieve
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

The `:receiver` is asked to want to try to make the `:content` true of the system. Of course this can always be done by just *inserting* the `:content` in the KB, but this performative makes sense when the `:receiver` has a representation of the real world in its KB and the result of the attempt to "make the `:content` true" will be some action in the real world the effect of which will be to modify the respective part of the representation of the real world and thus make the `:content` true in the KB. In other words, the `:content` can be made true only as a result of some action outside of the system, in the physical world. See Figure 5 for an example of an exchange that involves the *achieve* performative.

```
(unachieve
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

This performative is a request to reverse an *achieve* that took place in the past. See Figure 5 for an example of an exchange that involves the *unachieve* performative.

NOTE: An *unachieve* can only be used after a corresponding *achieve.*

```
(advertise
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                    :sender         <word>
                                    :receiver       <word>
                                    :in-reply-to    <word>
```

Agent A sends the following performative to agent B (the `:in-reply-to` value suggests that B has sent an *advertise* for such an *achieve* message), requesting that B set a new value for the motor torque of `motor1`

```
(achieve     :sender        A
             :receiver      B
             :in-reply-to   id1
             :reply-with    id2
             :language      Prolog
             :ontology      motors
             :content       "torque(motor1,5)" )
```

After achieving the requested motor torque (assuming that it was not already set to 5), agent B sends the following message to A (although this is not required)

```
(tell        :sender        B
             :receiver      A
             :in-reply-to   id2
             :reply-with    id3
             :language      Prolog
             :ontology      motors
             :content       "torque(motor1,5)" )
```

Some time later, agent A sends the following message to B, in effect requesting that the previous setting (unknown to A) be achieved

```
(unachieve   :sender        A
             :receiver      B
             :in-reply-to   id1
             :reply-with    id4
             :language      Prolog
             :ontology      motors
             :content       "torque(motor1,5)" )
```

Agent A responds with the following message that serves as acknowledgment (although this is not required), which implies that the motor torque for `motor1` has been sent to its previous value (as a result of the *unachieve*)

```
(untell      :sender        B
             :receiver      A
             :in-reply-to   id4
             :reply-with    id5
             :language      Prolog
             :ontology      motors
             :content       "torque(motor1,5)")
```

A could choose to send a *tell* instead, in which case A would give information to B about the original value (before the *achieve*) of the motor torque of `motor1`.

Figure 5: An *achieve* performative and the appropriate response, later followed by an *unachieve* request.

```
                                    :language       <word>
                                    :ontology       <word>
                                    :content        <expression> ))
```

This performative indicates that the `:sender` commits to process the whole embedded message if the $\texttt{sender}_{advertise}$ receives it (presumably from $\texttt{receiver}_{advertise}$ in the future). The subsequent KQML message ought to be identical to whatever the $\texttt{content}_{advertise}$ is, except for the `:reply-with` value that is going to be set by the `:receiver` of the *advertise*. There are constraints that apply to such a message:

- `performative_name` can be one of `ask-if`, `ask-one`, `ask-all`, `stream-all`, `insert`, `delete-one`, `delete-all`, `achieve` and `subscribe` (or one of the *facilitation performatives* if the `:sender` is not a facilitator; see also Table 4).
- $\texttt{sender}_{advertise} = \texttt{receiver}_{performative\_name}$
- $\texttt{receiver}_{advertise} = \texttt{sender}_{performative\_name}$
- $\texttt{reply} - \texttt{with}_{advertise} = \texttt{in} - \texttt{reply} - \texttt{to}_{performative\_name}$

See Figure 6 for an example of an exchange that involves the *advertise* performative.

> NOTE: Advertising to a facilitator is like advertising, *i.e.*, potentially sending an *advertise*, to all agents that the facilitator knows (or might learn) about. So, when an agent sends an *advertise* to a facilitator, the agent will process messages like the $\texttt{content}_{advertise}$ from *any* agent and not only from $\texttt{receiver}_{advertise}$. For all practical purposes, an *advertise* to a *facilitator* is an *advertise* to the community. Since in order for the $\texttt{sender}_{advertise}$ to process such a message, the proper value for the $\texttt{in} - \texttt{reply} - \texttt{to}_{performative\_name}$ is needed, the $\texttt{sender}_{advertise}$ can rest assured that such knowledge was acquired only through the facilitator that was the $\texttt{receiver}_{advertise}$.

```
(unadvertise
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                    :sender         <word>
                                    :receiver       <word>
                                    :in-reply-to    <word>
                                    :language       <word>
                                    :ontology       <word>
                                    :content        <expression> ))
```

Agent A sends the following performative to agent B

```
(advertise
              :sender        A
              :receiver      B
              :reply-with    id1
              :language      KQML
              :ontology      kqml-ontology
              :content       (ask-if
                                       :sender        B
                                       :receiver      A
                                       :in-reply-to   id1
                                       :language      Prolog
                                       :ontology      foo
                                       :content       "bar(X,Y)" ))
```

Later B sends the following message to A, making use of the *advertise*

```
(ask-if
              :sender        B
              :receiver      A
              :in-reply-to   id1
              :reply-with    id2
              :language      Prolog
              :ontology      foo
              :content       "bar(X,Y)" )
```

and agent A responds accordingly, as committed to do

```
(tell
              :sender        A
              :receiver      B
              :in-reply-to   id2
              :reply-with    id3
              :language      Prolog
              :ontology      foo
              :content       "bar(X,Y)" )
```

At some later time, B sends another *ask-if* message, with a new $\mathtt{reply-with}_{ask-if}$ this time, and agent A will respond promptly again.

Figure 6: An example of an *advertise* and appropriate follow-ups to that.

This performative essentially cancels an *advertise*. Its `:content` has to be the same with the `:content` of the *advertise* that it cancels.

```
(subscribe
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                    :sender         <word>
                                    :receiver       <word>
                                    :in-reply-to    <word>
                                    :language       <word>
                                    :ontology       <word>
                                    :content        <expression> ))
```

This performative is a request to be updated every time that the would-be response to the message in `:content` is different than the last response delivered to the $\texttt{sender}_{subscribe}$. Additionally, since a point of reference is needed for the `receiver` of a *subscribe*, it should issue the first response immediately after receiving the performative and then store the last response in order to compare. We do not need something like an *unsubscribe* performative because a *subscribe* does not affect the VKB, so there is nothing to be undone. If an agent has lost interest to the responses to a prior *subscribe*, a *discard* (see page 29) may be used to inform the other agent. See Figure 8 for an example of an exchange that involves the *subscribe* performative.

NOTE: The `performative_name` in the $\texttt{content}_{subscribe}$ might be any of the performatives that require a response (see Table 3).

```
(advertise
     :sender      B
     :receiver    A
     :reply-with  id0
     :language    KQML
     :ontology    kqml-ontology
     :content     (subscribe
                       :sender        A
                       :receiver      B
                       :in-reply-to   id0
                       :language      KQML
                       :ontology      kqml-ontology
                       :content       (ask-all
                                           :sender        A
                                           :receiver      B
                                           :in-reply-to   id0
                                           :language      Prolog
                                           :ontology      foo
                                           :content       "bar(X,Y)" )))
```

There is no $\texttt{in-reply-to}_{advertise}$ because *advertise* messages are starting points for conversations, and there is no $\texttt{reply-with}_{subscribe}$ value because this is not to be provided by the agent that advertises.

Figure 7: An example of an *advertise* of a *subscribe* of a *ask-all.*

4.2 Intervention and mechanics of conversation performatives

The role of those performatives is to intervene in the normal course of a conversation. The normal course of a conversation is as follows: agent A sends a KQML message (thus starting a conversation) and agent B responds whenever it has a response or a follow-up. The performatives of this category, either prematurely terminate a conversation (*error*, *sorry*), or override this *default protocol* (*standby*, *ready*, *next*, *rest* and *discard*).

```
(error
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>)
```

This performative suggests that the `:sender` received a message, indicated by the value of `:in-reply-to`, that it does not comprehend. The cause for an *error* might be: 1) syntactically ill-formed message, 2) the message has wrong performative parameter values, or 3) it does not comply with the *conversation protocols*. This performative can appear as a response to *any* performative, if necessary. See Figure 9 for examples of cases that may lead to an *error* performative being sent.

Agent A sends to agent B the following KQML message, whose `:in-reply-to` tag suggests that is a follow-up to an *advertise* (see Figure 7 for this advertise; it is an example of a really long KQML message)

```
      (subscribe
                    :sender       A
                    :receiver     B
                    :in-reply-to  id0
                    :reply-with   id1
                    :language     KQML
                    :ontology     kqml-ontology
                    :content      (ask-all
                                              :sender       A
                                              :receiver     B
                                              :in-reply-to  id0
                                              :reply-with   id2
                                              :language     Prolog
                                              :ontology     foo
                                              :content      "bar(X,Y)" ))
```

Upon receiving this *subscribe* message, B responds immediately with an appropriate message (as if processing the *ask-all*)

```
      (tell
                    :sender       B
                    :receiver     A
                    :in-reply-to  id2
                    :reply-with   id3
                    :language     Prolog
                    :ontology     foo
                    :content      "[bar(a,b),bar(a,c)]" )
```

Some time later, when the would–be response to the *ask-all* message changes, B sends another message to A

```
      (tell
                    :sender       B
                    :receiver     A
                    :in-reply-to  id2
                    :reply-with   id4
                    :language     Prolog
                    :ontology     foo
                    :content      "[bar(a,b)]" )
```

In the future, whenever B decides that the would–be response to the *ask-all* message would have been different than the last response sent to A, B will sent a new update to A. Note that B's responses are to the *ask-all* (and not to the *subscribe*), which explains the values of the `:in-reply-to` parameters.

Figure 8: A *subscribe* request and appropriate responses.

```
(sorry
        :sender        <word>
        :receiver      <word>
        :in-reply-to   <word>
        :reply-with    <word>)
```

This performative indicates that the `:sender` comprehends the message, which is correct in every syntactic and semantic aspect, but has nothing to provide as a response. The *sorry* performative may be used also when the agent could give some more responses (assuming the agent has provided responses in the past, as in when responding to a *subscribe*), *i.e.*, theoretically there are more responses, but for whatever reason decides not to continue providing them. When an agent uses *sorry* as a response to a `<performative>` this means that the agent did not process till the end the message to which it is responding to, *e.g.*, an agent that responds with a *sorry* to a *insert*, never inserted the `:content` to its KB. This performative can appear as a response to *any* performative, if necessary.

```
(standby
        :sender        <word>
        :receiver      <word>
        :reply-with    <word>
        :language      <word>
        :ontology      <word>
        :content       (performative_name
                                    :sender        <word>
                                    :receiver      <word>
                       *            :in-reply-to   <word>
                                    :reply-with    <word>
                                    :language      <word>
                                    :ontology      <word>
                                    :content       <expression> ))
```

Normally the `:receiver` of a performative will deliver its response as soon as a response is generated. The *standby* performative that takes a `<performative>` as its content, acts like a modifier on the usual order of affairs. It is a request to the $\texttt{receiver}_{standby}$ to handle the embedded performative as it would normally do, *but* in addition, the `:receiver` should inform the $\texttt{sender}_{standby}$ that it has generated the response and then withhold it until the `:sender` requests for it. In effect, *standby* warns the `:receiver` that the response to the `:content` should not be delivered until the `:sender` of the standby sends an appropriate notification. From the above it is obvious that `performative_name` may be any of the performatives of Table 3 that require a response.

NOTE: In short, *standby* transfers control of the timing of the responses to the `:sender` of the original query, thus reversing the *default protocol*, according to which the `:receiver` delivers its responses at will.

See Figure 10 for an example of an exchange that involves the *standby* performative.

```
(ready
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>)
```

This performative is used by an agent to announce its readiness to deliver at least one of its responses to a KQML message that has been embedded in a *standby* performative. The use of *standby* does not put the additional constraint on the $\texttt{receiver}_{standby}$ (which is also the $\texttt{sender}_{ready}$) to generate all of its possible responses before announcing its readiness. See Figure 10 for an example of an exchange that involves the *ready* performative.

```
(next
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>)
```

This performative is used by an agent that has sent a *standby* in order to request a response from its interlocutor, after the interlocutor (the `receiver` of the *standby*) has announced that it has the response(s) (with the use of *ready*). See Figure 10 and Figure 11 for an example of an exchange that involves the *next* performative.

```
(rest
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>)
```

This performative is to be used by an agent to request for the remaining of the responses, in an exchange that started with a *standby*. In effect, *rest* results to an undoing of the *standby*, since it puts in effect the default protocol where the `:receiver` is in charge of the pace of the conversation and may deliver its responses at will. See Figure 10 and Figure 11 for an example of an exchange that involves the *rest* performative.

```
(discard
```

Agent B has received the *ask-all* message of Figure 2. If B sends either of the following 3 messages as a response to agent A, agent A will respond with an *error*.

Example 1

```
(tell        :sender       B
             :receiver     A
             :reply-with   id2
             :language     Prolog
             :ontology     foo
             :content      "[bar(a,b),bar(c,d)]" )
```

The response is incorrect because it is syntactically ill-formed (the value of the `:in-reply-to` tag is missing).

Example 2

```
(tell        :sender       B
             :receiver     A
             :in-reply-to  id5
             :reply-with   id2
             :language     Prolog
             :ontology     foo
             :content      "[bar(a,b),bar(c,d)]" )
```

The response is incorrect because the value of the `:in-reply-to` is incorrect (assuming that A has sent no message to B with such a `:in-reply-to` tag).

Example 3

```
(tell        :sender       B
             :receiver     A
             :in-reply-to  id1
             :reply-with   id2
             :language     Prolog
             :ontology     foo
             :content      "[foo(a,b,c),foo(c,d,e)]" )
```

The response is semantically incorrect because the value of the `:content` is not an instantiation of the value of $\texttt{content}_{ask-all}$ to which this message serves as a response (the response could also be semantically incorrect if the *performative_name* used in the response had not been one of those allowed by the *conversation policies*, *e.g.*, an *insert*).

Had agent B responded with either of the above messages, agent A would have sent

```
(error       :sender       A
             :receiver     B
             :in-reply-to  id2
             :reply-with   id3)
```

Figure 9: Examples of the three situations that may result in an *error*.

Agent A sends a message identical to the *stream-all* of Figure 3 but this time a *standby* is used.

```
    (standby
                  :sender       A
                  :receiver     B
                  :reply-with   id00
                  :language     KQML
                  :ontology     kqml-ontology
                  :content      (stream-all
                                    :sender       A
                                    :receiver     B
                                    :reply-with   id1
                                    :language     Prolog
                                    :ontology     foo
                                    :content      "bar(X,Y)" ))
```

and agent B this time responds with

```
    (ready
                  :sender       B
                  :receiver     A
                  :in-reply-to  id00
                  :reply-with   id01)
```

Then, agent A requests the first response with

```
    (next
                  :sender       A
                  :receiver     B
                  :in-reply-to  id01
                  :reply-with   id02)
```

and finally A delivers

```
    (tell         :sender       B
                  :receiver     A
                  :in-reply-to  id1
                  :reply-with   id2
                  :language     Prolog
                  :ontology     foo
                  :content      "bar(a,b)" )
```

Note that the `:in-reply-to` value of the *tell* matches the `reply-with` value of the *stream-all* and not that of the *next*, since *tell* is the response to the *stream-all*. From that point on, a couple of different scenarios are possible (see Figure 11).

Figure 10: The exchange of Figure 3 when *standby* is used.

Scenario 1: Agent A requests the second response and B delivers it

```
(next        :sender        A
             :receiver      B
             :in-reply-to   id01
             :reply-with    id03)
(tell        :sender        B
             :receiver      A
             :in-reply-to   id1
             :reply-with    id3
             :language      Prolog
             :ontology      foo
             :content       "bar(c,d)" )
```

Agent A requests for the next response with *next* and B ends the exchange, since there are no more responses, by delivering the end-of-stream marker

```
(next        :sender        A
             :receiver      B
             :in-reply-to   id01
             :reply-with    id04)
(eos         :sender        B
             :receiver      A
             :in-reply-to   id1
             :reply-with    id4)
```

Scenario 2: Agent A might request for the remaining responses all together

```
(rest        :sender        A
             :receiver      B
             :in-reply-to   id01
             :reply-with    id05)
```

in which case the exchange ends with B delivering

```
(tell        :sender        B
             :receiver      A
             :in-reply-to   id1
             :reply-with    id3
             :language      Prolog
             :ontology      foo
             :content       "bar(a,b)")
(eos         :sender        B
             :receiver      A
             :in-reply-to   id1
             :reply-with    id4)
```

Scenario 3: Agent A is not interested in any more responses and lets B know that, by

```
(discard     :sender        A
             :receiver      B
             :in-reply-to   id00
             :reply-with    id06)
```

Figure 11: The possible scenarios that the exchange of Figure 10 might continue with (Figure 10 shows the exchange of Figure 3 when *standby* is used).

```
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :in-reply-to    <word>)
```

This performative indicates to the `:receiver` that the `:sender` is not interested in any more responses (presumably to a multi-response performative). See Figure 10 and Figure 11 for an example of an exchange that involves the *discard* performative.

> NOTE: Performatives that may result to a multi-response are: *stream-all*, *subscribe*, *recommend-all.*

4.3 Networking and Facilitation performatives

The performatives of this category are not speech acts in the pure sense. They are primarily performatives that allow agents to find other agents that can process their queries. Although regular, non–facilitator agents could choose to process them, it would not be particularly helpful since the *facilitation* performatives rely on *advertise* messages and only *facilitators* have the power to make *advertise* messages community–wide.

```
(register
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <expression>)
```

This performative is used by an agent to announce to a facilitator its presence and the symbolic name associated with its physical address. The `:content` comprises of the agent's symbolic name and other information about the agent suggested by some `KQML-agents` ontology.

```
(unregister
        :sender         <word>
        :receiver       <word>
        :in-reply-to    <word>
        :reply-with     <word>)
```

This performative is used to undo a previously sent *register* and can only be used if a *register* has been sent before by the same agent (the $\texttt{sender}_{unregister}$). This also automatically cancels all the commitments made by the agent in the past, *i.e.*, all *advertise* messages sent by the agent to the facilitator become invalid.

```
(transport-address
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <word>)
```

This performative may be used by an agent to announce its relocation in the network (mail forwarding with the U.S. Postal Service meaning). Using *transport-address* updates the information provided by a *register*. Essentially this is a *unregister* (from the physical address where the *register* was sent from), followed by a *register* from the new (current) physical address.

> NOTE: The physical address is automatically captured by the router of a receiving *register* and is not part of the KQML message. Performatives like *register*, *unregister* and *transport-address* generate an association between a symbolic name (which is part of the KQML message) and a physical address and port (captured by the router of a receiving agent, by virtue of the message being sent across the network).

```
(forward
        :from          <word>
        :to            <word>
        :sender        <word>
        :receiver      <word>
*       :in-reply-to   <word>
        :reply-with    <word>
        :language      <word>
        :ontology      <word>
        :content       (performative_name
                                    :sender        <word>
                                    :receiver      <word>
                       *            :in-reply-to   <word>
                                    :reply-with    <word>
                                    :language      <word>
                                    :ontology      <word>
                                    :content       <expression> ))
```

This performative is a request from agent `:sender` to agent `:receiver` to deliver a message that originated from agent `:from`, to agent `:to`. The `:receiver` of the *forward* might be the `:to` agent, in which case the `:receiver` just processes the message in `:content`. Agent `:receiver` might not be able to deliver the message to agent `:to` in which case it should send a *forward* to some other agent that has a better chance to get the message to the `:to` agent. The following constraints hold:

- $\texttt{from}_{forward} = \texttt{sender}_{performative\_name}$
- $\texttt{to}_{forward} = \texttt{receiver}_{performative\_name}$

See Figure 12 and Figure 13 for an example of an exchange that involves the *forward* performative.

NOTE: The `:in-reply-to` parameter for *forward* is optional and as far as we know only makes sense in the context of responding to *recommend-one*, *recommend-all*, *broker-one* and *broker-all* in which case the *forward* is a *direct* response to the `<performative>`. In the case of *forward* being used to respond to *broker-one* and *broker-all*, the `:sender` value of the embedded performative is omitted.

```
(broadcast
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <performative>)
```

This performative is a request to *forward* the `<performative>` to all agents that the `:receiver` knows of, *i.e.*, to all agents that have registered (using *register* with the `:receiver`, if `:receiver` is a *facilitator*), or that the `:receiver` might know of. A *broadcast* is equivalent (and implemented in such a manner) to a series of *forward* messages to all such agents.

NOTE: All agents (both facilitators and regular agents) are by default capable of processing *forward* and *broadcast*, so agents do not have to send *advertise* messages for those performatives. This is the reason why *broadcast* requires no `:in-reply-to` value. What might have been *advertised* is the $\texttt{content}_{broadcast}$ and it is the `:content`'s `:in-reply-to` value that is of interest.

```
(broker-one
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                    :sender         <word>
                                    :reply-with     <word>
                                    :language       <word>
                                    :ontology       <word>
                                    :content        <expression> ))
```

Let us consider the following situation: agent C knows of agent A, agent A knows of agent B and agent B knows of agent D ("knows of" is synonymous to "is able to deliver messages to"). Agent C wants agent D to process an *ask-if* for which agent D has advertised its ability and commitment to do so (it is possible for C to know that agent D exists but still not being able to deliver messages to it, *e.g.*, C learned about D after a *recommend-one* message similar to that of Figure 15). So, agent C sends the following *forward* message to agent A.

```
(forward
        :from           C
        :to             D
        :sender         C
        :receiver       A
        :reply-with     id00
        :language       KQML
        :ontology       kqml-ontology
        :content        (ask-if
                                        :sender         C
                                        :receiver       D
                                        :in-reply-to    id1
                                        :reply-with     id2
                                        :language       Prolog
                                        :ontology       foo
                                        :content        "bar(a,b)" ))
```

Agent A is not the `to`$_{forward}$, and cannot deliver to it, so it sends another *forward* to B, hoping that B will have a better chance to accomplish the task. If B is incapable of doing so, B will respond with a *sorry* to A and A will eventually respond with a *sorry* to C's original *forward* request (such a *sorry* will be a response to the *forward*, so the `:in-reply-to` will be `id00`). This *sorry* will not get back to A wrapped in a *forward*.

```
(forward
        :from           C
        :to             D
        :sender         A
        :receiver       B
        :reply-with     id01
        :language       KQML
        :ontology       kqml-ontology
        :content        (ask-if
                                    :sender         C
                                    :receiver       D
                                    :in-reply-to    id1
                                    :reply-with     id2
                                    :language       Prolog
                                    :ontology       foo
                                    :content        "bar(a,b)" ))
```

See Figure 13 for the continuation of this exchange.

Figure 12: A conversation involving the *forward* performative. See Figure 13, also.

Continuing the exchange that is shown in Figure 12, agent B sends to agent D the following *forward* message.

```
(forward
        :from           C
        :to             D
        :sender         B
        :receiver       D
        :reply-with     id02
        :language       KQML
        :ontology       kqml-ontology
        :content        (ask-if
                                      :sender         C
                                      :receiver       D
                                      :in-reply-to    id1
                                      :reply-with     id2
                                      :language       Prolog
                                      :ontology       foo
                                      :content        "bar(a,b)" ))
```

There are two possible scenarios for D upon receiving this last message.
Scenario 1: D can deliver directly to C, *i.e.*, D knows of C even though C does not know of D. In this case C sends the following message

```
(tell   :sender         D
        :receiver       C
        :in-reply-to    id2
        :reply-with     id3
        :language       Prolog
        :ontology       foo
        :content        "bar(a,b)" )
```

Scenario 2: If D cannot deliver directly to C, then the response has to follow a similar path back to C, *i.e.*, the response is wrapped in *forward* messages that travel from D → B → A → C, and D starts this by

```
(forward
        :from           D
        :to             C
        :sender         D
        :receiver       B
        :reply-with     id03
        :language       KQML
        :ontology       kqml-ontology
        :content        (tell         :sender         D
                                      :receiver       C
                                      :in-reply-to    id2
                                      :reply-with     id3
                                      :language       Prolog
                                      :ontology       foo
                                      :content        "bar(a,b)" ))
```

that is followed by messages similar to those of Figure 12.

Figure 13: The rest of the exchange of Figure 12.

The constraint is that `performative_name` can be one of the performatives that can be used with *advertise* (see page 19). This is a request to find an agent that *can* and *will* process the `:content`, (*i.e.*, an agent that has sent an *advertise* with such a `:content`) in the name of the receiver of the *broker-one* (so all responses from the third party will be directed to the broker, *i.e.*, the $\texttt{receiver}_{broker-one}$). After receiving the response, the broker will sent it to the `:sender` of the *broker-one*, wrapped in a *forward* originating from the broker-ed agent. See Figure 14 for an example of an exchange that involves the *broker-one* performative.

> NOTE: The `in-reply-to` value only makes sense if `:receiver` is not a *facilitator*, in which case it might have advertised the *broker-one*. The same holds for the remaining performatives of this category.

```
(broker-all
        :sender          <word>
        :receiver        <word>
*       :in-reply-to     <word>
        :reply-with      <word>
        :language        <word>
        :ontology        <word>
        :content         (performative_name
                                      :sender          <word>
                                      :reply-with      <word>
                                      :language        <word>
                                      :ontology        <word>
                                      :content         <expression> ))
```

This performative is a request to find **all** agents that *can* and *will* process the content (similar to *broker-one*). The constraint is again that `performative_name` can be one of those that may be used with advertise (see page 19).

```
(recommend-one
        :sender          <word>
        :receiver        <word>
*       :in-reply-to     <word>
        :reply-with      <word>
        :language        <word>
        :ontology        <word>
        :content         (performative_name
                                      :sender          <word>
                                      :language        <word>
                                      :ontology        <word>
                                      :content         <expression> ))
```

Agent facilitator has received an *advertise* message from agent A, identical to the first message in Figure 6, except for $\texttt{receiver}_{advertise} = facilitator$ and $\texttt{sender}_{ask-if} = facilitator$). Later, agent C sends the following message to the facilitator

```
(broker-one  :sender      C
             :receiver    facilitator
             :reply-with  id3
             :language    KQML
             :ontology    kqml-ontology
             :content     (ask-if   :sender      C
                                    :reply-with  id4
                                    :language    Prolog
                                    :ontology    foo
                                    :content     "bar(X,Y)" ))
```

Agent *facilitator*, after searching through the *advertise* messages that have been sent to him, decides to send the following KQML message to agent A

```
(ask-if      :sender       facilitator
             :receiver     A
             :in-reply-to  id1
             :reply-with   id4
             :language     Prolog
             :ontology     foo
             :content      "bar(X,Y)" ))
```

Agent A responds with the following message

```
(tell        :sender       A
             :receiver     facilitator
             :in-reply-to  id4
             :reply-with   id5
             :language     Prolog
             :ontology     foo
             :content      "bar(X,Y)" ))
```

and finally, agent facilitator sends the following KQML message to agent C, as a response to the original *broker-one* message from C.

```
(forward     :from         C
             :sender       facilitator
             :receiver     C
             :in-reply-to  id3
             :reply-with   id6
             :language     KQML
             :ontology     kqml-ontology
             :content      (tell     :receiver    C
                                     :language    Prolog
                                     :ontology    foo
                                     :content     "bar(X,Y)" ))
```

The `:from` of the *forward*, which is also the value of the `:sender` of the *tell*, is omitted for reasons that are made clear in the semantic description (see [3]).

Figure 14: An example of a *broker-one* performative and the follow-up

The constraint is that **performative_name** be one of the performatives that can be used in *advertise* (see page 19). This is a request to suggest an agent that *can* process the `:content` (again, as is the case with *broker-one*, use is made of the *advertise* messages that the $\texttt{receiver}_{recommend-one}$ has received). Since more than just an agent name is needed in order for $\texttt{sender}_{recommend-one}$ to be able to contact this agent, the appropriate response of $\texttt{receiver}_{recommend-one}$ will be to *forward* the *advertise* message that satisfies the request. See Figure 15 for an example of an exchange that involves the *recommend-one* performative.

```
(recommend-all
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                    :sender         <word>
                                    :language       <word>
                                    :ontology       <word>
                                    :content        <expression> ))
```

The constraint is that **performative_name** can be one of the performatives that can be used in *advertise* (see page 19). This is a request to suggest **all** agents that *can* process the content (similar to *recommend-one*).

```
(recruit-one
        :sender         <word>
        :receiver       <word>
*       :in-reply-to    <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                    :sender         <word>
                                    :reply-with     <word>
                                    :language       <word>
                                    :ontology       <word>
                                    :content        <expression> ))
```

The constraint is that **performative_name** can be one of the performatives that can be used in *advertise* (see page 19). This performative is like a *broker-one* but responses will be directed back to the issuer of the *recruit-one*. In effect, *recruit-one* is equivalent to

Agent *facilitator* has received an *advertise* message from agent A, identical to the first message in Figure 6 (except $\texttt{receiver}_{advertise}$ = *facilitator* and $\texttt{sender}_{ask-if}$ = *facilitator*). Later, agent C sends the following message to the *facilitator*

```
(recommend-one
              :sender       C
              :receiver     facilitator
              :reply-with   id3
              :language     KQML
              :ontology     kqml-ontology
              :content      (ask-if
                                     :sender       C
                                     :language     Prolog
                                     :ontology     foo
                                     :content      "bar(X,Y)" ))
```

Agent *facilitator* sends the following KQML message to agent C, after searching through the *advertise* messages that have been sent to it.

```
(forward
     :from          A
     :to            C
     :sender        facilitator
     :receiver      C
     :in-reply-to   id3
     :reply-with    id5
     :language      KQML
     :ontology      kqml-ontology
     :content       (advertise
                         :sender       A
                         :receiver     C
                         :reply-with   id1
                         :language     KQML
                         :ontology     kqml-ontology
                         :content      (ask-if
                                           :sender       C
                                           :receiver     A
                                           :in-reply-to  id1
                                           :language     Prolog
                                           :ontology     foo
                                           :content      "bar(X,Y)" )))
```

Note that $\texttt{receiver}_{advertise}$ = C instead of *facilitator* which was the value of $\texttt{receiver}_{advertise}$ in A's *advertise*. Since A's *advertise* was made to the *facilitator*, the value of the $\texttt{receiver}_{advertise}$ may be set by the *facilitator* to the name of any agent that has registered with the *facilitator*.

Figure 15: An example of a *recommend-one* and a response to it.

```
(forward
          :from          <word>
          :to            <word>
          :sender        <word>
          :receiver      <word>
*         :in-reply-to   <word>
          :reply-with    <word>
          :language      <word>
          :ontology      <word>
          :content       (performative_name
                                    :sender        <word>
                                    :receiver      <word>
                                    :in-reply-to   <word>
                                    :reply-with    <word>
                                    :language      <word>
                                    :ontology      <word>
                                    :content       <expression>))
```

with the additional constraint that $\texttt{to}_{forward} = \texttt{receiver}_{performative\_name} = X$, where X is to be provided by the $\texttt{receiver}_{forward}$, *i.e.*, the $\texttt{receiver}_{recruit-one}$, making use of the *advertise* performatives known to it (likewise for the $\texttt{in}-\texttt{reply}-\texttt{to}_{performative\_name}$) See Figure 16 for an example of an exchange that involves the *recruit-one* performative.

```
(recruit-all
          :sender        <word>
          :receiver      <word>
*         :in-reply-to   <word>
          :reply-with    <word>
          :language      <word>
          :ontology      <word>
          :content       (performative_name
                                    :sender        <word>
                                    :receiver      <word>
                                    :in-reply-to   <word>
                                    :reply-with    <word>
                                    :language      <word>
                                    :ontology      <word>
                                    :content       <expression> ))
```

The constraint is that `performative_name` can be one of the performatives that can be used in *advertise* (see page 19). This performative is like a *broker-all* but responses will be directed to the issuer of the *recruit-all*. In effect *broker-all* is equivalent to a series of *forward* messages, like those mentioned in the description of *recruit-one*.

Agent *facilitator* has received an *advertise* message from agent A, identical to the first message in Figure 6 (except for $\texttt{receiver}_{advertise} = facilitator$ and $\texttt{sender}_{ask-if} = facilitator$). Later, agent C sends the following message to the *facilitator*

```
(recruit-one
              :sender       C
              :receiver     facilitator
              :reply-with   id3
              :language     KQML
              :ontology     kqml-ontology
              :content      (ask-if
                                        :sender       C
                                        :reply-with   id4
                                        :language     Prolog
                                        :ontology     foo
                                        :content      "bar(X,Y)" ))
```

Agent *facilitator* sends the following KQML message to agent A, after searching through the *advertise* messages that have been sent to it.

```
(forward
              :from         C
              :to           A
              :sender       facilitator
              :receiver     A
              :reply-with   id4
              :language     KQML
              :ontology     kqml-ontology
              :content      (ask-if
                                        :sender        C
                                        :receiver      A
                                        :in-reply-to   id1
                                        :reply-with    id4
                                        :language      Prolog
                                        :ontology      foo
                                        :content       "bar(X,Y)" ))
```

Agent A responds with the following message that is sent to C and **not** to the *facilitator*

```
(tell
              :sender        A
              :receiver      C
              :in-reply-to   id4
              :reply-with    id5
              :language      Prolog
              :ontology      foo
              :content       "bar(X,Y)" )
```

Figure 16: An example of a *recruit-one* and its follow-up.

Summary

Let us summarize the features of a domain of KQML–speaking agents:

- In each domain of KQML–speaking agents there is at least one agent with a special status called *facilitator* that can always handle the networking and facilitation performatives. Agents advertise to their facilitator, *i.e.*, they send *advertise* messages to their facilitators, thus announcing the messages that they are committed to accepting and properly processing. Advertising to a facilitator is like advertising to the community (either of their own domain or of some other domain). Agents can still advertise on a one-to-one basis, if they so wish, and such advertisements do not commit them to processing messages from agents other than the `:receiver` of the *advertise*. Actually, such advertisements will never be shared with other agents, because of the "personal" nature of the *advertisements*, *i.e.*, they are addressed to particular agents and only *facilitators* can supersede that; see Table 5, also. Agents can use their facilitator either
  - to have their queries properly dispatched to other agents, using *recruit-one*, *recruit-all*, *broker-one* or *broker-all*, or
  - to send a *recommend-one* or a *recommend-all* to get the relevant *advertise* messages and directly contact agent(s) that may process their queries.
- Agents can access agents in other domains either through their facilitator, or directly. This implies that a smart facilitator may be built in such a way that whenever it cannot find a useful, relevant *advertise* from an agent in its domain, it may query another facilitator, in some other domain. Such an action initiates a sub-dialogue with another facilitator in order to serve the original query. Elaborate protocols of this kind are examples of conversations (interactions) that be built on top of the conversation policies presented in [3]
- Facilitators may request the services of other facilitators in the same way that regular agents may request the services of their facilitator. Facilitators do *not advertise*, not even to other facilitators. The model we imply is one where regular agents *advertise* their services to their facilitators and thus facilitators become providers of *query–processing* information about the agents in their domain; such information can then be accessed by any agent (regular or facilitator), using the *facilitation performatives*.
- We use the term *facilitator* to refer to all kinds of special services that may be provided by *specialized* agents, such as *Agent Name Servers* (ANS), *proxy agents*, or *brokers* ([2]).

References

[1] ARPA Knowledge Sharing Initiative. Specification of the KQML agent-communication language. ARPA Knowledge Sharing Initiative, External Interfaces Working Group working paper., July 1993.

[2] Tim Finin, Anupama Potluri, Chelliah Thirunavukkarasu, Don McKay, and Robin McEntire. On agent domains, agent names and proxy agents. In *CIKM Intelligent Information Agents Workshop*, Baltimore, MD, December 1995.

[3] Yannis Labrou. *Semantics for an Agent Communication Language.* PhD thesis, University of Maryland, Baltimore County, August 1996.

4. A SEMANTICS APPROACH FOR KQML

Intelligent Agent Integration Technology

Prepared by:
Yannis Labrou
and
Tim Finin
Co-Principal Investigator
University of Maryland - Baltimore County
finin@cs.umbc.edu, (410)455-3522

This section includes the first comprehensive published paper on a semantics for KQML. It appeared in the 1994 Conference on Information and Knowledge Management.

A semantics approach for KQML – a general purpose communication language for software agents *

Yannis Labrou
Computer Science Department
University of Maryland, Baltimore County
Baltimore MD 21228
email: jklabrou@cs.umbc.edu
voice: (410) 455-2667
fax: (410) 455-3969

Tim Finin
Computer Science Department
University of Maryland, Baltimore County
Baltimore MD 21228
email: finin@cs.umbc.edu
voice: (410) 455-3522
fax: (410) 455-3969

Abstract

We investigate the semantics for Knowledge Query Manipulation Language (KQML) and we propose a semantic framework for the language. KQML is a language and a protocol to support communication between software agents. Based on ideas from speech act theory, we propose a semantic description for KQML that associates descriptions of the cognitive states of agents with the use of the language's primitives (performatives). We use this approach to describe the semantics for the basic set of KQML performatives. We also investigate implementation issues related to our semantic approach. We suggest that KQML can offer an all purpose communication language for software agents that requires no limiting pre-commitments on the agents' structure and implementation. KQML can provide the Distributed AI, Cooperative Distributed Problem Solving and Software Agents communities with an all purpose language and environment for intelligent inter-agent communication.

1 Introduction

Let us picture a company where employees keep calendars in their personal computers. A database keeps information on the employees, such as names, offices, phone numbers. Another database may register conference rooms, with additional information regarding capacity, availability, scheduled activities and so on. One may want to build a system that can schedule group meetings in the company, according to the availability of employees and locations. The well-known approach is to built an application from scratch, so that *one* application holds all necessary information and knowledge. The alternative would be to use the existing applications. Doing that, would require: 1) the applications to be able to comprehend each other's knowledge stores, despite differences in implementation languages and knowledge representation schemes, and 2) the applications to communicate with each other and dynamically make queries, answer them, assert or remove facts from their knowledge stores, in short, to interact intelligently.

This example is an instance of the larger problem of providing for an environment where software agents may effectively communicate and exchange knowledge and information. Addressing this problem is the primary goal of the *ARPA Knowledge Sharing Effort* (KSE) [23]. KSE is an initiative to develop the technical infrastructure to support the sharing of knowledge among systems [22]. Its goal is to develop new systems by selecting components from libraries of reusable modules and assembling them together. One of the key areas identified by KSE was that of protocols for communication between separate knowledge-based modules, as well as between knowledge-based systems and databases. The result was *Knowledge Query Manipulation Language (KQML)* (see [1, 2, 14] for documentation on KQML) a message format and a message-handling protocol to support run-time knowledge sharing and interaction among agents.

Interaction is more than an exchange of messages. Issues associated with it, are: *models of agents* (beliefs, goals, representation and reasoning), *interaction protocols* (an interaction regime that guides the agents) and *interaction languages* (languages that introduce standard message types that all agents interpret identically). KQML is intended to be a *universal* interaction language, that supports communication through explicit linguistic actions. Our focus in this paper is the formal description of the semantics of the language. Although the language is partly designed and in use, it lacks a formal semantics, and its current description [2] is based on natural language descriptions of its primitives called *performatives*. We believe that a formal semantics is necessary for the unambiguous definition of the language, and its appropriate use. Furthermore, the semantic description is related to implementation issues.

Research communities with a potential interest in such a language are those of *Distributed Artificial Intelligence* (DAI[1]) ,the subfield of AI concerned with concurrency in AI computations, *(Cooperative) Distributed Problem Solving*, that studies how a loosely coupled network of problem solvers can work together to solve problems that are beyond their individual capabilities [12], and *Multi Agent Systems*, concerned with coordinating behavior among a collection of (possibly pre-existing) autonomous intelligent agents. The rising demand for *software agents* that can interoperate [16], and for *intelligent agents* that can take advantage of the

[1]For an introduction to the issues that DAI is concerned with, see [4] and [15].

enormous resources of today's Internet (like Etzioni's *Internet softbots* [13]) provide a proving ground for a communication language. KQML can be used in any environment where software agents need to communicate something more than pre-defined and fixed statements of facts and provides for dynamic run-time interaction, so that intelligent agents can combine their efforts, or make use of other agents' abilities, in order to achieve their goals.

In the remainder of this paper we will begin by providing a brief introduction to speech act theory which underlies our approach to defining the semantics of KQML. We will then associate KQML messages with speech acts and present a general semantic framework for KQML. Following this framework, we will give the semantics for a small set of KQML performatives. In the final two sections of the paper we discuss the impact of our analysis on some software implementation issues and discuss the kinds of applications which are appropriate for KQML.

2 Speech act theory and speech act semantics

Speech act theory is a high-level theoretical framework developed by philosophers and linguists to account for human communication. It has been extensively used, formalized and extended within the fields of Computational Linguistics and AI as a general model of communication between arbitrary agents. As such, we believe that speech act theory can provide us with a framework for the semantics of KQML, a language focused on the communication between software agents. Speech act theory is primarily concerned with the role of language as action. The following three distinct actions can be identified in a speech act: (1) a *locution*, i.e., the actual physical utterance (with a certain context and reference), (2) an *illocution*, i.e, the conveying of the speakers intentions to the hearer, and (3) the *perlocutions*, i.e., actions that occur as a result of the illocution. For example, "I order you to shut the door" is a locution, its utterance is the illocution of a command to shut the door and the perlocution may be (if all goes well) that the hearer shuts the door. An illocution is usually considered to have two parts: an *illocutionary force* and a *proposition*. The illocutionary force classifies speech acts into the following classes[2]: 1) *assertives*, that are statements of facts, 2) *directives*, that are commands, requests or suggestions, 3) *commisives*, e.g., promises, that commit the speaker to a course of action, 4) *declaratives*, that entail the occurrence of an action in themselves[3], and 5) *expressives*, that express feelings and attitudes.

There is no consensus in the literature regarding the semantic approaches for speech acts but no matter what one may consider as speech act semantics, it is necessary to make reference to the cognitive states of the agents that use them. After all, speech acts are supposed to be the result of agents' efforts to act upon the world and/or other agents. The representation of and reasoning about the states of agents and the world and how agents' actions affect them is a prerequisite for any semantic approach. There is a plethora of approaches regarding the abstractions (models) used for capturing and describing such states, depending on one's motivations. They range from informal references to propositional attitudes, like "believe" or "want", as in Searle's early work [25] where speech acts are used in the context of the investigation of *reference* and other *Philosophy of Language* issues, to strict formalisms, as in the work of Cohen and Levesque [6, 8, 7] that define a formal model of the cognitive state of an agent and then use it to interpret speech acts as actions that are derived, guided and controlled in the context of the cognitive states of the related agents. Campbell [5] uses predicates (that stand for epistemic operators), and propositions to describe mental states associated with specific speech acts (like warning or bargaining). Cohen and Perrault in their *plan-based theory of speech acts* [9] use a *believe* modal operator based on Hintikka's ideas about propositional attitudes, knowledge and belief [20]. Singh is interested in modelling agents in terms of beliefs and intentions [26] and uses this description to provide a semantic approach for speech acts [27], enhancing the usual model-theoretic framework with modal operators for the primitive concepts of *intention* and *know-how*. The common denominator of most of the formal semantic approaches is the *possible-world model* that has an axiomatization in terms of modal logic (for an introduction to the possible worlds model and the issues related to it see [21]).

We adopt Searle's description (approach) for speech acts [25, 24]. A speech act may be described as*F(P)* where *F* is the illocutionary force indicator and *P* is the propositional content of the illocutionary act[4]. Searle suggests the following seven components of the illocutionary force:

1. The *illocutionary point* is a fundamental primitive notion. The illocutionary points are: *assertive*, *directive*, *commisive*, *declarative*, and *expressive*. The illocutionary point of a type of illocutionary act is achieved if the act is successful. The illocutionary point of a promise to do act P (commisive), is for the speaker to commit himself to doing P and the illocutionary act will be successful if the promise is to be kept in the future.

2. The *degree of strength* of the illocutionary point can distinguish between "shut the door!" and "could you please close the door?" that are both directives, but the first is a command and the second is a plea.

3. The *mode of achievement* suggests the special ways or set of conditions under which the illocutionary point has to be achieved in the performance of the speech act. A command may require a position of authority on behalf of the speaker; use of this authority may be necessary in issuing the utterance and eventually achieving the illocutionary point.

4. The *propositional content conditions* impose what can be in the propositional content *P* for a specific force *F*. For example, a speaker can not promise that a third agent will do something.

5. The *preparatory conditions* are conditions that should hold for the successful performance of an illocutionary act. In the case of a promise, such conditions might be that whatever was promised is in the hearer interest and the hearer in fact wanted him to issue the promise.

6. The *sincerity conditions* relate to the psychological (or cognitive) state of the agent. Agents have beliefs, intentions and desires. The propositional content of the illocutionary act should be identical to the propositional content of their psychological state.

[2] variations of this classification appear also in the literature

[3] as in "I name this ship the *Titanic*"

[4] The truth might be a little more complicated because P can by a proposition plus syntactic features and a context for the utterance.

7. Finally, the *degree of strength of the sincerity conditions* suggests the existence of a degree of strength in the expression of the psychological state of the speaker. "Requesting" and "begging", do not suggest the same level of desire for something to occur.

3 KQML and speech act theory, as a context for its semantics

KQML is intended as a general purpose communication language for the exchange of information and knowledge between software agents. Here is an example of a KQML message:

```
(tell        :language      prolog
             :ontology      Genealogy
             :in-reply-to   q1
             :sender        Gen-1
             :receiver      Gen-DB
             :content       ''father(John,Alice)'')
```

In KQML terminology, "tell" is a *performative*[5] (see Table 1 for more KQML performatives). Performatives explicitly suggest the illocutionary force. The value of the `:content` slot is an expression in some "computer interpreted" language[6], in other words it is the propositional content of the illocutionary act (technically, the illocutionary act is the "delivery" of a KQML message). The other parameters (*keywords*), introduce values that provide a context for the interpretation of the propositional content and at the same time hold information to facilitate the processing of the message. In this example, "Gen-1" is stating to "Gen-DB" (these are symbolic names for applications), in *Prolog*, that "father(John,Alice)". This is a response to the KQML message (illocutionary act) identified by "q1". The ontology[7] named "Genealogy" may provide additional information regarding the interpretation of the content.

We will use the term semantics to refer to: 1) everything that provides for an unambiguous interpretation of the performative, viewed as an illocutionary force indicator, 2) the perlocutionary effects, i.e, how agents' states change after sending or receiving a KQML message, and 3) criteria that suggest when the illocutionary point of the performative is satisfied.

Searle broke down the illocutionary force into seven components (presented in Section 2). Next, we examine those components that are of interest to us, and how they relate to our effort to provide meaning to performatives. The performative's *illocutionary point* and *degree of strength* are axiomatically defined by the designers (in our current analysis we ignore the degree of strength). Table 1 shows the illocutionary points for the performatives of this presentation. The *sincerity conditions* and their *degree of strength* are of no immediate interest, because we assume that all agents are sincere to the best of their ability. The *propositional content conditions* assure that agents do not make promises about other agents, they do not respond to queries not directed to them, etc. They are enforced by the *conversation policies* (more about them in the Section 6.1) and the application programmer[8]. The *mode of achievement* refers to establishing certain relationships between speakers and hearers that make certain illocutionary acts, meaningful. The mode of achievement is set by the "organizational" hierarchy or interaction protocol that the agents may use in their interaction. In Contract Net [28] , the fact that some agents act as managers and others as potential contractors, creates a context for the negotiation [11] , through bidding, that characterizes the protocol. The *preparatory conditions* are viewed as preconditions on the cognitive state, for an agent to use a performative.

For the *perlocutionary effects* we provide suggestions for the states of the sender, after sending a message and for the receiver, after processing it (presented as postconditions). The objective is to help with the interpretation of the performative, by suggesting the desired effects of its use, and to link (and restrict) the possible responses that will be acceptable follow-ups to the sender's action, by establishing preconditions for the possible response.

Finally, we need to know when the illocutionary point of a performative is eventually satisfied, e.g., a query is satisfied when it is answered appropriately. Other illocutionary acts are satisfied just by being uttered, such as telling (*tell*), and others, like asking (*ask-if* and other query performatives), require a further exchange of messages, i.e., a "conversation". Thus, we provide satisfiability (completion) condition, that indicate the state of affairs *after* the completion of the speech act (performative).

4 A framework for the semantics of KQML

The central idea is to formally define cognitive states for agents, use them to describe the performative, the preconditions, postconditions and satisfiability conditions, mentioned before, and associate those states with the use of the performatives. We use expressions in *First Order Predicate Calculus* (FOPC), to do that. In these expressions we use operators that have a reserved meaning (the operators will be identified by predicates). The use of such operators, to describe mental states of agents that use speech acts, can be found in approaches as diverse as Campbell's [5] and Singh's [27]. The operators used in this presentation are:

1. **Bel**, as in bel(A,P) which has the meaning that P is true for A. P is an expression in the native language of A's application. We will further refer to this operator in Section 7. For now, it suffices to say that P "exists" in the agent's *knowledge base* (or *virtual knowledge base*).

2. **Know**, like the following two operators, refers to the cognitive state of the agents[9]. Know(A,P) expresses a state of knowledge awareness on behalf of A, about P.

3. **Want**, as in want(A,P), to mean that agent A desires the event (or state) described by P, to occur.

4. **Intend**, as in intend(A,P), to mean that A has every intention of doing P.

[5] term first coined by Austin [3], to suggest that some verbs can be uttered so that they perform some action (later, it was decided that *all* verbs may be considered as performatives)

[6] In the full version of KQML (not presented here), the content may also be a KQML message itself.

[7] An ontology is a repository of semantic and primarily pragmatic knowledge over a certain domain. Ontologies are part of the Shared and Reusable Knowledge Bases Group of the KSE.

[8] It is necessary for the programmer to guarantee that an application does not use bizarre propositional contents for a certain performative, due to their pragmatic nature. Since KQML is opaque to the content of the message, there is no way to guarantee that, for instance, an agent does not promise that "the time is 12:30PM". However, the conversation policies will ensure that if agent A poses a query to agent B, B will respond only to A and A will receive responses to this query, only from B.

[9] As such, all three could be termed as *epistemic operators*.

Name	*Illocutionary point*	*Meaning*
tell	assertive	A states to B that A believes the content to be true
deny	assertive	A states to B that A does not believe the content true
ask-if	directive	A wants to know what B believes regarding the truth status of the content
ask-all	directive	A wants to know all B's responses that would make the content true of B (the response will be a collection of expressions)
stream-all	directive	like *ask-all*, but the responses are to be delivered one by one
eos	declarative	end of a stream of responses to an earlier query
error	assertive	A states to B that B's message was not processed by A
sorry	assertive	A states to B that B's message was processed by A, but no reply can be provided

Table 1: Performatives mentioned in this presentation, for sender A and recipient B.

Roughly, *know*, *want* and *intend* stand for the psychological states of knowledge, desire and intention, respectively. Only for the *bel* predicate, it is the case that P is an expression in the agent's implementation language. For all other three operators, P is an expression that combines other operators, and stands for an event or a state of affairs. For example, it is correct to say "know(A,bel(B,foo(a,b)))" (if B "speaks" Prolog) but not "know(A,foo(a,b))". One can ask if (and how) those operators are implemented in an application. The short answer is that only the *bel* operator has to have a concrete meaning (that depends on the application language or knowledge representation language and scheme), and the others prescribe a state of affairs for the agent that is associated with the use of the language. The use of a specific performative suggests an associated state for the speaker, as in assuming when one asks X, that he wants to know X.

The semantics are implemented through the *conversation policies* to be provided by the KQML developers, and the *handler functions*, to be provided by the application programmer [10]. The conversation policies indicate what performatives can follow the utterance of a certain performative, so that agents can have meaningful conversations. The conversation policies are an integral part of the semantics and are consistent with the preconditions, postconditions and completion conditions, to be introduced for the performatives. For example, when an *ask-if* is uttered, it can only be followed (see Figure 1) by a *tell* or *deny*[11] which, in return, can only be uttered as a response to an "asking". Figure 1 gives an example of the conversation policy for the small subset of KQML performatives introduced here. It as part of an Augmented Transition Network specification, with the constraints and relating actions missing. Details about the implementation and functionality of conversation policies (along with details for the structure and construction of KQML speaking agent) can be found in Section 6. The *handler functions* are defined in order to process messages *received* by an application and should be consistent with the semantics described here. Handler functions are not application dependent, but rather language dependent[12], in the sense that all applications using the same language share the same handler functions.

5 Semantics for KQML performatives

The general semantic description of a KQML performative has the following six constituents:

1. A natural language description of the performative's intuitive meaning.
2. An expression in our logic that describes the illocutionary act. For all practical purposes, this is a formal representation of the natural language description.
3. Preconditions that indicate the necessary state for an agent in order to send a performative and for the receiver to accept it and process it.
4. Postconditions that describe the states of agents after the utterance of a performative (for the sender) and after the receipt (but before a counter utterance) of a message (by the receiver)[13].
5. Completion conditions for the sender that indicate the final state of the sender, after possibly a conversation has taken place and the intention suggested by the performative that started the conversation, has been fulfilled.
6. Any natural language comments that we might find suitable to enhance the understanding of the performative.

If there are non-null preconditions for the receiver, this will mean that the performative can only be some-kind of response to the use of another performative that established

[10] The software architecture of a KQML speaking agent is shown in Figure 2 and more details about it are given in Section 6.

[11] A *sorry* or an *error* may also occur.

[12] For an application written in Prolog, a handler function to handle *ask-if* messages, looks like this:
```
handle(ask-if,Content):-
(call(Content) ->
(reply_to_message_with(tell,Content));
(reply_to_message_with(deny,Content))).
```
where reply_to_message_with interacts with the conversation module, that implements the conversation policies, to provide the appropriate values for the other message parameters and finally deliver the response.

[13] After the receiver replies, a new *cycle* of preconditions and postconditions gets started.

those preconditions[14]. No preconditions are necessary for the receiver of a performative that starts a conversation (see Pre(B) for the query performatives, such as *ask-if, ask-all, stream-all*).

In a conversation, the postconditions for the sender of a message should be a subset of the preconditions for the receiver of the message that may follow (compare Post(A) for *ask-if* and Pre(A) for *tell*).

When no conversation is necessary after the utterance of a performative, completion (satisfiability) conditions are a subset of the postconditions. Such performatives are satisfied just by being successfully uttered and processed by the intended recipients.

In the rest of this section we give the semantic descriptions for the eight performatives in Table 1. In these descriptions A is the sender, B is the receiver and X is the propositional content. All expressions mentioned as preconditions, postconditions and completion conditions, are the minimum necessary for our specification of KQML.

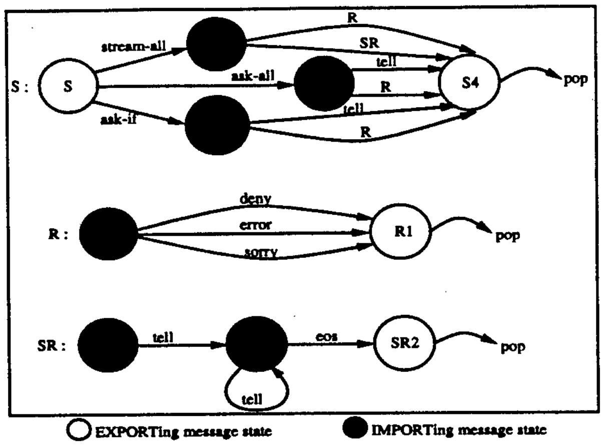


Figure 1: A simple example of an ATN to parse sequences of KQML messages.

- **ask-if(A,B,X)**
  1. A wants to know what B believes regarding the truth status of the content.
  2. want(A,know(A,Y)),
     where Y may be one of the following:
     bel(B,X), bel(B,NOT(X)), NOT(bel(B,X))
     (this means that Pre(A) could also be stated as:
     want(A,know(A,bel(B,X))) OR
     want(A,know(A,bel(B,NOT(X)))) OR
     want(A,know(A,NOT(bel(B,X)))))
  3. Pre(A): want(A,know(A,Y))
     (optionally, NOT(know(A,Y)) should also hold)
     Pre(B): NONE[15]
  4. Post(A): intend(A,know(A,Y))
     Post(B): know(B,want(A,know(A,Y)))
  5. Completion(A): know(A,Y)
  6. Not believing something is not necessarily the same as believing its negation, although this may be the case for certain systems.

- **ask-all(A,B,X)**
  1. A wants to know all of B's responses that make X true of B. X is an expression with variables and A wants *all* the expressions that are true for B and have values for these variables[16]
  2. want(A,know(A,Y)),
     where Y is bel(B,Y') and Y' is a finite collection of Y_1, Y_2, ... Each Y_i is an instance of X with values for the variables in X, identified by the :aspect parameter and each Y_i appears once in this collection (the collection might be empty).
  3. Pre(A): want(A,know(A,Y))
     (optionally, NOT(know(A,Y)) should also hold)
     Pre(B): NONE
  4. Post(A): intend(A,know(A,Y))
     Post(B): know(B,want(A,know(A,Y)))
  5. Completion(A): know(A,Y)
  6. An *ask-if* would be appropriate to ask "is it past 5 o'clock?" and an *ask-all* would be more suitable to ask "what time it is?". It is not necessary that when X has free variables, an *ask-all* should be used. An *ask-if* with content *foo(X,Y)* makes perfect sense (for PROLOG "speaking" agents), if one wants to know if there exist X such that foo(X,Y) is true. But if the same expression is used with an *ask-all*, one expects something like [foo(a,b),foo(a,c)]. The use of *ask-all* assumes that the application's language provides built-in features for collections (such as a list in our PROLOG example).

- **stream-all(A,B,X)** Everything mentioned for *ask-all* holds for *stream-all*, too. A is interested in a series (possibly infinite) of statements of facts, as a response. The only difference is in the expected delivery format of the response. Either because the sender can not (or does not want to) process collections or due to receiver's inability to provide collections, the elements of the would be collection are to be delivered one by one (using *tell* since they are statements of facts for B). This performative also allows for responses to be delivered one at a time, as they are computed, thus permitting "pipelining" and efficient handling of very large, or even infinite, collections. The *eos* performative is to be used to mark the end of this multi-response (this is for A's benefit).

- **tell(A,B,X)**
  1. A states to be that A believes the content to be true.
  2. bel(A,X)[17]

[14]To provide an example, consider the situation that A asks B the time and B responds *12:00PM*. From our point of view, two speech acts take place (so two messages with the appropriate performatives have to be exchanged), the asking and the response to the asking. A precondition for B to respond would be that A asked him and for B that he still wants to know the time. For A to pose the question, there is a precondition that A wants to know the time (and possibly that A does not know the time already).

[15]For expository purposes we have made the simplifying assumption that agents know what other agents know, so they only ask them questions that they can answer. We have to do that for the sake of completeness of the subset we present here. In the full KQML version, there are ways for agents to learn what other agents can answer.

[16]the variables for which A wants values, are specified by the :aspect parameter in the KQML message

[17]This interprets *tell* as an assertive. If interpreted as a directive, it should be want(A,know(B,bel(A,X))).

3. Pre(A): bel(A,X) , know(A,want(B,know(B,Y))) A does not lie and B is interested in knowing. Y is any of the Y's mentioned in ***ask-if, ask-all, stream-all.***
   Pre(B): intend(B,know(B,Y))
4. Post(A): know(A,know(B,bel(A,X))) (optional)
   Post(B): know(B,bel(A,X))
5. Completion(A): know(B,bel(A,X))
6. The completion condition holds, unless a *sorry* or *error* suggests B's inability to acknowledge properly the *tell.*

- **deny(A,B,X)** Everything mentioned about *tell* holds for *deny,* if `bel(A,X)` is replaced with `NOT(bel(A,X))`.

For the next two performatives, we will need three extra predicates. We consider three stages in the handling of a *received* message. First, it is physically *received* (something we implicitly assume throughout the analysis[18]), second, *processed,* in the sense that it is a valid KQML message and will be delivered to the application for processing, and third, *delivered* to the application (technically, a handler function takes over) and the application will reply to that accordingly. We will use the predicates **receive**, **process** and **respond**, for those 3 stages, respectively. The predicates refer to the stages when completed and reference of each of one of those, assumes that the prior stages have occurred. Reference to the message being handled is made through *Id* (specified in the `:reply-with` parameter), and *Id* refers to the message as a whole.

- **error(A,B,Id)**
  1. A states to B that is not going to process the KQML message identified by Id.
  2. NOT(process(A,Id))
  3. Pre(A): receive(A,Id)
     Pre(B): NONE
  4. Post(A): know(A,know(B,NOT(process(A,Id))))
     Post(B): know(B,NOT(process(A,Id)))
  5. Completion(A): know(B,NOT(process(A,Id)))
  6. An agent might respond with an *error* if either he cannot successfully parse it as a KQML message, or the message is not an acceptable one, in the context of a "conversation" between the two agents.

- **sorry(A,B,Id)**
  1. A states to B that although he processed the message, he has no response to provide.
  2. NOT(respond(A,Id))
  3. Pre(A): process(A,Id)
     Pre(B): NONE
  4. Post(A): know(A,know(B,NOT(respond(A,Id))))
     Post(B): know(A,NOT(respond(A,Id)))
  5. Completion(A): know(B,NOT(respond(A,Id)))
  6. The best analogy for understanding the performative, is what happens when you are asked the time and you do not know what time it is.

[18] Addressing the issue of agent notification for messages delivered and received, is among those considered in KQML's implementation.

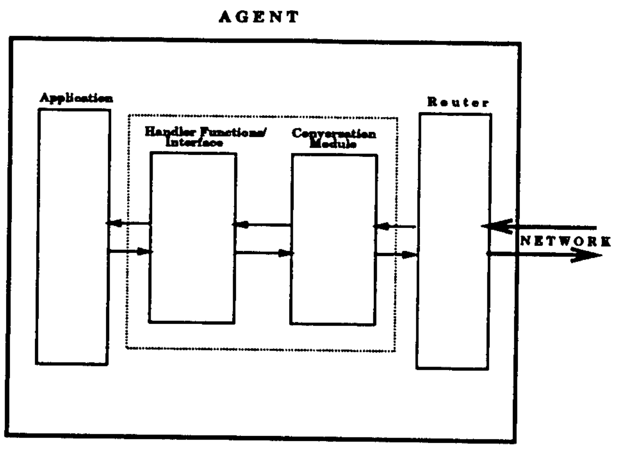


Figure 2: Logical architecture of a KQML speaking agent.

- **eos(A,B,Id)** This performative is somewhat unusual with respect to the other performatives mentioned because it is only purpose is to notify B that there are no more responses to a request for a multi-response query.

An example

Here, is an example of a conversation between agents with symbolic names Gen-DB and Gen1. Gen1 wants to know who are John's parents, and sends a ***stream-all*** to Gen-DB,

```
(stream-all :sender        Gen1
            :receiver      Gen-DB
            :language      Prolog
            :ontology      Genealogy
            :aspect        ``X''
            :reply-with    q1
            :content       ``parent(John,X)'')
```

and, in time, Gen-DB responds accordingly:

```
(tell       :sender        Gen-DB
            :receiver      Gen-1
            :language      Prolog
            :ontology      Genealogy
            :in-reply-to   q1
            :content       ``parent(John,Alice)'')

(tell       :sender        Gen-DB
            :receiver      Gen-1
            :language      Prolog
            :ontology      Genealogy
            :in-reply-to   q1
            :content       ``parent(John,Bob)'')

(eos        :sender        Gen-DB
            :receiver      Gen-1
            :in-reply-to   q1)
```

6 KQML semantics and architecture of KQML speaking agents

The logical architecture of a KQML speaking agent is shown in Figure 2. It is based in the KQML implementation developed at *UNISYS* [14] . We identify the following four parts:

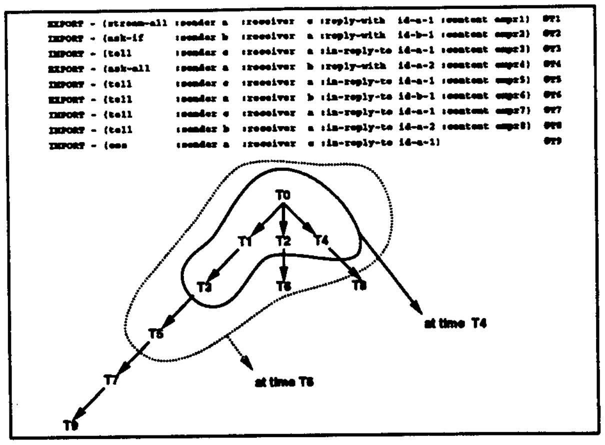
Application. In the case that this is a non–distributed application, the application programmer has to identify the points in the program where external information is needed. At those points, queries (in the general sense) have to be delivered to other applications (*agents*) that can answer them. The problem of what to send to whom can be attacked in several ways: 1) if the query-answering capabilities of each agent are well known in advance (like in [17] and in [10], where early versions of KQML were used for inter–agent communication) the application programer encodes the information in the distributed application so that when a query has to be answered by an agent in the outside world, the application knows in advance whom to query, 2) if the application operates in an environment mostly consisting of *open systems* [19, 18] the application can ask a *facilitator*[19] to appropriately deliver its query, or, 3) the application can ask the facilitator (or other agents) to take care of appropriately delivering the query or "discuss" the matter with the facilitator or other agents, in order to deliver the query on its own, or collect information from agents and facilitators, so that it can make its own decisions regarding the delivery of its queries (such an approach is also best suited for an open systems' community). KQML provides performatives to support the implementation of all the above mentioned approaches. Only in this last case, has the application programmer to provide code in order to use the extra information regarding other agents' capabilities.

Handler functions and Interface Module. The application programmer has to provide functions (called *handler functions*) that will process the various *performatives*. For example, for the *ask-if* performative the handler function (written in the application's native language) should access the application, check the truth status of the expression for the application and accordingly convey this information to the agent that made the query. Normally either the *tell* or the *deny* performative should be used in such a case. Through them, the application can state either that the expression is true, or that it is not known to be true or that the negation of the expression is true. In order for the application programmer to provide the handler functions he has to know the exact meaning of the various KQML performatives (here on called *semantics of the performatives*) and the policies that govern their use (*conversation policies*). We further refer to the conversation policies in Section 6.1.

Conversation Module. The conversation module lies between the router and the handler functions and interface module. Every message, either received by the agent or sent to some other agent, has to go through the conversation module. This module implements the conversation policies and checks all messages in order to decide if they are allowable continuations of the agent's current conversations with other agents. Our approach regarding the implementation of this module and its role and functionality in the overall architecture of an agent, is the subject of Section 6.1. We consider this module to be a partial implementation of the semantics.

Router. The router handles all KQML messages going to and from its associated application. Each KQML speaking software agent has its own router process but all routers are

[19]Facilitators are specialized agents that are designated with the task of facilitating the communication of agents by primarily holding information regarding the query answering capabilities of the agents in their network domain.



```
EXPORT - (stream-all :sender a  :receiver  c :reply-with  id-a-1 :content expr1)  @T1
IMPORT - (ask-if     :sender b  :receiver  a :reply-with  id-b-1 :content expr2)  @T2
IMPORT - (tell       :sender c  :receiver  a :in-reply-to id-a-1 :content expr3)  @T3
EXPORT - (ask-all    :sender a  :receiver  b :reply-with  id-a-2 :content expr4)  @T4
IMPORT - (tell       :sender c  :receiver  a :in-reply-to id-a-1 :content expr5)  @T5
EXPORT - (tell       :sender a  :receiver  b :in-reply-to id-b-1 :content expr6)  @T6
IMPORT - (tell       :sender c  :receiver  a :in-reply-to id-a-1 :content expr7)  @T7
IMPORT - (tell       :sender b  :receiver  a :in-reply-to id-a-2 :content expr8)  @T8
IMPORT - (eos        :sender a  :receiver  c :in-reply-to id-a-1)                 @T9
```

Figure 3: Sequence of imported and exported messages for agent "a".

identical. Routers are content independent message routers that provide the agent with a single point of contact for the rest of the network. It provides both client and server functions for the application and manages multiple simultaneous connection with other agents.

6.1 Implementation of the conversation policies for KQML performatives

The purpose of the conversation module is to assure that the agent is involved in meaningful conversations with other agents and keep track of them, despite the possibly asynchronous behavior of the agent. The conversation module is an implementation of the conversation policies that suggest: 1) which performatives start a conversation, and 2) which performative is to be used at any given point of a conversation. Figure 3 gives an example of a series of messages sent and received by an agent name *a* during some time period. Between times T_1 and T_9 messages from three different conversations are handled. The conversation module should handle something like that appropriately, keeping track of all three ongoing threads. Here is the scheme we suggest for doing that:

1. When a message (either to be imported to the application or to be exported to some other agent) reaches the conversation module, the module attempts to match it against one of the ongoing conversations.

2. If the message is not an acceptable continuation of some current thread, an attempt is made to start a new thread with it[20].

3. If no new thread can start with the current message, a message with the *error* performative will be sent to the sender (if the message is to be imported) or a signal is delivered to the application (if the message is to be exported).

We obviously have to define the acceptable threads of message exchanges and provide the module with the means to test them. We view the problem as one of parsing where the

[20]Not all performatives can be starting points for new threads. In the example of Figure 3 we consider the performatives *ask-if*, *ask-all*, *stream-all* to be acceptable starting points. We believe that eventually only *advertise* performatives (that are used to make known to other agents the capabilities of an agent) should be starting points.

grammar defines the conversation policies and messages are the terminals (so any series of messages in the *same* thread, is a "sentence" to be parsed). It differs from the usual parsing paradigm, though, in that the "sentence" might well be unfinished, meaning that the thread might not be complete (see Figure 3 as of time T_4 or T_6). Figure 1 shows part of an *Augmented Transition Network (ATN)* specification that can be used to perform this task for the subset of KQML performatives of Table 1, The ATN defines the conversation policies for this subset. For illustrative reasons the states where a message is to be imported are shaded. Not presented here are the *tests* and *actions* of the ATN that handle the necessary constraints among the various fields of the messages in order to define a thread[21] (conversation). The terminals are not known in advance. As mentioned before, the terminals are KQML messages with values for all their fields. Every time that a new message is to be handled by the module, the message becomes a potential new terminal. Referring to the described, top-level procedure, this new terminal is appended to the first "sentence" (thread) and an attempt is made to successfully parse the new sentence. If this fails, the second "sentence" is tried and so one.

An implementation of the conversation policy for a considerably extended set of KQML performatives is in progress. We believe that by providing a conversation module that can cooperate with the router the agent will be able to better handle asynchronous behavior, help the agent keep track of its business and provide the means to the application programmer to build more complex schemes of inter-agent communication (protocols like the Contract Net, see [28, 11]).

7 Software agents and KQML

We argue that our semantic approach does not constrain the kinds of software agents that can use KQML. Although the propositional attitudes represented by the predicates **know**, **want**, **intend** make reference to cognitive states for the agents, the cognitive states are necessary for understanding the performatives but not for using them. If the application designer wants to build a belief model to implement those mental states on top of the application, so that the application can better support a problem solving strategy or protocol, so be it. KQML does not require the existence of such a protocol or a cognitive model. Operators like **want** and **know** are materialized by virtue of use of a performative and are implied by the use of the language, rather, than the other way round, i.e., cognitive states implying a certain use of the language.

The really interesting question is how to interpret the **bel** operator in a given computer program. It depends on what the programmer ascribes to the program. For a PROLOG application (or a logic based system in general), **bel** might stand for whatever can be proved true in the system. Similar arguments can be made for other applications that adhere to the physical symbol-system hypothesis (frames, scripts, rule-based systems, semantic nets). How about a neural net? One can still suggest an interpretation that associates input and output. The same argument can be made for devices (such as thermostats), or databases. A functional approach to provide materialization for **bel** and common sense about how it should be interpreted for a given system, will do. If not, the `:ontology` slot can solve the problem. By choosing an interpretation from a library of such, the application can make known to its conversational partner what **bel** means for it.

It is our view that a belief model or a cognitive model is not necessary for a software agent to talk KQML. It can be useful to have one, either elaborate or primitive, but nothing more that a functional interpretation of the **bel** operator is necessary, for the semantics to make sense. All that is necessary is a program and handler functions. In between these two, many things can be included. A belief space, a cognitive model, a goal space, a problem solving strategy, or various combinations of the above. But none of that is mandatory for KQML to be used. In KQML, like in human communication, the personal agendas and beliefs of the agents suggest the choice of words, but the words themselves have an accepted meaning.

[21] The fields for the KQML subset presented here, are: :sender, :receiver , :reply-with and :in-reply-to.

8 Conclusion

We have presented an approach for the definition of the semantics of KQML. Although it is eventually the programmer that materializes the semantics through the handler functions that he writes, we have provided a framework that the programmer has to comply with. This framework is more detailed and formal than the existent so far ([2]), and will be supported by a software module (the *conversation module*) that will guide and restrict the possible uses of the languages primitives (performatives). The framework is based on *speech act theory* and primarily Searle's ideas.

We envision KQML as a general purpose communication language for software agents of all kinds. We believe that we offer an approach towards the semantics of the language that makes no commitments to application languages, agent models, programming paradigms, problem solving strategies and protocols. This approach stems from our belief that all those issues are peripheral to the communication language itself, which should be rich enough to accommodate a variety of propositional attitudes and offer enough leeway to implement all kinds of models, strategies and protocols, beneath the language. Ideally, KQML will rise to its full potential with the use of the results of the other research efforts of the KSE, because those efforts will provide the means for inter-agent understanding of the propositional context itself.

In the future, we intend to further apply our semantic approach to the full set of the up to date KQML performatives and refine the structure of a KQML speaking agent. All material related to KQML and the KSE can be accessed through the World Wide Web[22].

References

[1] External Interfaces Working Group ARPA Knowledge Sharing Initiative. KQML Overview. Working paper, 1992.

[2] External Interfaces Working Group ARPA Knowledge Sharing Initiative. Specification of the KQML agent-communication language. Working paper, June 1993.

[3] J.L. Austin. *How to do things with words.* Harvard University Press, Cambridge, MA, 1962.

[4] Alan H. Bond and Les Gasser. An analysis of problems and research in DAI. In *Readings in Distributed Artificial Intelligence*, pages 3–35. Morgan Kaufman Publishers, San Mateo, California, 1988.

[22] URL is http://www.cs.umbc.edu/kqml/

[5] John A. Campbell and Mark P. D'Inverno. Knowledge interchange protocols. In Y. Demazeau and J.-P. Muller, editors, *Decentralized A.I.: Proc. of the First European Workshop on Modelling*, pages 63–80. Elsevier Science Publishers B.V. /North Holland, Amsterdam, 1990.

[6] Philip R. Cohen and Hector J. Levesque. Intention = Choice + Commitment. In *Proceedings of the National Conference on Artificial Intelligence*, pages 410–415, July 1987.

[7] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[8] P.R. Cohen and H.J. Levesque. Persistence, intention, and commitment. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 33–69. MIT Press, Cambridge, MA, 1990.

[9] P.R. Cohen and C.R. Perrault. Elements of a plan-based theory of speech acts (1979). In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 169–186. Morgan Kaufman Publishers, San Mateo, CA, 1988.

[10] M. Cutkosky, E. Engelmore, R. Fikes, T. Gruber, M. Genesereth, and W. Mark. PACT: An experiment in integrating concurrent engineering systems. 1992.

[11] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 333–356, Morgan Kaufmann, 1988).

[12] E.H. Durfee, V.R. Lesser, and D.D. Corkill. Cooperative distributed problem solving. In A. Barr, P.R. Cohen, and E.A. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Vol. IV*, pages 83–147. Addison-Wesley Pub. Co., Reading, MA, 1989.

[13] Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *CACM*, 37(7):72–76, 1994.

[14] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing.* Ohmsha and IOS Press, 1994.

[15] L. Gasser. An overview of DAI. In Nicholas M. Avouris and Les Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 9–30. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[16] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *CACM*, 37(7):48-53, 1994.

[17] Mike Genesereth. Designworld. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 2,785–2,788. IEEE CS Press.

[18] Carl Hewitt. Offices are open systems. *Communications of the ACM*, 4(3):271–287, July 1986. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 321–330, Morgan Kaufmann, 1988).

[19] Carl Hewitt and Jeff Inman. DAI betwixt and between: From "intelligent agents" to open systems science. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6), December 1991. (Special Issue on Distributed AI).

[20] J. Hintikka. *Knowledge and Belief.* Cornell University Press, Ithaca, New York, 1962.

[21] Kurt Konolige. *A Deduction Model of Belief.* Pitman, London, 1986.

[22] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36 – 56, Fall 1991.

[23] R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA Knowledge Sharing Effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, San Mateo, CA, November 1992. Morgan Kaufmann.

[24] J. Searle and D. Vanderveken. *Foundations of illocutionary logic.* Cambridge University Press, Cambridge, UK, 1985.

[25] John R. Searle. *Speech Acts.* Cambridge University Press, Cambridge, UK, 1969.

[26] M.P. Singh. Towards a formal theory of communication for multiagent systems. In *Proceedings of the IJCAI'91*, 1991.

[27] M.P. Singh. A semantics for speech acts. (to appear in Annals of Mathematics and Artificial Intelligence), 1992.

[28] Reid G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 357–366, Morgan Kaufmann, 1988).

5. A SECURITY ARCHITECTURE FOR KQML

Intelligent Agent Integration Technology

Prepared by:
Chelliah Thirunavukkarasu
and
Tim Finin
Co-Principal Investigator
University of Maryland - Baltimore County
finin@cs.umbc.edu, (410)455-3522

This section includes the first comprehensive published paper on a security architecture for KQML. It was presented at the *ACM CIKM Intelligent Information Agents Workshop*, Baltimore, December 1995.

Abstract

KQML is a message protocol and format for agents to communicate with each other. In this paper we discuss the security features that a KQML user would expect and an architecture to satisfy those expectations. The proposed architecture is based on cryptographic techniques and would allow agents to verify the identity of other agents, detect message integrity violations, protect confidential data, ensure non-repudiation of message origin and take counter measures against cipher attacks.

1 Introduction

Agents, in their different manifestations as filter agents, personal agents, softbots, knowbots etc, have become an important topic and is one of the primary research areas in the academia and the industry. These agents, to successfully interoperate with each other and share their knowledge, need a common interface standard.

KQML, Knowledge Query and Manipulation Language [1] is such a message format and protocol, which enables autonomous and asynchronous agents to share their knowledge and or work towards cooperative problem solving.

With the popularity of internet and the possibilities offered by the agent technology we can expect an explosion of agents in the internet. For KQML to be an effective agent communication protocol in such an environment, it should provide some means for agents to communicate in a secure manner to protect the privacy and integrity of data and to provide for the authentication of other agents.

In this paper we discuss a security architecture which would enhance KQML and allow KQML speaking agents to authenticate senders, verify message integrity and have a private conversation.

2 Functional Requirements

We arrived upon the following requirements for a KQML security model based on the analysis of the security models for Privacy Enhanced Mail [4], Corba [3] and DCE [5]. Interested readers are referred to [2], for a thorough treatment of security threats and mechanisms to counter them.

- Independence of KQML and application semantics
  The security architecture should not depend on the semantics of KQML performative (i.e An *ask-all* from an agent will entail a *tell* or *sorry* from the receiver. The security model should not rely upon this kind of interaction semantics). The security model should be general and flexible enough to support different models of agent interaction (e.g contract net, electronic commerce).

- Authentication of principals
  Agents should be capable of proving their identities (they are who they actually claim to be) to other agents and verifying the identity of other agents.

- Preservation of message integrity
  Agents should be able to detect intensional or accidental corruption of messages.

- Protection of privacy
  The security architecture should provide facilities for agents to exchange confidential data.

- Detection of Message duplication or replay
  A rogue agent may record a legitimate conversation and later play it back to disguise its identity. Agents should be able to detect and prevent such playback security attacks.

- Non-repudiation of messages
  Non-repudiation of message is necessary to enforce accountability. An agent should be accountable for the messages that they have sent or received, i.e, they should not be able to deny having sent or received a message.

- Independence of transport layer
  The security architecture should not depend on the features offered by the transport layer. This is critical to facilitate agents to communicate across heterogeneous transport mechanisms and to extend the security model to accommodate embedded KQML messages.

- Non dependence on a global clock or clock synchronization
  The security architecture should not be clock dependent, as global synchronization of time is difficult to achieve and would lead to further security issues of its own. Further such a security model may have inherent security weaknesses [7].

- Prevention of message hijacking
  A rogue agent should not be able to extract the authentication information from an authenticated message and use it to masquerade as a legitimate agent.

- Authentication by crypto-unaware agents
  An agent need not have cryptographic capabilities to authenticate the sender of a message.

- Support for a wide variety of crypto systems
  Agents should be able to use different cryptographic algorithms. But for two agents to interact, they should have a common denominator. The security architecture should not depend on any specific cryptographic algorithm.

3 Architecture Overview

The proposed security architecture is based on data encryption techniques [9]. In tune with the asynchronous nature of KQML, the model expects a secure message to be self authenticating and does not support any challenge/response mechanism to authenticate a message after it has been delivered. The architecture supports two security models, basic and enhanced.

The basic security model supports authentication of sender, message integrity and privacy of data. The enhanced security model, in addition to the above, supports non-repudiation of origin (proof of sending) and protection from message replay attacks. The enhanced security model also supports frequent change of encryption keys to protect from cipher attacks.

3.1 Definitions

The following paragraphs define the cryptographic techniques used by this architecture and the new performative and the parameters that have been introduced to implement the architecture.

3.1.1 Data Encryption Keys

An agent that implements the proposed security architecture should have a master key, K_a, which it would use to communicate with other agents. This key can be based on a symmetric or asymmetric key cryptosystem.

If a symmetric key mechanism is used, we suggest that the agent, in addition to the general master key, also use a specific master key, $K_{a1,a2}$ for each agent that it communicates with, for better privacy and stronger authentication. If more than two agents share a single master key, any of those agents can masquerade as the other or eavesdrop on all the conversations between the agents sharing the key. If a master key is shared by more than two agents, the strength of security is directly related to the degree of trust between the agents.

If an agent does not share a master key, $K_{a1,a2}$ with another agent, it can use its master key, K_a, or can use the services of a central authentication server to generate such a key. The agents may use different keys in either direction of message flow i.e $K_{a1,a2}$ is created by *a1* and would be used when *a1* is sending a message to *a2* and $K_{a2,a1}$ is created by *a2* and would be used when *a2* is sending a message to *a1*.

If more than two agents share a single master key, any of those agents can masquerade as the other or eavesdrop on all the conversations between the agents sharing the key. If keys are shared by more than two agents, the strength of the security provided is directly related to the degree of trust between these agents.

If an asymmetric key mechanism is used, a unique key for each pair of agents is not necessary, as the agent can use the public key of its peer agent to encrypt the message and prevent eavesdropping. It can also use its private key to sign the message and prove its identity to its peer.

We assume that the agents know the master key of the other agents. We also suggest a secure mechanism to do master key lookup.

In the enhanced model, the agents use an additional key, the session key, to ensure privacy, message integrity and proof of identity. The session key can be symmetric or asymmetric and can be generated with the help of the authentication server. The session keys are set up by using a handshake protocol explained later. This handshake protocol requires the use of a master key to ensure security.

The agents can use either the session or master key for exchanging messages and must inform the other agent of the key that was used for encryption to ensure proper decryption.

When agents exchange keys, they encrypt them using the current session or master key. Keys are never exchanged in clear text form.

We recommend using the enhanced security model (if possible, as the enhanced model cannot be used under all circumstances) with an expensive master key and a cheap session key which is changed frequently.

3.1.2 Message Id

The message ID is used in the enhanced security model to protect agents from attacks by message replay. When the two agents establish a session key, they also exchange a message ID which the sender would use in the next message. Every message from an agent would carry a message ID and a new message ID for the next message. Each message ID is used only once to prevent replay and they are encrypted using the session or master key for security.

3.1.3 Message Digest

Each secure message generated using this architecture has a message digest or signature associated with it. The digest is calculated using a secure hash function like MD2, MD5 or SHS [9]. This hash function computes a digital fingerprint of the message (i.e acts as a "checksum" for the message). The sender then encrypts this digest using the session or master key and attaches it to the message.

This encrypted message digest forms the core of the security architecture. The receiver of a message uses this digest to verify the identity of the sender and the integrity of the message. The digest also protects the message ID field from being hijacked and used in a different message.

3.1.4 New KQML Parameters

The following new KQML parameters have been added to implement the security architecture.

:auth-master-key <boolean>
If T, the *:auth-digest* and *:auth-mesg* (if present) are encrypted using the master key. Else the session key is used. An agent would use the master key for encryption, if it does not share a session key with the receiving agent or if it does not know the receiver in advance. Under these circumstances, it could use this parameter to help the receiver in choosing the proper decryption key.

:auth-digest (<digest-type> <encrypted-digest>)
The *digest-type* specifies the hashing function used (MD4, MD5, etc.) to compute the message digest. The *encrypted-digest* is the message digest encrypted using the key specified by the *:auth-master-key* parameter. This parameter should be present to prevent message hijack, and to provide for sender authentication and integrity as-

surance.

:auth-mesg-id <string>
The value of this parameter is a pre-agreed random string. This parameter is required only in the enhanced security model to prevent message replay. After verifying the current message, to prevent a reuse of the same message ID, the receiver should reset its internal message ID field to the *:auth-new-mesg-id* or NIL.

:auth-new-mesg-id <encrypted-string>
The value of this enhanced model parameter is the message ID for the next message and is encrypted using the key specified by the *:auth-master-key* parameter. For effective prevention of message replay, this parameter should be present in each message.

:auth-new-session-key (<key-type encrypted-key>)
The value of this parameter specifies the session key for subsequent messages. If the value is T and the *:auth-shared-key* parameter is NIL, the current session key is destroyed and the sender will use the master key for subsequent messages. If the value is NIL, the session key, is left undisturbed. If it is not T or NIL, it is the new session key encrypted using the key specified by the *:auth-master-key* parameter. This parameter can be used to change the session key from time to time to protect from cipher attacks. Since the session key can be changed frequently, a cheap (computation-wise) cipher can be used as the session key.

:auth-mesg <encrypted-KQML-message>
This parameter is used only in *auth-private* performative. The value of this performative is a confidential KQML message which has been encrypted using the key specified by the *:auth-master-key* performative.

:auth-key-list ((<a1>, <key-type> <encrypted-key>) ...)
This parameter is used by an agent during master key registration with the authenticator. The value is a list of 3-tuple. The first element is the agent name, the second element is the key type and the third element is the encrypted master key. If the agent name is NIL, that key is shared with all the agents. If an agent uses asymmetric master key, the parameter contains only key agent name set to NIL.

3.1.5 New KQML Performatives

The following new KQML performatives have been added to implement the security architecture.

auth-link
The sender wishes to authenticate itself to the receiver and set up a session key and message ID.

auth-challenge
The sender challenges the identity of the receiver in response to an *auth-link*. The sender encrypts a random string using the master key $K_{s,r}$ or K_s and sends it as *:content.*

auth-link-request
The sender asks the receiver to send an *auth-link* and start the authentication process.

auth-private
The sender is sending a confidential message to the receiver. The *:auth-mesg* parameter contains the encrypted message and the *:auth-master-key* parameter specifies the encryption key. The *:auth-digest* parameter should be present to verify the identity of the sender and the *:auth-mesg-id*, *:auth-new-mesg-id* and *:auth-new-session-key* parameters may be present if enhanced security model is used.

auth-challenge-help
A crypto-unaware agent is enlisting the help of a trusted friend to construct a challenge message. The *:from* parameter will specify the agent to which the challenge message has to be sent.

auth-mesg-help
A crypto-unaware agent is enlisting the help of its trusted friend to authenticate a message with *:auth-digest* parameter. The message will contain the *:from* parameter whose value is the agent from which this message was received and the *:content* parameter's value will be the received message.

auth-key-request
The sender is requesting the authenticator to provide the master key for the agent specified in the *:from* field. If a master key pair exists for the two agents, the authenticator returns it. The *:content* parameter specifies the requested key's type.

This performative can also be used to generate a master or session key. A key is generated if *:to* is used instead of *:from* and it is an error to use both. If *:to* is used, the *:content* parameter is a 2-tuple. The first element is the key-type and the second element is a boolean flag which will be true, if a master key is requested. If a master key is requested, the generated key is added to the key list of the sender.

4 Basic security model

An implementation should support the following protocol to conform with the basic security model. This model supports authentication, integrity and privacy of data. If asymmetric keys are used for session and master keys, this model also supports non-repudiation of origin.

When R2D2 sends a secure message to C3PO, it would compute a message digest and encrypt it using the master key.

```
<performative>
    :sender R2D2
    :receiver C3PO
    :auth-master-key T
    :auth-digest (<digest-type> <encrypted-digest>)
    ...
```

Or, if R2D2 needs to send a confidential message to C3PO, it can encrypt the message and embed it in an *auth-private* performative.

```
auth-private
    :sender R2D2
    :receiver C3PO
    :auth-master-key T
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg <encrypted-KQML-message>
```

This model can be used when R2D2 does not know the recipient in advance e.g. broadcast and facilitation performative. Or if R2D2 and C3PO do not require prevention of message replay and can afford the cost of using the master key.

In the above message, the *:auth-digest* parameter can be used to verify the integrity of the message, authenticate the sender and ensure non-repudiation of origin (if the master key is asymmetric in nature). If the message has been corrupted, the message digest will not agree with the value of the *:auth-digest* parameter. Since the message digest is encrypted with the master key of the *:sender*, only the *:sender* or the agents with which the *:sender* shares the encryption key could have generated the message. If the master key is an asymmetric key, only the *:sender* could have generated the message, as only the *:sender* knows the private key that has been used for encryption. Note that we can only verify the identity of the generator (i.e. the message was encrypted by the *:sender* agent) of the message. This message can be a replay of a legitimate message previously sent by the generator.

5 Enhanced security model

This model in addition to the basic security, supports prevention of message replay, and stronger non-repudiation of message origin (if asymmetric keys are used). Even though non-repudiation can be achieved in the basic security model, we can only be sure that the message was generated by the sender, as a rogue agent can replay a message and we will not be able to detect it.

In the remainder of this section we will demonstrate how the new KQML performatives and parameters can be used to converse/communicate securely.

5.1 Self authentication

Agent R2D2 has cryptographic capabilities and would like to prove its identity to agent C3PO. The agents would follow the following handshake protocol to achieve it.

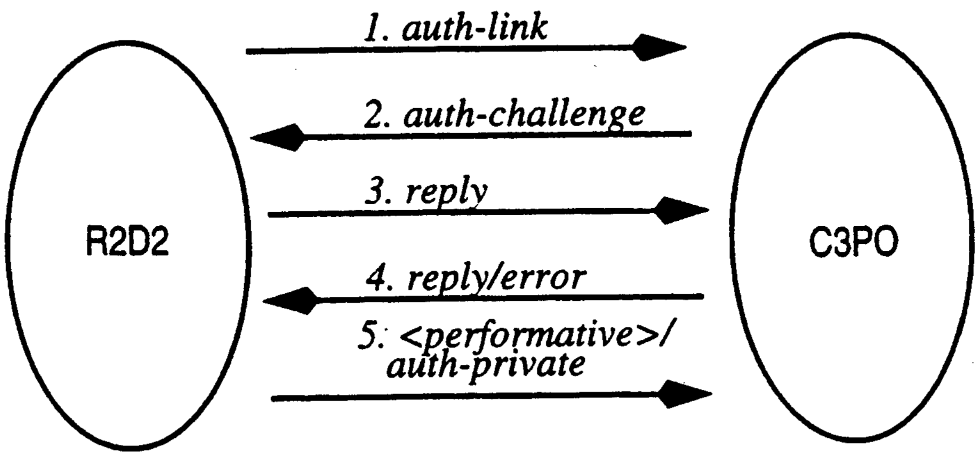


Figure 1: **Self authentication protocol**

1. R2D2 sends an *auth-link* performative to C3PO.

```
auth-link
    :sender R2D2
```

```
        :receiver C3PO
        :reply-with <expression>
```

2. If C3PO will not authenticate senders, it can respond with an *error*, otherwise it sends a *auth-challenge* with a random string encrypted using the master key. A random string is used to prevent message replay.

```
    auth-challenge
        :sender C3PO
        :receiver R2D2
        :in-reply-to <expression>
        :reply-with <expression>
        :content <encrypted-random-string>
```

3. R2D2 responds with a *reply* performative with the *:auth-digest*, *:auth-new-mesg-id* and *:auth-new-session-key* (if present) encrypted in the master key. The value of *:content* and *:auth-mesg-id* is the decrypted random string. The session key parameter is optional.

```
    reply
        :sender R2D2
        :receiver C3PO
        :in-reply-to <expression>
        :reply-with <expression>
        :auth-master-key T
        :auth-digest (<digest-type> <encrypted-digest>)
        :auth-mesg-id <mesg-id>
        :auth-new-mesg-id <encrypted-new-mesg-id>
        :auth-new-session-key (<key-type> <encrypted-key>)
        :content <random-string>
```

   Now, C3PO can verify if the sender is R2D2 by inspecting the random string. Only R2D2 (or in the case of symmetric key, one of the other agents which shares the same key) could have decrypted the random string as it was encrypted using the master key. The message digest can be used for non-repudiation if asymmetric keys are used.

4. C3PO responds with a *reply* or an *error* depending on the success of authentication.

5. Now, R2D2 can send an authenticated message to C3PO by using the session key or master key to encrypt the message digest and a non replayable message by using *:auth-mesg-id* and *:auth-new-mesg-id* parameters.

```
<performative>
    :sender R2D2
    :receiver C3PO
    :auth-master-key T or NIL
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-key>)
    ...
```

Or if R2D2 needs to send a confidential message to C3PO, it can encrypt the message and embed it in an *auth-private* performative.

```
auth-private
    :sender R2D2
    :receiver C3PO
    :auth-master-key T or NIL
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-key>)
    :auth-mesg <encrypted-KQML-message>
```

5.2 Authentication by request

R2D2 may expect some of the incoming messages from C3PO to be authenticated and it can initiate the authentication process by following the handshake protocol given below:

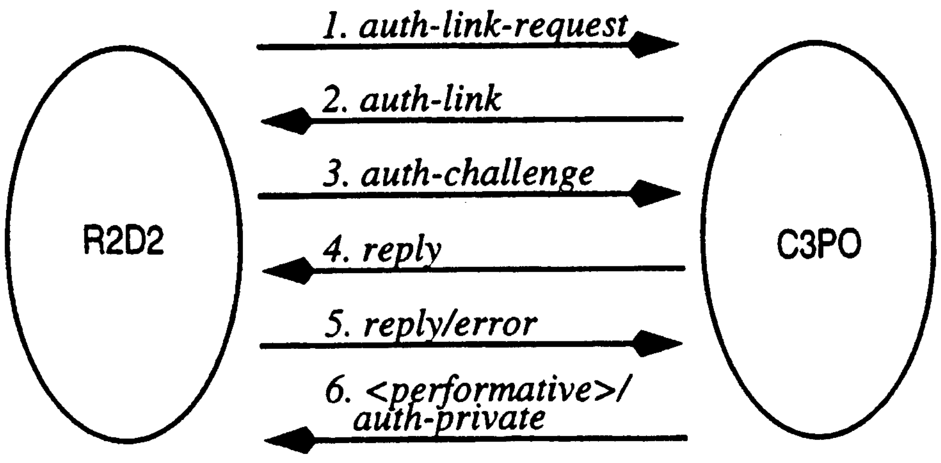


Figure 2: Authentication by request protocol

1. R2D2 can initiate the authentication process by sending an *auth-link-request* to C3PO.

```
auth-link-request
    :sender R2D2
    :receiver C3PO
    :reply-with <expression>
```

2. C3PO and R2D2 would then follow the steps outlined in *Self Authentication.*

5.3 Crypto un-aware agents

Agent Leia may not have crypto capabilities. But it trusts its friend R2D2 and R2D2 is prepared to authenticate messages on behalf of Leia. Since Leia does not have crypto capabilities, it will not accept *auth-private* performative. The agents would follow the handshake protocol given below to verify SkyWalker's identity.

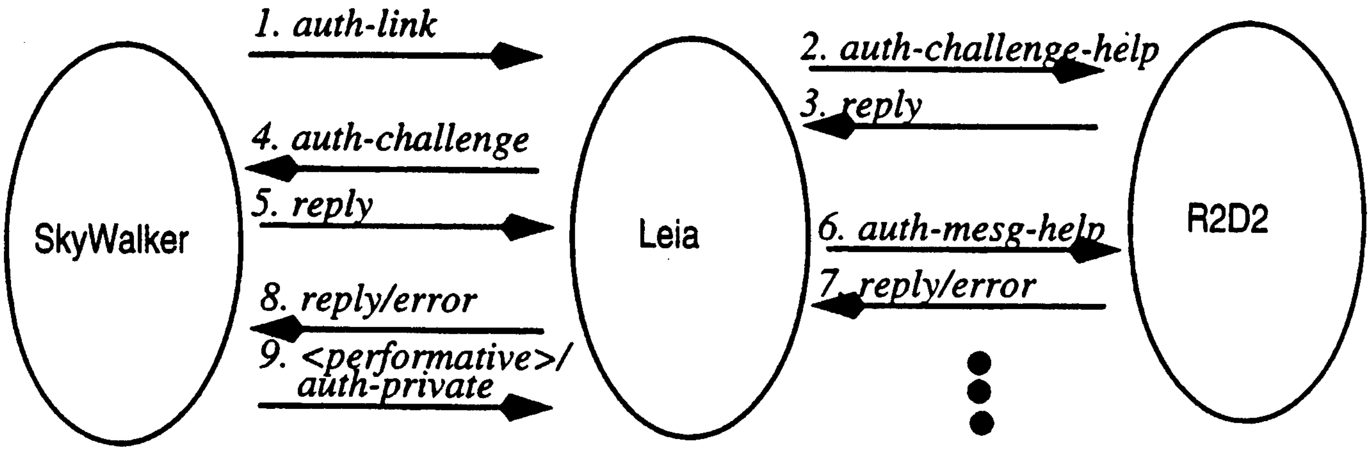


Figure 3: **Trusted friend protocol**

1. Agent SkyWalker sends Agent Leia an *auth-link* message to initiate the process of proving its identity to Leia.

```
auth-link
    :sender SkyWalker
    :receiver Leia
    :reply-with <expression>
```

2. When Leia receives an *auth-link* message from SkyWalker, Leia requests a challenge string from its trusted friend, R2D2.

```
auth-challenge-help
    :sender Leia
    :receiver R2D2
    :reply-with <expression>
    :from SkyWalker
```

3. R2D2 will generate a random string on behalf of Leia, encrypt it using the master key (shared by Leia and SkyWalker or Leia's master key, which R2D2 knows) and will forward it to Leia.

```
reply
    :sender R2D2
    :receiver Leia
    :in-reply-to <expression>
    :content (SkyWalker <encrypted-random-string>)
```

4. Leia will construct an *auth-challenge* performative and send it to SkyWalker. Subsequent performative from SkyWalker with an *:auth-digest* will be forwarded to R2D2.

```
auth-challenge
    :sender Leia
    :receiver SkyWalker
    :reply-with <expression>
    :in-reply-to <expression>
    :content <encrypted-random-string>
```

5. SkyWalker will respond with a secure *reply*.

```
reply
    :sender SkyWalker
    :receiver Leia
    :reply-with <expression>
    :in-reply-to <expression>
    :auth-master-key T
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-new-key>)
    :content random-string
```

6. Leia will wrap the response in an *auth-mesg-help* and send it to R2D2.

```
auth-mesg-help
    :sender Leia
    :receiver R2D2
    :reply-with <expression>
    :from SkyWalker
```

```
            :content message-from-SkyWalker
```

7. R2D2 will respond with a *reply* or an *error*.
8. Leia would forward the R2D2's reply to SkyWalker.
9. The handshake is now complete and SkyWalker can send secure performative to Leia, which Leia would verify with the help of R2D2.

6 Authenticator Agent

The authenticator acts as a repository of the agent's master keys. It can also generate session or master keys for the agents. The security architecture does not depend on the existence of an authenticator.

An agent and the authenticator share a master key which is known only to the agent and the authenticator. The master key may actually be a pair, one for the agent to send messages to the authenticator and the other for the authenticator to send messages to the agent.

The authenticator accepts only messages in the enhanced model, i.e., the messages should have an *:auth-mesg-id.* So, each agent should have established a secure link using *auth-link-request* and *auth-link* with the authenticator upon startup. It is the agent's responsibility to verify the identity of the authenticator and prove its identity to the authenticator.

6.1 Key lookup using the Authenticator

Agent Solo has received a message from Chewie and would like to know the master key used by Chewie. Solo uses the following protocol to get the master key from the authenticator.

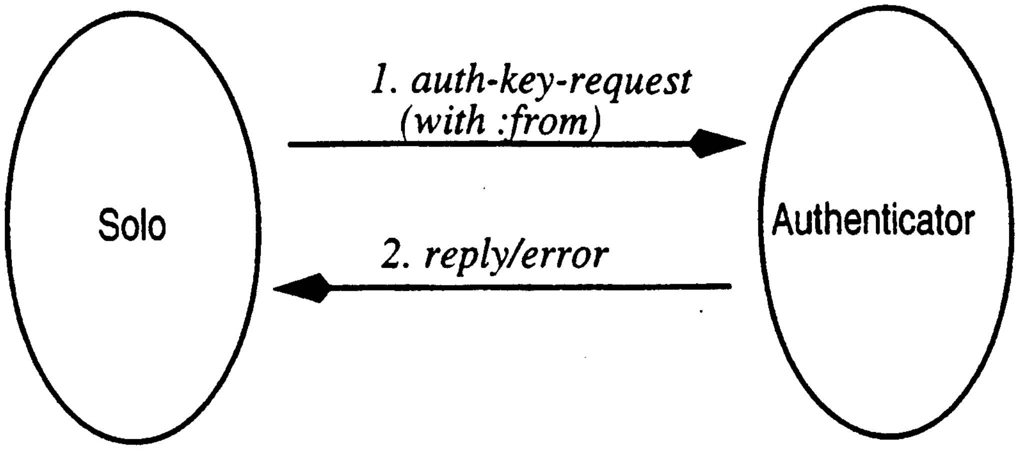


Figure 4: **Key request (lookup) protocol**

1. Agent Solo would send an *auth-key-request* to the authenticator to lookup the master key used by Chewie to send out messages. The *:content* parameter contains the requested key-type.

```
auth-key-request
    :sender Solo
    :receiver Authenticator
    :reply-with <expression>
    :from Chewie
    :auth-master-key T or NIL
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-new-key>)
    :content <key-type>
```

2. If Chewie had previously registered a master key for communication with Solo, the authenticator will return that key in a *reply* performative. If there is no such key, the authenticator will reply with an *error*.

```
reply
    :sender Authenticator
    :receiver Solo
    :in-reply-to <expression>
    :auth-master-key T or NIL
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-new-key>)
    :content (Chewie <key-type> <encrypted-master-key>)
```

6.2 Key creation using the Authenticator

Agent Solo would like to send a secure message to Chewie and needs a session or master key for that purpose. It can send an *auth-key-request* to the authenticator to create such a key. If a master key has been requested, the authenticator would store the key in its database.

A master key creation would not be necessary if asymmetric keys are used as a single master key per agent is suffice to talk securely to all the agents. Further, non-repudiation of message origin is not possible if the authenticator knows the private key.

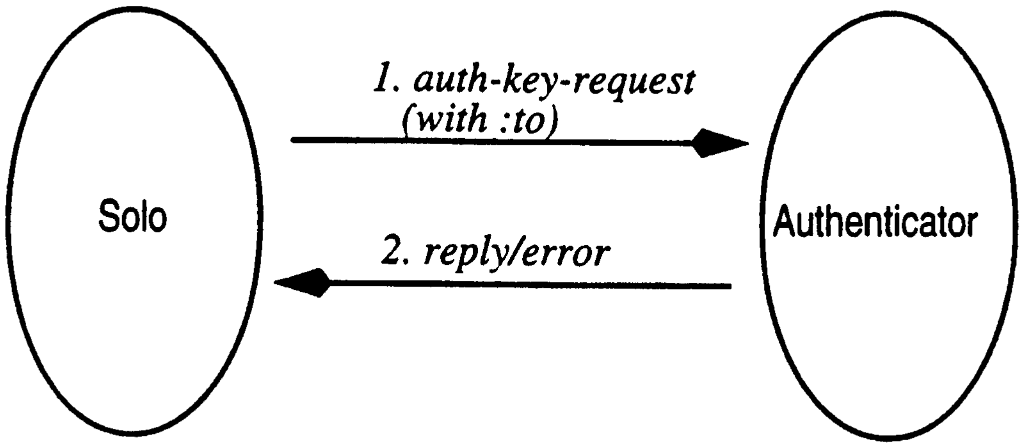


Figure 5: **Key request (creation) protocol**

1. Agent Solo would send an *auth-key-request* to generate a master or session key to send messages to Chewie. The *:content* parameter is a 2-tuple. The first element is the requested key's type and the second element is T if a master key is requested.

```
auth-key-request
    :sender Solo
    :receiver Authenticator
    :reply-with <expression>
    :to Chewie
    :auth-master-key T or NIL
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-new-key>)
    :content (<key-type> T-or-NIL)
```

2. Authenticator creates a key and sends it in a *reply* performative. If the requested key is a master key, the key is added to Solo's key list. If the authenticator is not able to create the key for whatever reason. it responds with an *error* performative.

```
reply
    :sender Authenticator
    :receiver Solo
    :in-reply-to <expression>
    :auth-master-key T or NIL
    :auth-digest (<digest-type> <encrypted-digest>)
    :auth-mesg-id <mesg-id>
    :auth-new-mesg-id <encrypted-new-mesg-id>
    :auth-new-session-key (<key-type> <encrypted-new-key>)
    :content (Chewie <key-type>
    <encrypted-master-or-session-key>)
```

6.3 Key registration with Authenticator using KQML

Agent Yoda would like to register its master keys with the authenticator.

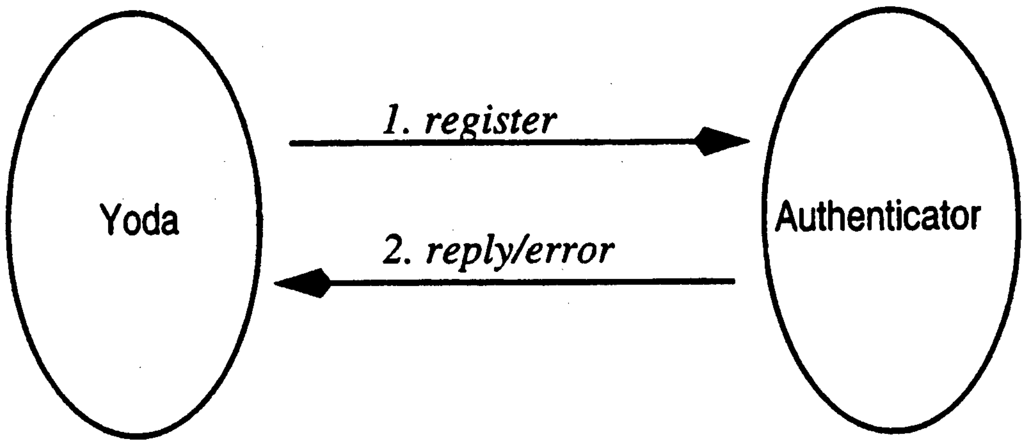


Figure 6: **Key register protocol**

1. Yoda would send a secure register with the keys in the *:auth-key-list* parameter. The keys are encrypted using the key specified by the *:auth-master-key* parameter. The agent can also use this performative to change the master key that it shares with the authenticator.

```
register
```

```
:sender Yoda
:receiver Authenticator
:reply-with <expression>
:auth-master-key T or NIL
:auth-digest (<digest-type> <encrypted-digest>)
:auth-mesg-id <mesg-id>
:auth-new-mesg-id <encrypted-new-mesg-id>
:auth-new-session-key (<key-type> <encrypted-new-key>)
:auth-key-list ((<agent> <key-type> <encrypted-key>)
...)
:ontology tcp-address-ontology
:content (tcp-host tcp-port)
```

2. If the key registration is successful, the authenticator responds with a *reply* else with an *error*.

6.4 Initial key registration with the authenticator

Agent Yoda is starting up for the first time and would like to register the master key that it shares with the authenticator. This can be achieved either using KQML or some other external mechanism.

If symmetric keys are used, KQML cannot be used to register the initial key as there is no master key to encrypt the key. If asymmetric keys are used, the initial master key is encrypted using the authenticator's public key. Even if asymmetric keys are used, there is a security problem. A rogue agent, agent DarthVader may know that agent Ben respects performative from agent Yoda. Agent DarthVader may also find out that Yoda has not registered with the authenticator and therefore the authenticator does not know the existence of such an agent. Now, DarthVader can register itself as Yoda. If this type of masquerading can be an issue, KQML should not be used for the initial registration.

The protocol would be same as the key register process. The *:auth-key-list* parameter will contain only one key pair and the agent name would be NIL as this is an asymmetric key and it is suffice to use a single asymmetric master key for all the agents.

If the authenticator does not have any entry for Yoda, it accepts the registration and adds it to its database and sends a *reply*.

7 Limitations of this model

- An agent can send out authenticated messages if and only if it has crypto capabilities (A fair limitation).
- The security architecture introduces state information. Agent Emperor has to keep track of the next message ID and optionally the next session key that will be used by agent DarthVader. The agents can choose not to use this feature if they are not concerned with message replay attack and cipher attack.
- Messages delivery must be reliable and in order. (A fair limitation considering that KQML itself assumes that).
- The model does not support non-repudiation of receipt of messages. This would be difficult to implement due to the asynchronous nature of KQML and can be done only at the application level.
- There is no support for a mechanism to exchange credentials. Lets say that agent Emperor trusts agent DarthVader and would like to delegate DarthVader to act on its behalf. There is no way for DarthVader to take up Emperor's credentials.
- The model does not support replay detection if *:auth-mesg-id* and *:auth-new-mesg-id* are not used. These parameters cannot be used if the recipient is not known in advance.
- The model should be enhanced to support the use of the Crypto APIs recommended by NSA (GSS, GCS and Cryptoki) [8], especially for the key-type and digest-type values.
- The architecture does not address traffic analysis by rogue agents. We feel that traffic analysis is best handled at the link/transport layers.

8 Conclusion

The proposed security model addresses privacy, authentication and non-repudiation (if asymmetric key mechanism is used for the master and session keys) in agent communication. It does not fully address the issue of message replay, especially if the recipient of a performative is not known in advance.

The security model depends on the strength of the crypto algorithm, message digest function and the random number generator used by the agent for its effectiveness.

LORAL and UMBC have a KQML implementation [10], and we shall discuss the modification required to that implementation to provide secure services to the agents. In the LORAL and UMBC architecture, each agent application is associated with its own separate router process. The routers used by all the agents (under this implementation) are identical; a copy of the same program. The router process handles all KQML messages going to and from its associated agent. The security enhancement can be easily added to this KQML implementation by modifying the router to be security aware, without involving any major change to the agent application.

The agent application only needs to specify the degree of security (any combination of provide for message authentication, protect from replay attack, send a confidential message and sign the message-non-repudiation of origin) of an outgoing message. The router would handshake with the receiving agent and secure the message to the extent possible (the receiving agent may not support asymmetric key cryptography, ***auth-private*** performative etc or the router may not know the receiving agent of the embedded message if it is sending out a broadcast or facilitation performative).

Similarly, when the router receives an authenticate request from another agent, it can handle the handshake itself, without involving the agent application. When the router receives a message from another agent, it would tag the message with a security level (confidential, authenticated, etc.). The agent application can decide to process or ignore the message based on the message's security level.

A similar approach can be followed to add security enhancement to most other KQML implementations. Most implementations would provide a library with at least a basic send and receive primitive to send and receive KQML messages. These primitives can be modified to add the authentication information to the outgoing messages or process the authentication information in the incoming messages. The implementations can use one of NSA recommended crypto APIs [8] for cryptographic capabilities. These APIs provide support for asymmetric and symmetric key cryptography, message digest, key generation etc. The use of a standard API would help agents using different KQML implementations to interact without any incompatibility problems.

9 Acknowledgments

I thank Mr. Arulnambi Kaliappan, Graduate Student, Computer Science, UMBC for his suggestion of agent names and and Mr. Senthil Periaswamy, Graduate Student, U of South Carolina for the stimulating discussions on possible security threats and attacks. Last but not least, I express my gratitude to my advisor Dr. Timothy Finin, for his support and guidance. But for his encouragement and enthusiastic support, this work would not have been possible.

References

[1] Draft specification of the KQML agent communication language, Tim Finin, Jay Weber et al. Jun 15 1993, http://www.cs.umbc.edu/kqml/kqmlspec/spec.html

[2] Security Mechanisms in High-Level Network Protocols, Victor L.Voydock, Stephen T. Kent, ACM Computing Surveys, Vol.15, No. 2, 135-171, Jun 83

[3] OSTF RFP3 Submission, Corba Security, OMG Document Number 95-3-3, Mar 8 1995, http://www.omg.org/docs/95-3-3.ps

[4] Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures, J. Linn, Oct 02 1993, http://ds.internic.net/rfc/rfc1421.txt

[5] Security in a Distributed Computing Environment, OSF-O-WP11-1090-3, http://www.osf.org/comm/lit/OSF-O-WP11-1090-3.ps

[6] Project Athena Technical Plan, Section E.2.1, Kerberos Authentication and Authorization System, S.P.Miller, B.C.Neuman, J.I.Schiller and J.H.Saltzer, Oct 27 1988, ftp://athena-dist.mit.edu/pub/kerberos/doc/techplan.PS

[7] Limitations of the Kerberos Authentication System, S.M. Bellovin, M. Merritt, Proceedings of the Winter 1991 Usenix Conference, Jan 1991, ftp://research.att.com/dist/internet_security/kerblimit.usenix.ps

[8] Security Service API: Cryptographic API Recommendation, NSA Cross Organization, CAPI Team, Jun 12 1995, http://www.omg.org/docs/95-6-6.ps

[9] RSA Labs' frequently asked questions (FAQ), http://www.rsa.com/rsalabs/faq

[10] Software Design Document for KQML, Revision 3.0, Mar 1995, LORAL Corporation, Paoli PA, USA

6. AGENT NAMES FOR A KQML AGENT SYSTEM

Intelligent Agent Integration Technology

Prepared by:
Tim Finin, Chelliah Thirunavukkarasu and Anupama Potluri
Co-Principal Investigator
University of Maryland - Baltimore County
finin@cs.umbc.edu, (410)455-3522

Donald P.McKay and Robin McEntinre
Lockheed Martin
Donald.P.McKay@lmco.com and Robin.A.McEntire@lmco.com

6.1 Introduction

Agents need to talk to other agents. If you are an agent A and there is another specific agent B that you want to send a message to, how do you manage it? Well, clearly there is a need for some kind of referential expression that A can use for B and which can be given to the underlying machinery which will convey the message to B. One solution is to use an expression that locates the agent with respect to the message transport system. Examples of such "transport address names" would be a structure which contains an IP address and a port number, or a URL, or an email address for the TCP/IP, http and SMTP protocols. This is a common practice in many of our primitive agent systems today.

Another approach allows agents to use one of more symbolic names and to provide some kind of mechanism by which names can be registered and associated with their appropriate "transport address name". This approach is only slightly more sophisticated than the first. The name registration can be done in any of several ways, such as hand coding the associations into all of the agents, or broadcasting the associations over the transport mechanism or assuming the use of "communication facilitator" type agents.

The KQML language and protocol includes special commands (the register and unregister performatives) by which agents can announce the symbolic names by which they wish to be known. Special agents (commonly known as "communication facilitators") traffic in this knowledge and provide a name registration and resolution service. In the Loral/UMBC "KQML Agent Technology Software" (KATS) architecture, this name registration and resolution is handled automatically by a generic router sub-agent attached to each agent. From the agents perspective, all it has to do is to specify the set of symbolic names it wished to be known by. The router sub-agent automatically contacts the local "facilitator agent" [4] to register the agent by its symbolic names.

For example, suppose the agent named A wants to send a query to agent B. It passes a KQML form like

- (ask-one :from A :to B :content ...)

to its router sub-agent (call it r(A)). This router is responsible, among other things, for resolving the agent name B into an address that can be given to the transport layer for delivery. In KATS, the router checks it's cache to see if it knows how to deliver a message to an agent named "B". If it does, it ships the message out. If not, it sends a KQML query to the local agent name server, asking for the address of an agent named "B". Upon receiving the information, it adds it to its cache and sends off the message.

There are additional wrinkles, of course, such as how to determine when a cache entry is stale and needs to be flushed, but this describes the current arrangement.

6.2 The Problem

Although this approach works quite well as far as it goes, it just does not go very far. The problem is that it only supports communication between two agents if they both register with a common agent name server. There are several possible solutions. All agents could use a single master name server possibly located deep under Cheyenne Mountain. Another approach is to have the name servers share their registration databases. Still a third, and more general, technique involves having the name servers use a distributed protocol to seek out the contact information on non-local agents. We next describe our protocol for such a distributed agent name resolution scheme.

6.3 Distributed agent-name resolution

We propose to organize agents into "agent domains" in much the same way that the Internet is organized into "host domains". An "agent domain" can be thought of as a collection of agents that are associated with a particular set of facilitator-class agents. In particular, every agent domain must have an "agent domain name server" (or "agent name server" or ANS for short) running. There may be other facilitator-class agents, such as brokers, associated with the agent-domain.

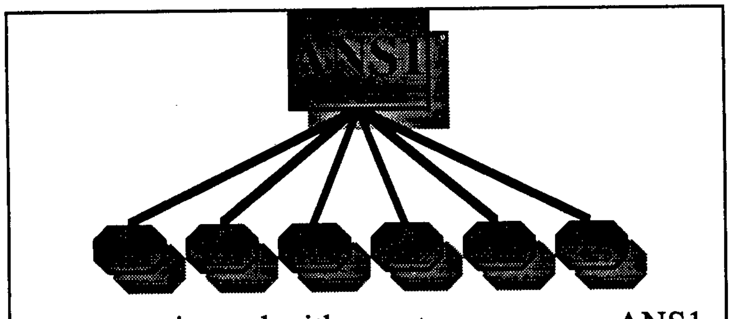


Figure 1 -- a set of agents are associated with an agent name server by sending it a KQML "register" performative.

Agent domains will be organized into a hierarchy. Agents will register with an ANS, as shown in figure one. An ANS, being an agent itself, will register with a "parent ANS", resulting in a hierarchy, as shown in Figure Two. Each agent will have one or more local names. An agent can also be referred to by its "domain qualified name". For example, consider the agent-domain hierarchy in Figure Three.

One possibility we might consider is just to "piggy-back" on the existing Internet host structure. For example, why not refer to the agent "colossus" running on the machine "cujo.cs.umbc.edu" as "colossus@cujo.cs.umbc.edu" and assume a standard port for KQML speaking agents. This idea is attractive in that it makes efficient use of a well thought out and implemented architecture. However, there are several problems which argue against this. The primary difficulty is that we do not want to tie KQML and agent communication in general to a single transport mechanism. Current research groups are using a variety of mechanisms to carry KQML messages -- TCP/IP, SMTP, CORBA objects, and HTTP. We would like to continue to keep KQML flexible in this regard. A consequence of this is that we need a general mechanism for naming agents that is independent of the transport mechanism.

6.4 What should agent names look like?

We propose a naming scheme similar to the one used for hosts on the Internet. Every agent will have one or more local names optionally followed by a domain qualifier. A local name can be any non-zero length sequence of characters chosen from the character set

- {a-z,A-Z,0-1,-,_,.,+,#}

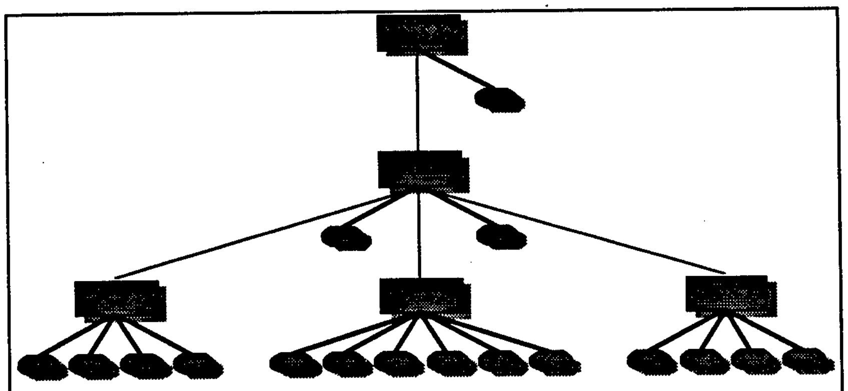


Figure 2 -- Agent name servers are organized into a hierarchy through the registration process.

A domain qualifier begins with the character "." and consists of one or more agent domain names separated with a "." character. Thus a fully qualified agent name has the structure:

- <local name> . <domain1>.<domain2>. <domain3>...<domainN>

The following would all be valid names for an agent with the local name "colossus" registered in the "umbc.edu." agent domain (and assuming that it is in turn registered in the "edu." domain which is in the top-level "." domain.)[5]

- colossus
  colossus.umbc
  colossus.umbc.edu.

Furthermore, we propose a correspondence between the names of agent domains and agent domain servers. Thus in the above example, agent *colossus* is registered with the ANS with local name *umbc* which is registered with the ANS with local name *edu* which is registered with the global ANS. Thus, the fully qualified name of an agent could be defined by its local name followed by a "." followed by the fully qualified name of its official agent name server.

There are obvious alternatives to the syntax we are proposing which would model agent names after email addresses (e.g., colossus@umbc.edu) or URLs (e.g., kqml://umbc.edu/colossus). There are several arguments against using either of these existing formats. One argument that applies to both is that we would like to avoid confusion about what a particular address means, e.g., is it the name of an agent, a reference to a document, or a reference to a mailbox. One might think that such a confusion could be a feature rather than a bug, since each of these might be a very reasonable way to think of and interact with an agent. However, there is clarity to be gained by separating the concept of a abstract reference to an agent that is independent of communication channel and a reference to an agent that implies a means of communication. The email style address has an advantage of using a special character (the @) to separate the "local name" from the "host name". When standards for SMTP were being developed, this was quite useful since it provided a mechanism to support gateways between email systems that used very different protocols[6].

6.5 How agent names are resolved

The process of resolving a name is similar to the one used for the Internet DNS. One difference is that agent with a given name can have many addresses -- one for each transport mechanism that it can use. Thus, the agent_address is a function from agent names and agent transport types to transport addresses. We assume that an agent can be referred to using its fully qualified name[7] or any non-ambiguous abbreviation.

Suppose agent A1 wants to resolve the fully qualified name N2 into an address for transport type T2. The process starts when A1 asks its agent name server.8 The query is passed up the hierarchy of agent name servers as long as the address is not known and N2, is not recognized as being the name of some descendant. If an agent name server gets the query and knows the address, the process stops and a response is sent to A1. If the root of the agent domain hierarchy is reached and the address was not found, the process fails and an appropriate error message is sent to A1. If an agent name server recognizes that N2 is the name of some descendant, it is passed down to the appropriate immediate child agent name server. This process continues until we find an agent name server that knows the address or we recognize that we can go no further. In this latter case, the process fails and an appropriate error message is sent to A1.

Resolving partially qualified agent names follows a very similar process. There are a number of details that must be decided on in standardizing this name resolution protocol -- i.e., whether answers are sent directly back to the agent initiating the query or passed back through the hierarchy and cached along the way. These details should only effect the performance of the name resolution process.

6.6 Taking names seriously

Agents should take names seriously. What we mean by this is that application agents should always refer to other agents by their names, and not the underlying transport addresses, if known.[9] Agents should leave the resolution of these names into transport addresses up to specialized agents (e.g., agent name servers) and sub-agents (e.g., routers). Adapting this convention will directly support the concept of a *proxy agent*, the use of logical agent services, and other important notions. We will discuss the concept of a proxy agent in more detail and sketch how it can be easily implemented by adopting a few simple conventions for agent name servers.

6.7 Proxy agents and their protocols

A *proxy agent* is an agent that handles all of the incoming and outgoing messages (perhaps with respect to a particular transport mechanism) for another agent. A simple proxy mechanism can be used to provide a number of services:

- *firewall gateways* -- agents which are behind a security firewall and use a proxy agent to communication to agents outside the firewall.
- *protocol gateways* -- An agent which is unable to send or receive messages via a particular transport mechanism (e.g., email) can still communicate with agents who only use that mechanism by having a proxy agent to mediate between two transport mechanisms.
- *message processing* -- The proxy can provide a processing service, such as logging incoming or outgoing messages, without altering the stream.
- *filtering and annotating* -- The proxy can alter the stream by filtering out certain incoming messages, blocking outgoing messages to particular destinations, annotating incoming messages, etc.
- *agent composition* -- A proxy agent facility allows one to develop a notion of "agent composition" similar to functional composition.

As an example, suppose we have two agents A and B, both of which use the agent name server F. A has proxy agent p(A) and B has proxy agent p(B). Suppose A wants to send a message to B. The following events take place:

- 1. A hands off the message to its router subagent r(A).
  2. r(A) asks F for B's address.
  3. f gives r(A) the address of p(A), A's proxy.
  4. r(A) delivers the message to p(A) but the :TO field equals b.
  5. p(A), knowing that it is a proxy for a (possibly among others) and noticing that it has received a message from a with the :TO field of B, understands that the message is not really intended for it, and asks its router r(p(A)) to deliver it to B.
  6. r(p(A)) asks F for the B's address.

7. F gives r(p(A)) the address of p(B) -- B's proxy).
8. r(p(A)) delivers the message to p(B) with the :TO field equals B.
9. p(B), knowing that it is a proxy for B (possibly among others) and noticing that the :TO field is B, understands that the message is not really intended for it, and asks its router r(p(B)) to deliver it to B.
10. r(p(B)) asks F for the address of B.
11. F recognizes that p(B)) is B's proxy so it gives p(B) the real address of B.

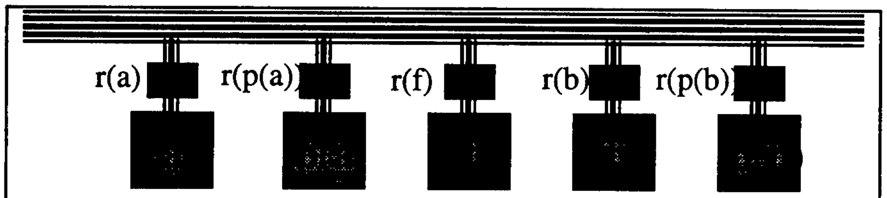


Figure 3 -- A conversation among five agents and their sub-agents.

This example demonstrates the use of proxy agents for both outgoing messages and messages. The proxy agents may do some additional processing of the messages they get, of course, like logging or traffic analysis, etc. The scenario above is the worst case in that it assumes all of the router subagent caches are empty. Subsequent communications would find the caches filled, so the agent name server would not have to be involved.

Implementing the concept of a proxy agent is rather trivial once we have agent name servers and agents who contact agents by name rather than by transport address. First, if an agent P is willing to serve as a proxy agent, it has to be able to provide some of the functionality that an agent name server does. Second, if A wishes to use P as a proxy for transport mechanism T, it must (1) get permission from ask P for this and (2) unregister with A's agent name server for transport T (if it was so registered). Third, P should register with A's agent name server *in A's name* for transport T. Good design dictates that all of the agents involved should also explicitly "know" that P is acting as A's proxy with respect to messages carried by transport mechanism T.[10]

6.8 Changes to KQML and standard utility agents

This naming scheme will not require any major changes to KQML such as the addition of new performatives or new parameters. It will have an impact on the form of the register performative and on the standard agent ontology and on the protocols used by standard utility agents such as an agent name server and a router. This, in turn, will effect the protocols that all agents who use these standard utility agents follow.

An agent name server will have to store more information about the agents that are registered with it and will have to handle some additional performatives. When an agent registers with an agent name server, it should provide a set of symbolic names it will respond to and a set of transport type/address pairs. Authentication information may be provided as described in (Thirunavukkarasu, Finin and Mayfield 95). A standard agent name server must handle requests to register and unregister from agents as well as various kinds of queries against its registration database.[11]

In reaching a consensus on the precise details of how to add these changes to KQML we will have to choose what aspects are expressed by adding to or modifying the basic components of KQML (i.e., performatives and parameters and their semantics) and which are expressed by extending the common "*agent ontology*" that is assumed by KQML.[12]

6.9 Conclusions

We have discussed the problem developing a global naming scheme for software agents and how such names can be resolved into usable addresses. We have assumed an agent environment which (1) is dynamic with agents being created and destroyed frequently; (2) undergoes re-organizations with agent groups and sub-groups forming and disbanding; and (3) supports agent communication by any of several transport mechanisms such as TCP/IP, email, http and distributed object systems. We proposed the use of *agent domains* which are organized into an *agent domain hierarchy.* Agent name resolution will be done *agent name server* agents which use a distributed protocol similar to that used by Internet *domain name servers.* This approach supports the definition of *proxy agents* which have a variety of uses. We have briefly discussed how this proposal would impact the KQML agent communication language and protocol and describe an ongoing implementation of a generic *KQML Agent Name Server* and its integration into the KATS framework.

6.10 Bibliography

Paul Albitz & Cricket Liu, *DNS and BIND*, O'Reilly, 1992, ISBN:1-56592-010-4.

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. *KQML: an information and knowledge exchange protocol.* In International Conference on Building and Sharing of Very Large-Scale Knowledge Bases, December 1993. A version of this paper will appear in Kazuhiro Fuchi and Toshio Yokoi (Ed.), "Knowledge Building and Knowledge Sharing", Ohmsha and IOS Press, 1994. An online copy can be obtained from "http://www.cs.umbc.edu/kqml/papers/kbks.ps".

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. *The KQML information and knowledge exchange protocol.* In Third International Conference on Information and Knowledge Management, November 1994.

Tim Finin, Yannis Labrou, and James Mayfield, *KQML as an agent communication language*, invited chapter in Jeff Bradshaw (Ed.), ``Software Agents", MIT Press, Cambridge, to appear, (1995).

Rich Fritzson, Tim Finin, Don McKay and Robin McEntire. *KQML - A Language and Protocol for Knowledge and Information Exchange*, 13th International Distributed Artificial Intelligence Workshop, July 28-30, 1994. Seattle WA.

Mark R. Horton, *What is a domain?,* available on-line as <http://www.dns.net/dnsrd/docs/domain.ps>.

Yannis Labrou and Tim Finin. *A semantics approach for KQML--a general purpose communication language for software agents.* In Third International Conference on Information and Knowledge Management, November 1994. Available on-line as http://www.cs.umbc.edu/kqml/papers/kqml-semantics.ps.

R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. *The DARPA knowledge sharing effort: Progress report.* In B. Nebel, C. Rich, and W. Swartout, editors, Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92), San Mateo, CA, November 1992. Morgan Kaufmann.

R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. *Enabling technology for knowledge sharing.* AI Magazine, 12(3):36--56, Fall 1991.

James Mayfield, Yannis Labrou and Tim Finin. *Evaluation of KQML as an Agent Communication Language*, the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages.

7. ONTOLOGICAL MEDIATION

Intelligent Agent Integration Technology

Prepared by:
Alistair E. Campbell
and
Stuart C. Shapiro
Co-Principal Investigator
State University of New York at Buffalo
shapiro@cs.buffalo.edu, (716)636-3935

This paper originally presented at the International Joint Conference on Artificial Intelligence – Workshop on Basic Ontological Issues in Knowledge Representation.

Ontologic Mediation : An Overview

Alistair E. Campbell and Stuart C. Shapiro
Department of Computer Science
and Center for Cognitive Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, New York 14260
{aec,shapiro}@cs.buffalo.edu

1 Introduction

One of the motivations behind the Knowledge Sharing Initiative [Neches *et al.*, 1991] is to enable effective communication between software knowledge agents so that independent, heterogeneous agents can share their knowledge with one another. Hence, many researchers are developing techniques, protocols, languages, etc. to facilitate the creation of new knowledge agents, incorporate them into a growing community of agents, and allow them to communicate and interact with fellow agents.

There is a wide variety of other prerequisites to interagent communication. One is that the agents use compatible communication protocols. The protocol aspect of communication encompasses establishing a communication channel, deciding on a content language, using the proper speech acts, e.g., assertion, query, etc., transmitting actual information, synchronization, error detection and recovery, etc. Tim Finin's Knowledge Query and Manipulation Language, KQML [Finin and others, 1992], is mainly concerned with that part of the communication between knowledge agents. Another important prerequisite of successful communication is that the exchanged content messages have the proper meaning so that they are understood correctly by the intended recipient. To achieve this, the language of one agent has to be translated into the language of the other. A form of translation — perhaps better called explanation — might be necessary even if the agents speak the same language, because they might have different expertise, use different terminology, etc.

One such technique, known as the *mediation approach* has been introduced by Gio Wiederhold [Wiederhold *et al.*, 1990]. This approach tries to assume as little as possible about the various knowledge agents, while enabling communication between them by providing a special class of agents called *Mediators* or *Facilitators*. These mediators normally speak the language of one particular knowledge agent as well as a common mediator language that lets them easily communicate with mediators of other knowledge agents. The advantage of the mediation approach is that it is applicable to a wide variety of already existing knowledge agents, The main disadvantage is that it involves possibly difficult translation at various levels.

Wiederhold justifies the use of mediators (called SoD's) this way: [Wiederhold *et al.*, 1990]:

Today, without the knowledge encoded in SoDs, the methods for retrieving the *best* information are explicitly specified by the user. It is likely to require distinct methods for multiple domains. Both in database and Prolog access styles, these specifications require knowledge of each the [sic] underlying domains and their structure. In today's database languages a sensible specification is likely impossible to state, so that all the data has to be retrieved into memory, and then processed and reduced by the application programs. (p. 67)

Our main interest lies in the mediation approach, and in this paper we investigate a particular kind of mediation. In our model, a mediator enables communication between agents by learning the meanings of new words, and forming appropriate models of the communicating agents' mental concepts

Such a mediator would facilitate the translation part of communication. Therefore, we are investigating the notion of an *ontological mediator* (OM), and the feasibility of implementing a computerized OM. An *ontological mediator* is an agent that enables communication among two or more intelligent agents who either speak different dialects of their common language or use different ontologies. Unlike KQML mediators who treat the messages of the agents as uninterpreted strings, OMs are to involve themselves in the meanings of the messages being sent among agents.

We need mediators *because* there is no common framework within which the community is developing knowledge agents. The interaction between specialized knowledge agents and users presupposes that the users already understand the meta-language required for knowledge queries, and will completely understand the responses they receive. When a human user doesn't understand a response, they will issue a new query in an attempt to gain clarification. Automated interaction between autonomous knowledge agents, however, was never intended. Mediators bridge the gaps between agents. They determine what clarifying questions to ask about concepts foreign to one of the agents. They rearrange the structure of queries and responses to smooth out inconsistencies and prevent miscommunications. Finally, they learn the correct translations for cases where use or ontology vary between agents.

2 What is an Ontology

The primary task of an OM is to mediate between agents using different ontologies, so, what are these ontologies? Or better, what do we mean when we say *ontology*? Let us start out with a few definitions from the literature and then give our definition of it.

From Webster's on-line dictionary:

> on.tol.o.gy noun [NL ontologia, fr. ont- + -logia -logy] 1: a branch of metaphysics relating to the nature and relations of being 2: a particular theory about the nature of being or the kinds of existence.

This definition characterizes ontology the way it is standardly used in philosophy.

In artificial intelligence (AI) circles people are less concerned with the actual nature of existence or reality than with the *modeling* of certain aspects of that reality. They use the term more along the lines of Tom Gruber, one of the principle designers of Ontolingua. He writes [Gruber, 1994, p.1]:

> An **ontology** is an explicit specification of a conceptualization. The term is borrowed from philosophy, where an Ontology is a systematic account of Existence. For AI systems, what "exists" is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is

called the universe of discourse. [...] Formally, an ontology is the statement of a logical theory.

Gruber's characterization of ontology qua logical theory actually subsumes more than what AI researchers usually consider as part of an ontology. In his view an ontology consists of a representational vocabulary with precise definitions of the meanings of the terms of this vocabulary *plus* a set of formal axioms that constrain the interpretation and well-formed use of these terms.

Most commonly however, the representational vocabulary is the only aspect considered to be part of an ontology. Such representational vocabularies are usually defined as *taxonomic class hierarchies*. For example, the taxonomy of a general natural language understanding system would have very general classes such as *abstract things* and *concrete things* at the top of the hierarchy, bottoming out at specific common nouns, verbs and adjectives. The ontology, together with the specific individuals of each taxonomic class, constitute the agent's knowledge base. The constraint axioms omitted from the specification of the ontology are added to the AI system separately.

In the following we will use the term ontology in this narrower sense. Hence, for us the primary function of an OM is to translate between differing taxonomies of communicating agents. One of the foci of our future research in this area will be to incorporate semantic constraints in inter-agent reasoning, as it would seem necessary if we wish to have agents understand each other's meanings.

3 Mediation interface

Inter-agent communication is based on a message passing protocol in which agents send messages to other agents, and receive messages from other agents. Each message has an explicit sender and an explicit receiver. The communication pathway can be any medium; we are thinking of a physical network connection employing standard data protocols such as TCP/IP. These protocols are separate from the protocols discussed here. We presume that both agents can employ the same language in the sense that they employ the same syntax, and use the same closed-class vocabulary. Open-class words may differ between communicating knowledge agents.

An ontological mediator sits like a two-way filter between two particular communicating knowledge agents. It serves as a translator for messages the agents send to each other. In our current model, every pair of knowledge agents will need its own translator. The OM monitors at least the syntactic content of sending agent's messages to ensure they will be received without misunderstanding. The translation problems mentioned in the previous section apply to the OM's ability to perform this task.

It is a requirement that agents who wish to communicate have ontologies that overlap. It is not necessary for the agents to share the same terms for every item of discourse, but at least they must share some intensional objects. In other words, if two agents' ontologies have nothing in common, they can have nothing to discuss with one another. Attempts by agent A to communicate with agent B when B doesn't share any of A's ontology need to be politely interrupted by the mediator.

Given that the communicating agents have some common ground in their ontologies, An ontological mediator can be used to extend the ontologies of either agent. If one agent begins to use a term unknown to the other agent, the mediator will introduce the term, and attempt to explain the term using language the other agent already understands. If the other agent is not satisfied by the explanation, or needs further information, it may, via the mediator, query the first agent. The ensuing meta-discussion ends when both agents are satisfied that the second agent understands the term well enough to use it appropriately in the context of communication. An appropriate task for the OM will be to determine when the meta-discussion has exhausted its usefulness for the agents, and it is time to continue the original communication.

A prerequisite to this process is that the mediator know something about the ontologies of both agents. In order to explain a term or translate terms between ontologies, the OM must be able to compare pieces of ontology from both agents until it finds sufficient similarity. It can extract pieces of an agent's ontology remotely by querying the agent, or it could store the complete ontologies of both agents in its own memory. The obvious advantage to maintaining local copies is reduced overhead, but it comes at the price of having to monitor dialog between agents and determine when ontologies are modified.

4 A comparison of ontologies

A number of different computer-based ontologies are used by existing knowledge agents to taxonomize the world and restrict the kinds of assertions that can be made about objects. We expect that communicating agents may use ontologies that differ. However, since any two rational agents who live in the real world have the common experience of real world, we expect to find that ontologies describing the same portion of reality will have significant overlap. We have collected a few ontologies from various sources in order to determine whether the extent to which ontologies overlap meets with our expectations.

4.1 Integrating ontologies

In building interlingua for communicating agents, one approach is to merge separate ontologies into one ontology that is consistent with both agent's world view. In order to determine whether this is feasible, we must examine the structure of ontologies. If they are identical except for concepts at the bottom of the taxonomy,

The Wordnet lexical database system provides relational information about synonym groups (*synsets*). The synsets for nouns are organized in a several hierarchies. The higest concepts for these hierarchies include {"act," "human action," "human activity"}, {"phenomenon"}, {"psychological feature"}, {"abstraction"}, and {"entity"}.

The Penman [Penman, 1989] taxonomy is designed to support natural language understanding. Part of this system is the Penman Upper Model, a LOOM [MacGregor, 1988] taxonomy that drives the natural language generation engine. Similarly, the CYC knowledge base [Lenat and Guha, 1989] is an ontology designed for general-purpose artificial intelligence systems.

We compare the upper levels of these three systems. Figure 1 illustrates the very top levels of the CYC and Penman system. In CYC, the top levels are in a relatively sparse tangled hierarchy. The Penman system, on the other hand, employs a dense tree. Notice also that there is very little obvious overlap in top-level concepts between CYC and Penman. Clearly, these are two orthogonal ways of taxonomizing the world.

It is not necessary for ontologies to match at every level, however. If two agents are discussing technical issues, the vocabulary will remain at the lower levels of their respective ontologies. It does not matter how the two agents partition their concepts as long as they mean the same things by use of particular concepts in dialogue.

Thus, a top-down approach to ontological augmentation, where agents discuss the classification of a concept beginning with the most general and abstract items in their taxonomies, will not always succeed because of a lack of a common frame of reference at that level. By the same token, a bottom-up approach, is not guaranteed either. Instead, if a common frame of reference can be found nested in both ontologies, an iterative graph traversal algorithm (growing both up and down) is the most successful heuristic. Our algorithm employs this approach by finding immediate subclasses and immediate superclasses of a noun concept, then expanding the search by part-whole relationship information.

Consider one of Wordnet's top levels, {"entity"}. It is divided into ten synsets, each of which has multiple words. The same word may belong to more than one sense, actually occupying multiple places in the ontology. Wordnet is much more dense and tangled hierarchy than both CYC and Penman. We omit a diagram of the {"entity"} top level for that reason.

Many approaches to the problem of heterogeneous knowledge sources involve merging two or more sources of knowledge into one large knowledge base. This is also known as building an interlingua. A major obstacle to building an interlingua is the task of translating each source into the knowledge representation language to be used in the interlingua. When ontologies are merged, it is often done by hand [Knight and Luk, 1994] It is not yet clear how to automate this process in the general case. Even if ontologies are structurally identical but use different words, there may not be direct translations between words at the same level of the ontology. The spreading search heuristic may lead to a solution for particular concepts. In addition to this basic paradigm we also require a model of agent-mediation-agent *explanation dialogue.*

Another problem related to merging is that of making use of common on-line resources, including machine-readable dictionaries. Given dictionaries such as the Longman's Dictionary of Contemporary English (LDOCE), or Roget's International Thesaurus (RIT), one may want to build up a lexicon that can be used by natural language understanding systems. Such a project has been attempted by [McHale and Crowter, 1994]. In order to build a complete lexicon, certain thematic roles for words have to be extracted from the dictionary. However, these roles are not explicitly represented in the LDOCE. A pattern matching heuristic is employed on sample sentences in the LDOCE to determine the ways words are used in the context of actual sentences. It can then be determined, for example, which verbs require direct or indirect objects. Additionally, their algorithm will attempt to correlate word senses between the LDOCE and RIT, essentially merging concepts from different sources and representing their common meanings. If a mediator understands and represents explicitly the *meanings* of words, or the concepts that the words label, rather than just the words themselves, it is better prepared to make proper translations.

Must translations be exact? Except in very precise and technical areas, the most intelligent of agents—humans—do not always mean the exactly the same thing, even when they use the same word. However, humans share enough of the meanings of words to allow them to communicate effectively, and understand *enough.* One theory that describes this phenomena is set forth in [Lehmann and Cohn, 1993]. Every concept is described by a two-part system, the EGG, which formally defines the concept, and the YOLK, which is a set of core exemplars, either formally defined, or enumerated. In order for effective communication to occur about a particular EGG/YOLK concept, both agents must agree on the YOLK, or agree to disagree about the meanings of the terms they are using. In addition, the EGG/YOLK formalism allows for a much finer distinction between concepts. There is a partial order of 46 distinct ways in which the EGGS and YOLKS of two concepts can overlap, each from total disjointedness to total equality. The agents can therefore work to integrate their knowledge automatically at the possible expense of accuracy, or they may choose not to allow integration except where the degree of concept overlap exceeds some threshold. This approach is useful to a mediator because one of the mediator's primary facilities is to determine when two different terms match well enough in context that they mean the same thing, or when to identical terms really mean different things to each knowledge agent.

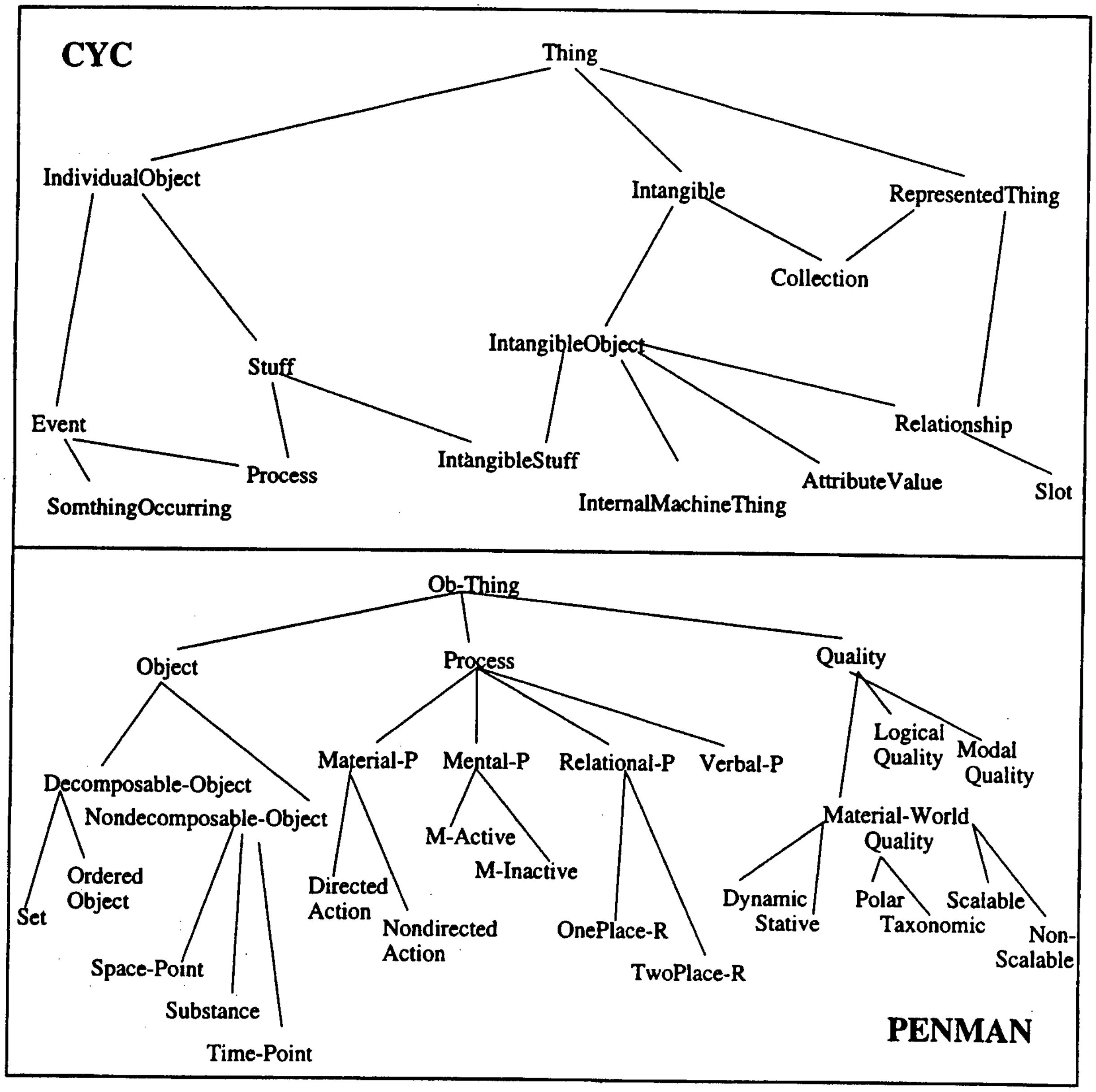


Figure 1: A comparison of ontologies

5 Terminological Representation and Translation

An ontological mediator needs to have knowledge of the base-level category words used by the agents whose communication it is mediating. We presume that the communicating agents speak the same language, and barring any evidence to the contrary, they use the same words to speak of objects known to both agents. Often, however, different agents will use different words to represent the same object, or the same word to represent different objects.

We have developed a representation scheme for ontologies and a mechanism for translating sentences when the agents use different words to represent the same object. Words used by a speaking agent are translated into the equivalent words used a listening agent. The agents themselves do not represent this meta-ontological translation knowledge. Rather, it is stored in the knowledge base of the Ontological Mediator. Ultimately, when agent A wants to sent a message to agent B, A will send the message to the OM for appropriate translation, then the OM will forward the message with translated terms to agent B.

Meta-ontological knowledge about the terms used by two agents is formed in the belief space of the ontological mediator by a set of asserted propositions of the form "Agent X believes that expression T denotes object O." Each agent is represented as an individual base node in the OM's semantic network. To specify a specific term translation, two propositions are asserted in OM's belief space: "Agent X believes that expression T_1 denotes object O_i," and "Agent Y believes that expression T_2 denotes object O_i." The O_i's in the OM represent the same concept for which the communicating agents use different terms. An O_i can be an individual or a class.

6 Definitions

The Ontological Mediator's primary function is to provide accurate translations between sentences of one agent and sentences of another. The agents use the same language, but might use open-class words unfamiliar to each other. The key to providing an accurate translation is to understand the content of the message. Most importantly, an agent needs to determine a definition for each unfamiliar open-class word it hears.

Karen Ehrlich [Ehrlich, 1994] is developing a system which will learn meanings of new words from the context in which they are used in a narrative. With respect to nouns, the system searches for the following narrative cues:

1. relationship to basic-level categories (identity, subclass, superclass)
2. function (what purpose the noun serves)
3. structure (what parts or possessions the noun has)
4. actions (what the noun does)
5. ownership (who or what can own a noun)
6. other properties (default size, color, etc...)
7. synonyms

Rather than read a narrative, an ontological mediator can ask questions of its agents. Given a new word, its task is to find a place for it in the agent's ontology. The mediator needs to ask the right questions in order to make this placement correctly.

Following the work of Ehrlich, we have begin to develop a system which interfaces with the Wordnet ontology and asks the user questions about a given noun. It asks the user to specify class inclusions and part-whole relationships about the word. It then displays possible words the given noun might match. If the user verifies a match, the system will place the new word in the corresponding synset, and update its internal database with this new information. Currently, our interface to Wordnet is read-only, but even if it were read-write, we don't believe it is the mediator's job to change the ontologies of its agents directly. If a word match cannot be found, but the user is satisfied that the given noun is a sibling of the hyponyms of a Wordnet word, the system will modify its internal database with a new synset of that sense containing just the new word. After this system is more fully developed, future queries of the system regarding the given noun will reveal its placement in the taxonomy.

The question/answer approach to determining the ontological status of a new word places a great deal of trust in the user's (or question-answering agent's) competence. For example, the agent must know the meaning of each word the mediator presents; otherwise important clues as to the new word's placement might go unnoticed. In our informal experiments to date, we have discovered that the user must almost know *a priori* the classification of sibling words in the destination synset. Otherwise, the user may cause the mediator to digress in a direction that ultimately fails to classify the new word. The user needs to be intelligent enough to realize when such a digression has occurred and to tell the mediator to back up to a point before the digression began.

7 Learning translations for isolated anomalies

Often it is the case that two agents really have the same ontology structure, but they use different words to refer to the same concept. A speaking agent may use an unfamiliar word to a listening agent. The mediator is then called in to learn an appropriate translation, and thus solve the communication discrepancy.

If the discrepancy happens in isolation, that is, where the words for surrounding concepts are the same, then A translation can be found by querying the speaking agent for the subclasses, superclasses, and coordinates of the misunderstood word.

Consider figure 2. Here are two agents which have some domain knowledge about machinery, but use different words. Perhaps the American agent uses the word "elevator" and the British agent uses the word "lift" to refer to the same concept. The British agent doesn't know the word "elevator." so the mediator needs to learn a translation.

The mediator first asks the American speaker for the subclasses and superclasses of "elevator" It finds that there are no subclasses, and the superclass is "lifting device." The next query is for the *coordinates* of "elevator." This query yields "winch," "crane," and "hoist." From these four concepts, the British listener can be asked if it has a concept which matches the superclass and coordinates retrieved from the speaker. In this case it does. We find that the concept corresponding to "lift" has superclass "lifting device" and coordinates "winch," "crane" and "hoist." From this match the mediator concludes with a high degree of certainty that the appropriate translation for "elevator" is "lift."

In many cases like this one, it will be able to respond with an appropriate term translation for the unknown word, although a simple examination of subclass, superclass, and coordinate matches is not sufficient to guarantee a high degree of accuracy in the translation. In these cases, other ontologic aspects of the words, such as part-whole relationships, roles in various actions, ownership, default properties such as color, relative size, etc.

All of this ontologic information can be used only if the agents are capable of providing it. Not

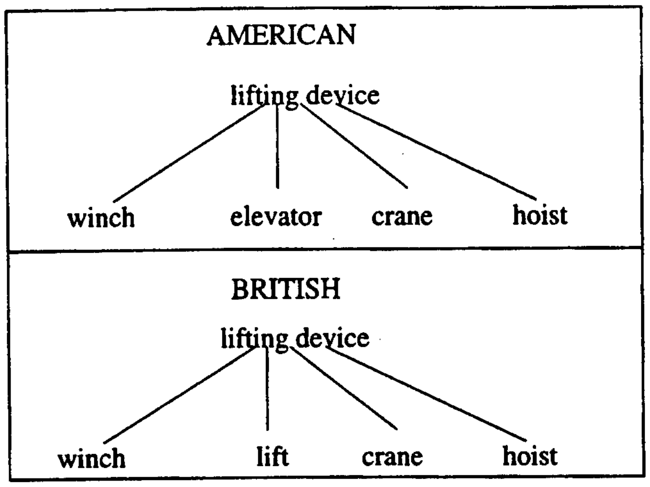


Figure 2: Isolated ontologic differences

all ontologies or agents are designed to be able to provide every piece of ontologic data a mediator could use to learn translation. Therefore, a mediator needs to be flexible enough to make use of the information it can get.

8 Conclusions and Future Work

A implementation of a computerized ontological mediator is being prepared. The mediator will store translations in its own knowledge base so that it does not have to re-learn them every time anomolies appear.

The simple isolated anomaly example illustrates the basic strategy to be employed in learning translations. It can be enhanced in a number of ways. First, there are more properties of ontologic entities than simple subclass, superclass, and coordinates. These properties, such as part/whole relationships, class of entity (object, relation, etc...) can be queried as well to further constrain the search for translations.

It is often the case that term-for-term translations don't exist. Here, the mediator might make a translation decision by determining which translation is the least anomalous, or it may choose to inform the agents that there is a terminolgy disagreement between them that can't be resolved by the mediator.

Domain knowledge is sometimes lacking, and one agent may not have as rich a knowledge base as another. In this case, the mediator may have to inform an agent that it has no concept corresponding to an unknown word. Perhaps in these cases the agent should be able to create a new concept. The mediator will be invaluable in helping the agent place the new concept into its ontology. This facilitates intellectual growth from agent dialogue.

An implementation of a computerized ontological mediator is being developed. The mediator will store translations in its own knowledge base so that it does not have to re-learn them every time anomolies reappear. This mediator program interfaces with a human user and an computerized agent such as Wordnet. A crucial requirement of both agents is that they be able to communicate directly about their ontologies.

The mediator program is given a word which is found in the ontology of the first agent (the human), but not in the ontology of the second agent (Wordnet). The OM then asks both agents to respond to queries about their ontologies. Since it is assumed that there is some ontological overlap, at least one common term related to the unknown word will be found in both agents' ontologies.

References

[Ehrlich, 1994] Karen Ehrlich. Automatic expansion of vocabulary through natural language contexts. Draft of Ph.D. dissertation, 1994.

[Finin and others, 1992] Tim Finin et al. An overview of KQML: A knowledge query and manipulation language. Unpublished Draft, 1992.

[Gruber, 1994] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. In Nicola Guarino and Roberto Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation.* Kluwer Academic, 1994. in preparation, also available as Stanford Knowledge Systems Laboratory Report KSL 93-04.

[Knight and Luk, 1994] Kevin Knight and Steve K. Luk. Building a large scale knowledge base for machine translation. In *Proceedings of AAAI*, Seattle, WA, August 1994.

[Lehmann and Cohn, 1993] Fritz Lehmann and Anthony G. Cohn. The EGG/YOLK reliability hierarchy: Semantic data integration using sorts with prototypes. Technical Report 93-2, GRandAI Software, 1993.

[Lenat and Guha, 1989] Douglas B. Lenat and R.V. Guha. *Building Large Knowledge-Based Systems.* Addison-Wesley, Reading, MA, 1989.

[MacGregor, 1988] R. MacGregor. A deduction pattern matcher. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 1988.

[McHale and Crowter, 1994] Michael L. McHale and John J. Crowter. Constructing a lexicon from a machine readable dictionary. Technical Report RL-TR-94-178, Rome Laboratory, Air Force Materiel Command, Griffiss Air Force Base, New York, November 1994.

[Neches *et al.*, 1991] Robert Neches, Richard Fikes, Thomas Gruber, Ramesh Patil, Ted Senator, and William R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3), Fall 1991.

[Penman, 1989] Penman. The penman documentation. Technical report, USC/Information Sciences Institute, 1989.

[Wiederhold *et al.*, 1990] Gio Wiederhold, Peter Rathmann, Thierry Barsalou, Byung Suk Lee, and Dallas Quass. Partitioning and composing knowledge. *Information Systems*, 15(1):61–72, 1990.

	Key-Key	Key-Foreign Key	Non Key-Non Key
Materialized Join View	7436	13275	45492
Materialized OuterJoin View	6457	12473	44491
Join ViewCache	2739	4689	4815
Partial Mater. Join ViewCache	671	1271	1714
Partial Mater. Projection ViewCaches	582	879	1093

Table 5: Comparison of I/O of View Incremental Maintenance Algorithms

8. CACHED KNOWLEDGE FRAGMENTS

Intelligent Agent Integration Technology

Prepared by:
Nick Roussopoulos
Co-Principal Investigator
University of Maryland - College Park
College Park, MD 20742
nick@cs.umd.edu, (301)405-6707

1 Summary of Research- Framework

In a distributed knowledge-based system, interactions among interoperating knowledge/database sources are carried out by transferring query results from one ore more sites to the requesting site. Therefore, a client-server architecture is appropriate. Many of these query results have a long usable life and can, therefore, be cached and maintained by a cache manager. These *Cached Knowledge Fragments (CKFs)* are used in subsequent broader or narrower queries and, because they are on the local cache, they are accessed at a fraction of the cost of a remote access. A client-server architecture has been enhanced to deal with the cached CKFs. This architecture provides the client full DBMS functionality for managing cached CKFs and other locally maintained (and perhaps privately owned) client data.

The basic goal of this research is to capitalize on CKFs which have been accessed from one or more source knowledge/data bases, hereafter the Servers, and delivered to a consumer, hereafter the Client. CKFs carry a lot of information which can be used during optimization of follow-up queries. During each interaction between a client and the servers, we observe the total cost of accessing them, the amount of data delivered, and cardinalities of the results. We then use *query feedback*, a novel approach introduced by our group, to adaptively improve query optimization and execution strategies.

We use cardinalities of CKFs along with the query predicates in figuring out attribute selectivities, attribute value distributions in the accessed sources, and total cost of prior queries in estimating the cost of follow-up queries. Attribute value distribution are approximated by a curve-fitting function. A similar technique is used for the interoperation cost model we introduce.

We have investigated the following areas:

1. update propagation strategies for CKFs
2. adaptive attribute selectivity and value distribution estimation
3. utilization of CKFs
4. adaptive query optimization in a heterogeneous environment.
5. adaptive query optimization in a heterogeneous environment.
6. efficient maintenance of remotely cached materialized views.
7. data dissemination for mobile clients

In the following subsections we describe the results obtained from this research and the resulted publications.

2 Update Propagation Strategies for CKFs

Client–Server models have emerged as the main paradigm in modern database computing [RD91], [DR92],[DR93]. The Enhanced Client–Server architecture takes advantage of disks and most of the DBMS functionality. Clients can cache server data fragments into their own disk units if such data is useful and accessed frequently. However, when updates occur at the server, some of the client CKFs may be affected by the updates. In such situation, the CKSs can either be invalidated or incrementally updated using the update logs of the servers. Such propagation of updates is crucial for the overall performance. In [DR94], we examine five update propagation strategies and techniques in the context of the Enhanced Client–Server DBMS architecture and examine their performance through detailed simulation experiments. These strategies are outlined below:

Lazy, On–Demand strategy: The server does not keep a catalog of client CKFs and does not propagate updates unless it is asked from a client. Each client makes a specific request for receiving relevant to a particular CKF updates just before it uses it. This strategy may increase response time, but has the least overhead and is scalable.

Eager Blind Broadcasting: This strategy broadcasts all update as soon as they occur to all clients. Although this has relatively small overhead for the server, it has a significant overhead on the network and the clients simply because all relevant and irrelevant updates have to be delivered to everyone who in turn has to examine them and take appropriate action.

Eager Informed Multicasting: This strategy sends eagerly relevant updates to a the affected CKFs of clients. This means that the server has to keep track of where the CKFs are located.This reduces network traffic and client processing time at the expense of servers' which now have to do catalog management for all CKFs.

Periodic Blind Broadcasting: Instead of being eager, blind updates are sent periodically or whenever a number of updates have passed a threshold.

Periodic Informed Multicasting: Again, informed updates are propagated at prespecified time intervals or thresholds.

Other strategies can be composed from these five.

3 Adaptive Selectivity Estimation

In most database systems, the task of query optimization is to choose an efficient execution plan. Best plan selection requires accurate estimates of the costs of alternative plans. One of the most important factors that affects plan cost is *attribute selectivity*, which is the number of tuples satisfying a given predicate. Attribute selectivities are directly related to *attribute value distributions*, However, real value distributions are not available and all query optimizers make various assumptions about them. Most of them assume uniform distribution that is already known to be very bad.

A study on error propagation [IC91] revealed that selectivity estimation errors can increase exponentially with the number of joins and, thus, affect the decisions in query optimization. Accurate selectivity estimation has become even more important in much larger database systems distributed over a network. In such systems, query plans have significance cost variance due to database size, volume of data transmission, and network latency. Therefore, accurate selectivity estimation is even much more crucial for distributed systems than centralized ones.

Typically, selectivity estimates are collected periodically by background processes which do sampling on different parts of the database. However, this solution is inadequate because of the heavy overhead and the lack of dynamically sampling the active part of the database.

In [CR94b], we present a new approach which accurately computes the attribute selectivities and value distributions using query feedback. Since queries are executed anyway, this approach has practically no overhead. The idea is to use the cardinalities from *query feedback* to "regress" the distribution gradually and, as queries proceed, the approximation becomes more and more accurate especially for the active part of the database. This adaptive "learning" from query executions not only "remembers" and "recalls" the selectivities of repeated query predicates, but also "infers" (predicts) the selectivities of other query predicates.

The advantages of this new approach are:

- Efficiency — Unlike the previous methods which do background database scans proportional to the database size, the overhead of our method is negligible. It only adds some computation cost to regress the query feedback and, thus, is independent of the database size and is very efficient.

- Adaptation — The technique is used during both queries and updates.

None of the previous methods can do this.

The technique of query feedback and the regression of value distributions will be applied to adaptively estimate the interoperation cost of heterogeneous systems.

4 Utilization of Cached Knowledge Fragments

As mentioned above, query results are cached on a client site to be reused at subsequent queries. The goal of this part of this study is to find an efficient algorithm for utilizing CKFs on a client's database with as little as possible access to the servers' databases.

Let Q be an SQL query of the form:

$$Q : \text{SELECT} * \text{FROM } R_1^Q, R_2^Q, \ldots, R_t^Q \text{ WHERE } Q^p(R_1^Q, R_2^Q, \ldots, R_t^Q)$$

where the *selection predicate* $Q^p(R_1^Q, R_2^Q, \ldots, R_t^Q)$ is a disjunction of conjunctions (or has been transformed to a disjunction of conjunctions) of atom predicates. The atom predicates that we consider are the usual $=, <, >, \geq, \leq, \neq$. We have taken the assumption that we do not make any *projection* on the attributes of the *cartesian product* $R_1^Q \times R_2^Q \times \ldots \times R_t^Q$.

Since CKFs are materialized views defined on the base relations of the srvers, we will refer to them by the term views. Clearly, only materialized views are useful in a remote client so that the data can be accessed locally without having to transmit the result over the network.

Let us name $V = V_1, \ldots, V_n$, the set of cached views that reside at the disk of the system. We suppose that the system keeps a description of each view, say V_i, in the form

$$V_i : \text{SELECT} * \text{FROM } R_1^{V_i}, R_2^{V_i}, \ldots, R_{t_i}^{V_i} \text{ WHERE } V_i^p(R_1^{V_i}, R_2^{V_i}, \ldots, R_{t_i}^{V_i})$$

where V_i^p is the selection predicate (again we made the assumption that no projection has be done).

We could alternatively say that the views $V_1, \ldots, V_n$ and the query Q are expressed in a *selection on cartesian product form* as :

$$\sigma_{V_i^p(R_1^{V_i}, R_2^{V_i}, \ldots, R_{t_i}^{V_i})}(R_1^{V_i} \times R_2^{V_i} \times \ldots \times R_{t_i}^{V_i})$$

A base relation R can be described as

$$\sigma_{true} R$$

Note that the form that we use for the description of the query Q as well as the views $V_1, V_2, \ldots, V_n$ is sufficient to describe any view that is the result of the *join*, *select* and *cartesian product operators.* It can also support the *union,difference* and *intersection* operators as long as they are applied on views of the *same type*. For a query that is defined as a selection on a cartesian product, we define as *type* the set of the relations that appear on the cartesian product. Ie,the type of V_i is the set $\{R_1^{V_i}, R_2^{V_i}, \ldots, R_{t_i}^{V_i}\}$. Our form of representing cached views is sufficient for the description of

$$\sigma_{F_1}(R_1 \times \ldots \times R_n) \cup \sigma_{F_2}(R_1 \times \ldots \times R_n)$$

since the above can be converted to

$$\sigma_{F_1 \vee F_2}(R_1 \times \ldots \times R_n)$$

Similar conversions apply for the intersections and differences of views of the same type:

$$\sigma_{F_1}(R_1 \times \ldots \times R_n) \cap \sigma_{F_2}(R_1 \times \ldots \times R_n) = \sigma_{F_1 \wedge F_2}(R_1 \times \ldots \times R_n)$$

$$\sigma_{F_1}(R_1 \times \ldots \times R_n) - \sigma_{F_2}(R_1 \times \ldots \times R_n) = \sigma_{F_1 \wedge \neg F_2}(R_1 \times \ldots \times R_n)$$

Our goal is to determine, *without accesing the content of CKFs* but solely working with the selection predicates of the views, a subset of $\{V_1, V_2, \ldots, V_n\}$, if any, which can be used in the *incremental computation* of Q. More specifically, we will try to find out a subset S of $\{V_1, V_2, \ldots, V_n\}$ such that the union of suitably selected *disjoint horizontal fragments* of the views of S will give the needed query Q , in a cost effective manner. Since we want to get Q as a union of selections on the cached views of S, we have to consider as possible candidates for the set S cached views of the same type with Q.

5 Adaptive Query Optimization in a Heterogeneous Environment

We are developing a query cost model in an environment of interoperating heterogeneous systems, such as DBMSs. In such an environment, where the optimizers are hidden inside these foreign systems, we want to use adaptive techniques for estimating the cost of mediators interfacing the application layer interface as opposed to the internals of the system. We are developing an adaptive cost estimation (ACE) module based on query feedbacks and statistics yielding more and more accurate cost estimates as the system learns from experience. The novelty of this approach is that the coefficients of the cost formula are determined and dynamically modified at run time and, thus, there is no need for sampling or calibrating databases.

In traditional distributed DBMSs, the formula for determining the total cost of distributed queries is:

Total-Cost = CPU-Cost + I/O-Cost + Message-Cost + Transmission-Cost

Such a formula can only be used if all participating DBMSs on different sites are compatible to each other such that the global query optimizer is capable of knowing the internal details of the participating DBMSs and obtaining necessary information from them.

We targeted our cost model for a client-server architecture in which the client accepts each query and ships it over to the server for execution. The server runs the query and sends back the result. The total query cost in terms of the elapsed time can be viewed as:

Total-Cost = Initialization-Cost + Query-Shipping-Cost + Query-Execution-Cost + Result-Transmission-Cost where

- Initialization-Cost is constant with respect to the query types.
- Query-Shipping-Cost is dependent on the number of messages passed between client and server for a given query.
- Query-Execution-Cost is based on the characteristics of the DBMS, the statistics of the operand relations and the type of a given query.
- Result-Transmission-Cost is dependent on the selectivity of the query and the size of the query result.

All of the above components will be adaptively estimated using ACE.

6 Adaptive Query Optimization in a Heterogeneous Environment

As in traditional distributed DBMSs, query optimization is important in HDBMS [Day85, SY+89, SL90, DKS92, LS92, ZL94, DSD95], particularly for *global queries* which are joins between tables from separate foreign DBMSs. A *global execution plan* for a global query constitutes of a sequence of subqueries which specifies the join order, table/result shipping direction, and execution sites. Although the optimization techniques used in traditional distributed DBMSs [ML86] can be adapted to HDBMSs [DKS92], they induce some non-trivial problems. One of the problems is *cost estimation* for query plans, which has been a recent research issue in HDBMS [DKS92, ZL94]. Cost estimation is essential in selecting the best plan among various global query plans. The problem is harder in HDBMSs than in traditional distributed DBMSs because foreign DBMSs from different vendors have different access methods, optimization strategies and cost models, all of which may be hidden from the global optimizer of the HDBMS.

This report presents a practical method for estimating the costs of global query plans for distributed HDBMSs based on experience acquired from previous query executions. The basic idea is to use *query feedback* to adapt a parametric cost function. The parameters of the cost function are gradually adjusted after each query execution, using *query and database dependent* feedback such as table size and predicate selectivities measured during the execution of the query, and *query execution time* measured after the query. Query and database feedback is independent of the performance characteristics of the underlying DBMS, network, and HDBMS client-server implementation, while query execution time is totally determined by them. An *Adaptive Cost Estimation (ACE)* module has been designed and implemented which adapts its parameters by distributing amongst them the estimation error which is the difference between estimated and actual values. The adapted parameters are then used for estimating follow-up queries. ACE is operational in $ADMS\pm$, an Enhanced Client-Server HDBMS prototype developed at the University of Maryland [RK86, RES93a, DR94], and obtains accurate cost estimates with small CPU overhead but no I/O. The ACE module works together with another adaptive module of $ADMS\pm$ which estimates the selectivities from exactly the sizes of the returned results [CR94b].

6.1 Cost Model

In traditional distributed database systems where all sites are running the same DBMS, the following formula is typically used in estimating the cost of a distributed query plan (with known local access methods) [ML86, OV91]:

$$Total\ cost \ = \ W_{CPU} * (number\_of\_instructions) + W_{I/O} * (number\_of\_I/Os) +$$

Cost Factor	Meaning
f_1	constant initialization overhead cost of 1 unit
f_2	number of messages required to execute an LCF
f_3	cardinality of the first operand table
f_4	average tuple length of the first operand table in bytes
f_5	cardinality of the second operand table
f_6	average tuple length of the second operand table in bytes
f_7	cardinality of query result
f_8	average tuple length of query result in bytes
f_9	total size of query result in bytes

Table 1: Notations for Cost Factors

$$W_{MSG} * (number\_of\_messages) + W_{BYTE} * (number\_of\_bytes) (1)$$

where $W_{CPU}, W_{I/O}, W_{MSG}$, and W_{BYTE}, are system-wide constants that denote the *weighted* (relative) cost per instruction execution, per I/O operation, per message transmitted and per byte of data transmitted over the network, respectively. Usually, these weights are empirical values obtained by running a large set of sample queries and are hard-coded into the DBMS kernel. Details about the local access methods of the query plan must be known a priori in order to estimate the parameters including number of instructions executed, number of I/Os performed, number of messages and total bytes of data transferred over the network. These parameters depend not only on the query characteristic and data profile (including table sizes and query selectivities), but also on the DBMS kernel's characteristics (including the size of the code that implements each access method, the buffer manager's strategies, and the network interface parameters). These parameters can only be available in proprietary solutions of homogeneous systems, and therefore are refereed to as *system-dependent* parameters.

The above formula, however, is of no use for the HDBMS case because the global query optimizer has no access to the system-dependent parameters and/or knowledge on how the optimizer of each foreign DBMS will perform. Since system-dependent parameters are unavailable, we must use a cost model that is solely based on *query/data-dependent* parameters such as query expression, data statistics, and estimated sizes of results. Like [ASC+79, K+85, ZL94], we assume that all four parameters of formula 1 are in proportion to a few basic quantitative query/data dependent parameters, called *cost factors*. ACE uses nine cost factors $f_1 \sim f_9$ whose meaning is shown in Table 1. For a *sp* query, the factors f_5 and f_6 are omitted and their values are zeros. The CPU and I/O costs on the server are captured in the $f_3 \sim f_7$ factors while the rest of them model the communication network and the client-server inter-operation cost. Exact or pretty accurate estimates of the above factor values can be obtained by the query feedback as these factors do not depend on the internals of the foreign DBMSs, and can readily be obtained.

For each query class $QC_j (1 \leq j \leq 6)$, ACE maintains and uses a cost estimation function:

$$\hat{c}(q) = \sum_i a_{i,j} \cdot f_i(q) \tag{2}$$

where $\hat{c}(q)$, the estimated cost of a LCF query $q \in QC_j$, is a linear combination of cost factors $f_i(q)$, with $a_{i,j}$ being the *cost coefficients* (of query class QC_j) that map the cost factors to the estimated cost. Note that unlike formula 1 where the CPU, I/O, and network costs are considered separately, the ACE cost formulae model the cumulative cost of all these costs regardless of the idiosyncrasies of the underlying DBMS and network.

Consider a *sp* query q of class QC_1 where the clustered index is maintained as a B-tree. The cost of shipping the query to the foreign DBMS will be subsumed by $a_{1,1}f_1(q)$ and $a_{2,1}f_2(q)$. The cost of navigating the B-tree, which includes the initialization overhead (a constant) and the number of B-tree nodes retrieved (depending on the height of the tree and the selectivity of the predicate), will be subsumed by $a_{1,1}f_1(q)$, $a_{3,1}f_3(q)$ and $a_{7,1}f_7(q)$. Similarly, the cost of retrieving and processing the qualified tuples from the relation will be subsumed by $a_{4,1}f_4(q)$, $a_{7,1}f_7(q)$, and $a_{8,1}f_8(q)$; the cost of transmitting the result over the network will be subsumed by $a_{9,1}f_9(q)$. Similarly, if a linear scan, rather than the B-tree, is chosen as the access method, the cost of the linear scan can still be properly subsumed by different products in the cost function. The purpose of the cost coefficients $a_{i,j}$ is to map a query to the cost of the access method that is most likely to be chosen, based on the characteristics of the query (which are quantified by the cost factors). The values of $a_{i,j}$ determine the accuracy of the cost estimation.

6.2 Implementation of ACE

We implemented ACE inside $ADMS\pm$, an enhanced multi-site client/server (E-CS) HDBMS, in which the clients are fully-fledged DBMSs capable of caching and maintaining downloaded data subsets obtained as a result of

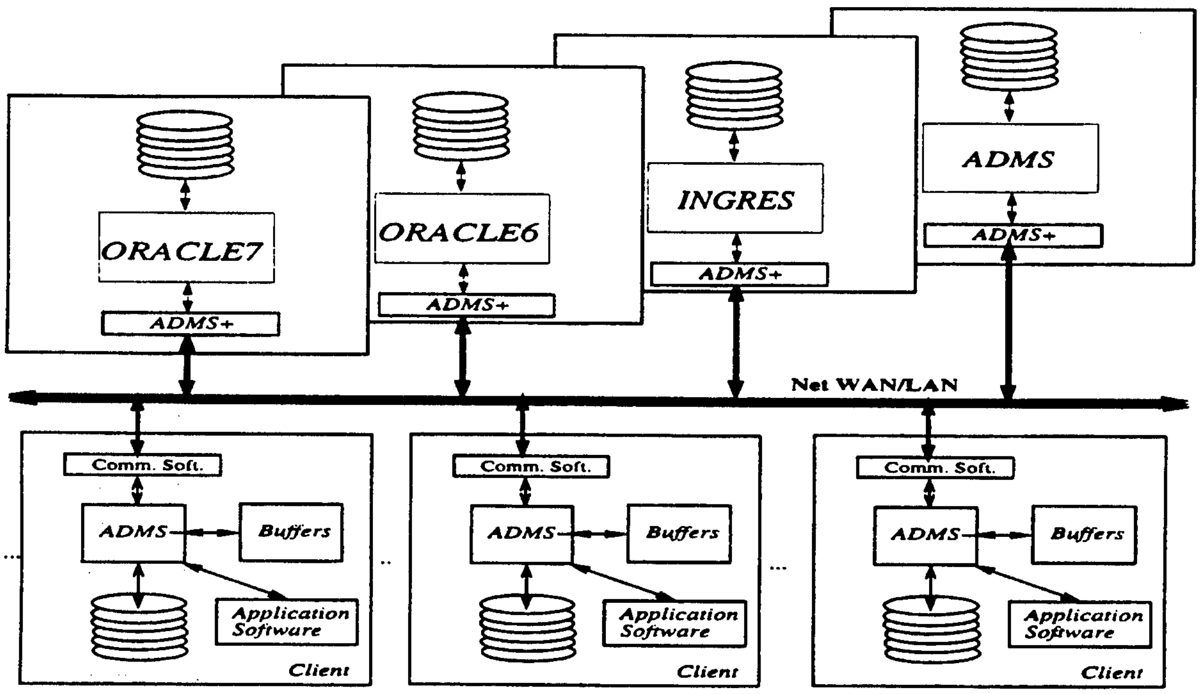


Figure 1: *ADMS*± System Architecture

running global queries on multiple DBMSs [RK86, RES93a, DR94]. The database servers are commercial and other prototype DBMSs accessed through application level gateway software, called *ADMS+*, which capitalizes on incremental access methods [Rou91b] for downloading and maintaining cached data results in the form of *materialized views.* The communication between servers and clients is based on TCP/IP Networking Protocol over LAN/WAN. Figure 1 shows the system architecture of *ADMS*± with three commercial DBMSs and our own ADMS prototype. Each client runs a single-user version of ADMS, called *ADMS−*, which maintains on its own local storage materialized views, cached catalogs, and statistics.

ACE is built into the *Global-Query Optimizer* of *ADMS*± to estimate the costs of different classes of LCF queries. The *Global-Query Optimizer* parses a global HDBMS query into a sequence of LCF-subqueries, obtains statistic information about the operand tables(cardinality, tuple length, indexes etc.) from the locally stored system catalogs, maps each LCF-subqueries into its corresponding query class based on the classification criteria defined in Table ??, then invokes the ACE module to produce the cost estimation for each LCF-subquery. The total cost of a global query plan can be obtained by summing up the costs of the composing LCF-queries. The Global-Query Optimizer then prunes off costly query plans and generates an execution plan with the minimum cost estimate.

One of the key factors related to the cost estimation is the *predicate selectivity*, which is the number of tuples satisfying a given predicate. The accuracy of selectivity estimation directly affects the accuracy of the query cost estimation. ADMS uses another adaptive module, called *Adaptive Selectivity Estimator (ASE)* [CR94a], for interpolating the value distributions of attributes which are then used to estimate selectivities. ASE produces accurate estimates of record selectivities from real attribute value distributions which are adaptively approximated by a curve-fitting polynomial using the query feedback mechanism. Its accuracy and performance have been reported in the above paper.

Both ACE and ASE are modules of the global-query optimizer of $ADMS\pm$ using query feedback to adapt. ASE is invoked by ACE when a LCF-subquery with a selection predicate is generated by the global-query optimizer and its selectivity needs to be estimated. In $ADMS\pm$, the query feedback consists of (a) the actual selectivity obtained after running the query, (b) the actual real time cost of the query execution, (c) catalog statistics from the server(s). ASE uses (a) and (c) while ACE uses (b) and (c). Catalog statistics basically include table cardinalities and indexing information. These are piggy-backed with the query result from the server(s) and used to update the locally cached client catalogs.

ACE and ASE require some matrix manipulation and mathematical computation but only incur CPU cost. In our $ADMS\pm$ implementation of ACE and ASE, the overhead of the ACE and ASE module computation is only a small fraction of the optimization cost and negligible when compared to the real query execution cost.

As mentioned above, ACE uses real wall-clock time observed on the client to adapt. For each query, a start timestamp is obtained by the client just before it begins to transmit the query to the server(s) and an end timestamp is recorded after the last record of the result has been received. The elapsed time between the timestamps is our metric of cost and measures all other costs, inter-operation, server CPU, server I/O, communication, and server and network contention factors.

6.3 Experimental Results

We performed extensive experiments to estimate LCF query cost in $ADMS\pm$. The configuration of these experiments included three commercial DBMS servers Oracle (v7.0), Oracle (v6.0) and Ingres (v6.0), our own prototype ADMS (v3.3) server, and the $ADMS\pm$ (v2.0) Enhanced Client-Server HDBMS. Oracle 7 runs on a SparcStation 20, Oracle 6 on a SparcStation 2, Ingres on a DECstation 5000/200, and ADMS on a SparcStation 2. The clients were run on separate SparcStation 2s. All client and server machines are connected via a shared Ethernet network. All the experiments were conducted during the night under low network/server loads.

We used the Wisconsin Benchmark relations [BT94] in all our experiments. The eight tables used along with their statistics are shown in Table 2. These are pre-loaded into each of the server DBMSs before the experiments are run. A range-varying parametric query for each of the six LCF classes of queries was used to generate randomly distributed range queries. These are shown in Table 3 in their $ADMS\pm$ extended SQL syntax with C_1 and C_2 being the variable range parameters and, $REL1$ and $REL2$ being relation variables from the database. The result is to be downloaded to and stored in a materialized view on the client.

For each LCF class, one hundred different ranges were randomly generated but with controlled selectivity to generate results of varying sizes from 10 to 10000 records. Each of these groups of one hundred queries was regenerated for four different sets of varying size relations to obtain 400 queries for each class or 2400 queries for the whole experiment. These 2400 queries were randomly mixed to generate the final query stream used in all server runs of our experiment. The random ranges, the variations of relations and the random mix were employed to reduce side-effects of shared buffers by similar queries.

The query stream was run from *cold start* and a single log for each server was generated. From these logs we make our observations and draw conclusions. We compare the ACE estimated cost with the actual real-time costs and generate histograms and graphs showing the accuracy of the estimates, the relative errors, and the adaptive capability of ACE.

Figure 2 ~ Figure 5 show the statistical and confidence analysis of ACE's runs on all four servers. The histograms show the percentage of queries for each 10% intervals of relative error. On Oracle7, 92% of the queries had relative error between 0 and 10%. The corresponding figure for Oracle6 is

90%, for Ingres 78% and for ADMS[1] 70%. The percentage of queries for which ACE had relative error of less than 20% range from an impressive 97% for Oracle7 down to 92% for Ingres.

The right hand side of Figure 2 ~ Figure 5 illustrate the confidence analysis on the mean relative error for each 10-second query time interval ranging from 0 to the maximum query time on each server. The *confidence coefficient* was set to 95%. These graphs show the mean relative error and its standard deviation contained below the 20% value.

[1]ADMS is more susceptible to Unix mannerisms and its eager prefetching which are more difficult to estimate. Another reason for the lower figure for ADMS is that query execution times are much shorter than all the other server DBMS and thus the standard deviation of the relative error is much more sensitive.

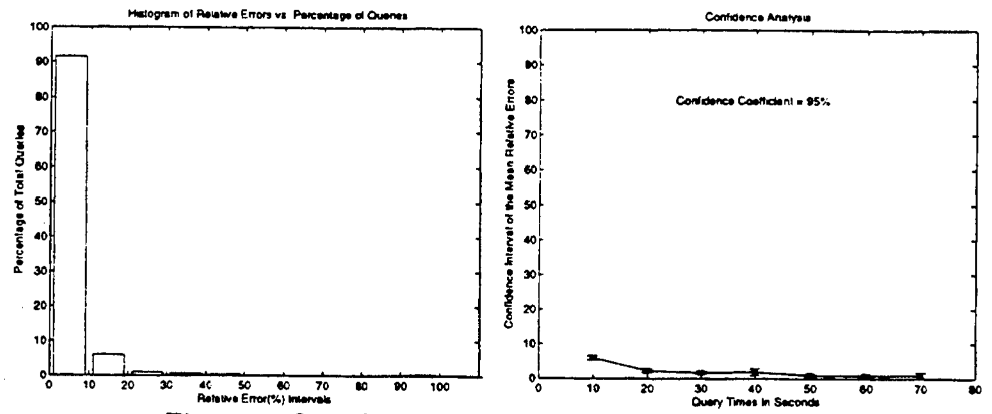


Figure 2: Complete Mixed Queries on Oracle7 Server

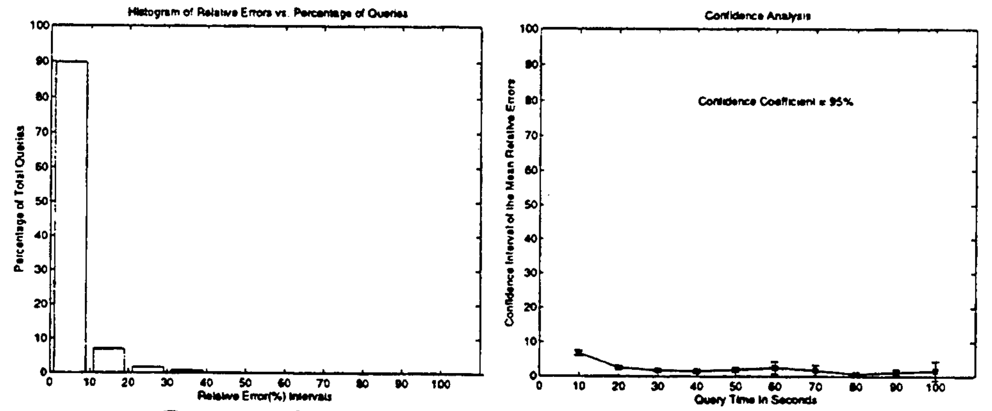


Figure 3: Complete Mixed Queries on Oracle6 Server

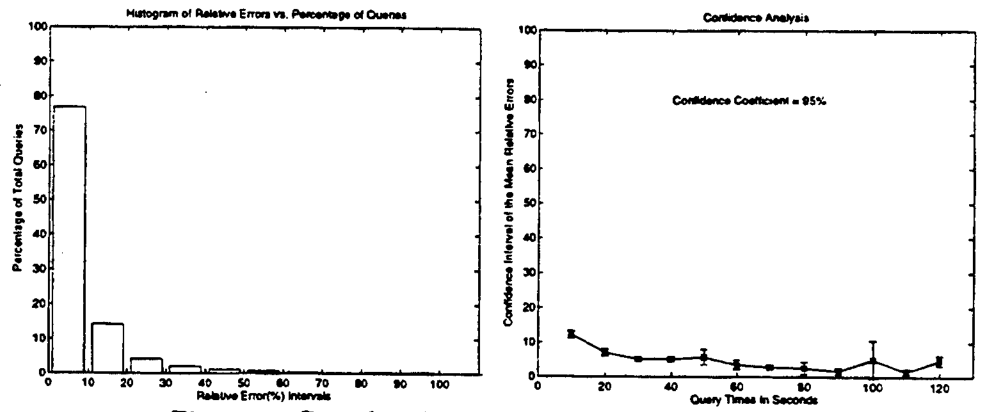
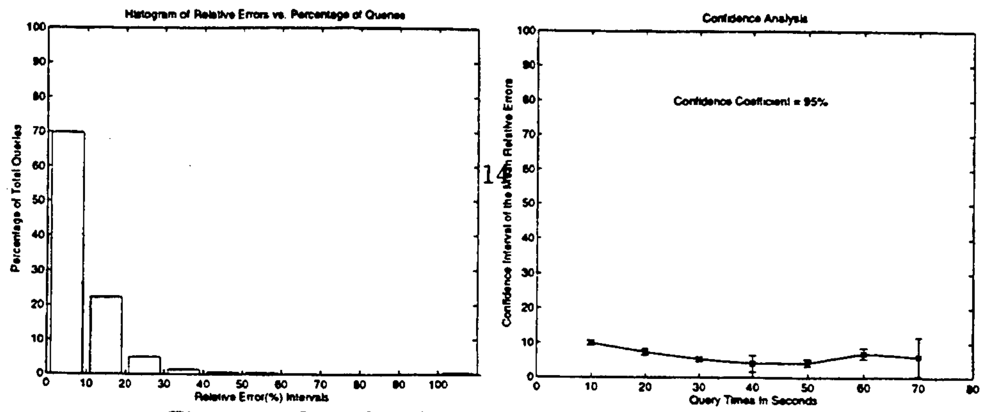


Figure 4: Complete Mixed Queries on Ingres Server



[illegible]

7 Efficient Maintenance of Cached Materialized Views

View refreshment is a central issue in distributed database environments where efficient access to a set of source databases must be supported. This is the case in data warehousing environments where views defined on physically separated, heterogeneous, and autonomous databases must be supported [ZGMHW95]. Changes in the source databases must be transformed into a data model format used by the warehouse and integrated into the warehouse.

A data warehouse can be used as an integrated and uniform basis for decision support, data mining, data analysis, and ad-hoc querying across the source databases. A data warehouse is likely to be a growing collection of possible time-stamped raw data. In some cases a data warehouse will be an append-only database. For example, a department store chain may want to support central access to all sales transactions. In some cases a data warehouse will be a temporally constrained database. For example, the department store chain may want to limit the access to sales transactions within the last year.

A data warehouse must not necessarily be completely up to date. The reason is that the warehouse is used for analytical, managerial data processing rather than for day-by-day operational data processing. Inmon has even recommended that a data warehouse is refreshed with a 24-hour delay in order to ensure that warehouse data are not confused with operational data [Inm93]. Consequently, it makes sense to assume that warehouse use and warehouse refreshment are separate processes. We believe that such separation will make it easier to answer warehouse queries efficiently and to solve the problems of warehouse consistency [ZGMHW95].

In this report we address some of the problems of efficient incremental refreshment of warehouse views. We focus on the following questions. *What information should be stored at the warehouse in order to support efficient view refreshment?* It is much more expensive to retrieve a remote disk page than to retrieve a local disk page. Therefore, network communication should occur only when a source database signals relevant changes to a warehouse. The warehouse should be able to process the changes without querying source databases. Otherwise, warehouse refreshment may become too dependent on the activity, availability, and efficiency of the source databases. This implies that a copy of all data objects that are needed for view refreshment must be present at the warehouse.

What data structures and algorithms should be used for view organization and refreshment? Incremental view refreshment should be performed with as little access to raw data as possible. The raw data objects may be very large compared to the attributes that are used in selection and join predicates. Indexing and view caching can be used to reduce or avoid raw data access during view refreshment. A combination of three basic methods can be used to support data warehouse views. A computed view is stored as a view definition, ie., as a query expression. A computed view is reconstructed each time it is accessed by methods like query modification [Sto75]. A pointer cached view is stored as a set of pointers that identify the data objects in the view. The pointers can be used to create a materialized view [Rou91a]. A materialized view is stored as the set of data objects that belong to the view [Han87]. Pointer cached views and materialized views must be maintained incrementally.

We have developed five refreshment algorithms that are based on various combinations of materialized views, partially materialized views, and pointer caches. Existing data warehousing approaches focus solely on materialized views. We present the results of an experiment that strongly indicates that refreshment algorithms based on a combination of materialized views, partially materialized views, and pointer caches outperform algorithms based solely on materialized views. We have assumed that all involved databases are relational databases.

7.1 Experiments

In this section we describe a limited experiment testing some of the incremental join algorithms developed. All the experiments were run in the ADMS prototype [RES93b]. ADMS engine has been developed to take advantage of cached views.

We ran the experiments to measure the I/O cost of the five algorithms described in the above subsection: the two basic categories of fully Materialized Views, the ViewCache Pointer based one, and two Partially Materialized and partially ViewCaches.

1. Materialized Join View: the Warehouse stores the tuples of the result and, in separate relations, the tuples which do not appear in the Join View. These are necessary for discovering tuple joins that were not joining before, but they may join with newly inserted tuples on the other relation.

2. Materialized OuterJoin View: the Warehouse stores the outer join. This view does not need any additional information as the not joining tuples are stored in the OuterJoin view.

3. Pointer ViewCache: Only ViewCache pointers to the underlying relations which, like all pointer based views, are also stored in the warehouse.

4. Partially Materialized ViewCache: the warehouse stores the ViewCache augmented with all the joining values of the underlying tuples next to the pointers.

5. Partially Materialized projections of the ROWID and the joining attribute values, and a ViewCache pointer view.

7.1.1 Experiment design

We used a variation of the Wisconsin Benchmark relations [BT94]. Two of them with 10,000 records each and another with 24,000. In each of the 10,000 relations, 20% of the tuples do not join with any of the 24,000 tuples in the third relation. The join views were created and stored on the warehouse. They included:

1. *key-key join* where the joining attribute was a key on both relations, which produce a tuple count of 10,000.

2. *key-foreign key join* where the joining attribute was a key on the first and foreign key on the second, which results in a tuple count of 24,000s.

3. *non key-non key join* where neither of the joining attributes were keys in their respective relations, and produces 48,000 tuples.

After the creation, 10% insertions were applied to each of the base relations and the incremental algorithms were performed. The I/O needed to perform the algorithms were obtained from the ADMS buffer manager statistics of page faults.

We did not test the algorithms under deletions. The reason is that, materialized view based methods are very different than pointer based ones. The first category requires sophisticated preprocessing of the logs and non-negligible I/O – comparable to duplicate elimination – pointer based methods incur no cost for deletions when done in the same pass with the processing of insertions [Rou91a].

7.1.2 Storage Overhead

First, we provide storage statistics for each of the view categories. Table 4 indicates the amount of storage in excess of the total storage cost required for the base relations. Note that in these sizes we assume that for the Materialized Join and OuterJoin Views, the warehouse does not store a copy of the base relations but, for the pointer based categories, it does.

The table shows that materialized views have an overhead ranging from 20-324% and this is caused by the multiplicity factor of the joining tuples, with the worst case being the non-key to non-key join. On the other hand, the pointer based methods incur overhead that ranges between 2-12%. This significant difference in storage overhead is the main reason for similar difference in I/O performance discussed in the next subsection.

7.1.3 Performance of Algorithms

In each of the described algorithms, we applied the best, to our knowledge, method. Most of the incremental algorithms are based on two-way hash joins but for the Materialized view category, we used a 3-way hash based join algorithm which hashes the insertion logs of the two relations, and then scanning the materialized join view or outer join view once. Then duplicate elimination of the resulting tuples was done afterwards. For the pointer-based algorithms, duplicate elimination is achieved by bookkeeping on the fly using hash bit vectors on the ROWIDs of the tuples.

Table 5 shows the dramatic performance difference of the methods. Clearly, the materialized view based methods incur high I/O volume due to their mere size. It is doubtful that algorithm improvements can reduce the I/O to a point to compete with the pointer based techniques. From the pointer based ones, the straight pointer ViewCache spends more than 94% of its I/O is from the underlying base relations for obtaining the joining tuple attribute values. This lead us to the last two pointer based algorithms which keep these values in a easier and a lot less I/O intensive disk cache.

8 Data Dissemination for Mobile Clients

We extended our architecture and algorithms to mobile computers and wireless networks. In this environment, mobile computing can only be fully utilized if the data associated with the mobile applications becomes equally mobile. Our initial results were published in [SRB96] and a more thorough tratment of the work will appear in [SRB97].

With smaller and yet more powerful computers becoming more and more common, with disks shrinking in size but increasing in capacity, and with networks providing more and more wired and/or wireless connectivity, the *ADMS*± project is exploring how to efficiently achieve data dissemination and caching on ever moving mobile clients based on their need . The need of each client may be changing rapidly, and in many cases, depending on the client's location, the network's bandwidth and reliability, and security.

The *ADMS*± project provides an SQL interoperability over TCP/IP. It allows wired connection to multiple commercial databases, including Oracle, Ingres, Sybase, and Illustra, over LAN/WAN. Query results are dynamically downloaded and cached to the client's, and maintained incrementally thereafter. As the client accesses remote database systems,*ADMS*± builds a working data set pertinent to the client's application data needs. Updates applied on the servers are This architecture has two significant advantages. First, the latency of data access gets significantly reduced by accessing locally cached data, and, second, network data transmission is reduced to absolutely minimum by only transmitting incremental updates. Experiments and simulations have shown that, depending on the update ratios, the throughput rate of the *ADMS*± architecture is one to two orders of magnitude higher than a standard client-server architectures.

We applied the *ADMS*± architecture to broadcast data through *Data AirWaves.* We are addressed the issues of asynchronous delivery of data and updates to clients connected through wireless links. More specifically, we proposed an adaptive scalable architecture based on wireless data broadcasting. We use a broadcast channel to transmit the update logs of servers as well as query results requested by the clients.

We compiled a list of applications that have a need for wireless data dissemination such as Battlefield Management, Tele-medicine, Doctor Hospital Rounds, Road Emergency, and Road Service. The computing environment in these applications is highly dynamic in terms of network topology, availability, load factor, and data location. Current distributed database technology is geared towards static environments in which communication is reliable and bandwidth is readily available. Therefore, classic distributed query processing and data management cannot cope with the dynamics of ubiquitous client mobility in the above environment. There is a need for intelligent staging and data dissemination to the mobile units while they are being deployed or moving to their field position. Data migration must be done in an asynchronous mode that is not intrusive to the mobile unit. This can be done during idle time or as a light background process.

To achieve this, we developed robust three level database architecture which extends the client-server paradigm. Level 0 includes the legacy database servers (LBS) which, in many cases, exist and are placed in secure and/or secluded locations. Level 1 contains database subserver units (SU) capable of storing significant subsets of data pertinent to an area or a mission and can be transported near the field of operation. Finally, level 2 contains a very large number of mobile clients (MC). Each client can capture data and carry it on-board for subsequent use. Depending on their functionality and area of deployment, clients may need to have relatively large storage capacity to reduce the number of remote data access. For data that is not stored on a client's store, the client will make one or more requests to the nearest SUs. If the data is available in the SUs, it is transferred to the client. Otherwise, the SUs turn themselves into clients and place requests to the servers. The servers then broadcast the requested data and all SUs may tune in to receive it. The SUs act as mediators which reduce the amount of interactions between the clients and the legacy database servers. Each of the SUs can be preloaded with the subsets of the databases that are pertinent to the location and field conditions in which it is to be deployed and it will continue to adjust its data subsets in order to satisfy their clients requests.

A client's query will be first attempted against its locally stored data. If the data is not available, the nearest SUs will be queried which may, themselves have to pose the same or subqueries to the LBSs. In the latter case, the LBSs will broadcast the results to all the SUs which may tune-in to "listen" to the results for "relevant" to their mission data. The broadcasting of the server and the filtering of the results on the SU sides are instrumental in the scalability of the centributed architecture.

Updates are posted to the servers who are then responsible for broadcasting the update logs. The logs are then filtered and, if relevant, applied to the locally stored data in each of the SUs. The incremental update also reduces network load by avoiding of retransmitting complete and much larger data objects.

We implemented this wireless architecture using the Altair system and the Hughes DirecPC. One server, one SU serving as a proxy, and one client were connected. The server was pushing data through broadcast channel over the DirecPC satellite. The SU was receiving and locally storing the received data. The client gets its data from the SU. If the data is not there, a direct request is sent to the server which would then gets broadcasted over the satellite.

Although the wireless data dissemination is well beyond the scope of the current contract, it was done as a demonstration of the versatility the caching techniques developed in this project. The enhanced client-server architecture and the capability of caching query results as query fragments is general and applies to a wide variety of applications. During the life of this contract, it became apparent that data dissemination and management of disseminated data is crucial to most data intensive applications.

References

[ASC+79] M.M. Astrahan, P.G. Sellinger, D.D. Chamberlain, R.A. Lorie, and T.G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 22–34, 1979.

[BT94] D. Bitton and C. Turbyfill. A Retrospective on the Wisconsin Benchmark. In M. Stonebraker, editor, *Readings in Database Systems*, pages 422–441. Morgan kaufmann, 1994.

[CR94a] C. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In *Proc. of ACM SIGMOD*, 1994.

[CR94b] C.M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[Day85] U. Dayal. Query Processing in Multidatabase System. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*. Springer Verlag, 1985.

[DKS92] W. Du, R. Krishnamurthy, and M. Shan. Query Optimization in Heterogeneous DBMS. In *Proc. of the 18th International Conference on VLDB*, Vancouver, Canada, 1992.

[DR92] A. Delis and N. Roussopoulos. Performance and Scalability of Client–Server Database Architectures. In *Proc. of the 19th Int. Conference on Very Large Databases*, Vancouver, BC, Canada, August 1992.

[DR93] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. *IEEE–Transactions on Software Engineering*, 19(2):120–138, February 1993.

[DR94] A. Delis and N. Roussopoulos. Management of Updates in the Enhanced Client–Server DBMS. In *Proccedings of the 14th IEEE Int. Conference on Distributed Computing Systems*, Poznan, Poland, June 1994.

[DSD95] W. Du, M.-C. Shan, and U. Dayal. Reducing Multidatabase Query Response Time by Tree Balancing. In *Procs. of the 1995 ACM-SIGMOD Int'l Conf. on Management of Data*, 1995.

[Han87] E.N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, San Fransisco, California*, pages 440–453, 1987.

[IC91] Y.E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Procs. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–277, Denver, Colorado, 1991.

[Inm93] W.H. Inmon. *Building the Data Warehouse.* John Wiley & Sons, Inc., 1993.

[K+85] W. Kim et al., editors. *Distributed Database Query Processing.* Springer-Verlag, 1985.

[LS92] H. Lu and M. Shan. Global Query Optimization in Multidatabase Systems. *1992 NFS Workshop on Heterogeneous Databases and Semantic Interoperability*, 1992.

[ML86] L.F. Mackert and G.M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Procs. of the 12th Intl. Conf. on Very Large Data Bases*, 1986.

[OV91] M. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RD91] N. Roussopoulos and A. Delis. Modern Client–Server DBMS Architectures. *ACM-SIGMOD Record*, 20(3):52–61, September 1991.

[RES93a] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. *IEEE Trans. on Knowledge and Data Engineering*, 5(5):762–774, 1993.

[RES93b] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. Technical Report UMIACS-TR-90-103, University of Maryland Institute for Advanced Computer Studies, 1993.

[RK86] N. Roussopoulos and H. Kang. Principles and Techniques in the Design of *ADMS±. Computer*, December 1986.

[Rou91a] N. Roussopoulos. An Incremental Access Method for Viewcache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[Rou91b] N. Roussopoulos. The Incremental Access Method of View Cache: Concept, Algorithms, and Cost Analysis. *ACM Transactions on Database Systems*, 16(3):535–563, September 1991.

[SL90] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.

[SRB96] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive data broadcasting using air–cache. In *First International Workshop on Satellite-based Information Services (WOSBIS)*, Ray NY, November 1996.

[SRB97] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras. Adaptive data broadcast in hybrid networks. In *1997 Proceedings of the 23rd International Conference on Very Large Databases, Athens, Greece*, August 1997.

[Sto75] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. In *Proceedings of the 1975 SIGMOD Workshop on Management of Data, San Jose, California*, pages 65–78, 1975.

[SY+89] P. Scheuermann, C. Yu, et al. Report on the Workshop on Heterogeneous Database Systems held at Northwestern University, Evanston, Illinois, Dec. 1989.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA*, pages 316–327, 1995.

[ZL94] Q. Zhu and P. Larson. A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System. In *Proc. of the 10th International Conference on Data Engineering*, 1994.

Relation	Tuple_Length	Cardinality	Clustered_Index	Non_Clustered_Index
onek1	182	1000	Y	Y
onek2	182	1000	N	N
twok1	182	2000	Y	Y
twok2	182	2000	N	N
fivek1	182	5000	Y	Y
fivek2	182	5000	N	N
tenk1	182	10000	Y	Y
tenk2	182	10000	N	N

Table 2: Experiment Relations

Query Type	$ADMS\pm$ Extended SQL Format
sp with a cluster-indexed attribute	select $a_1, \ldots, a_n$ from $DB.REL1$ where $un2 > C_1$ and $un2 < C_2$ into $VIEW1$;
sp with a non-cluster-indexed attribute	select $a_1, \ldots, a_n$ from $DB.REL1$ where $un1 > C_1$ and $un1 < C_2$ into $VIEW2$;
sp with no indexed attribute	select $a_1, \ldots, a_n$ from $DB.REL1$ where $k1 > C_1$ and $k1 < C_2$ into $VIEW3$;
spj with a cluster-indexed attribute	select $a_1, \ldots, a_m, b_1, \ldots, b_n$ from $DB.REL1$, $DB.REL2$ where $DB.REL1.un2 = DB.REL2.un2$ and $DB.REL1.un2 > C_1$ and $DB.REL1.un2 < C_2$ into $VIEW4$;
spj with a non-cluster-indexed attribute	select $a_1, \ldots, a_m, b_1, \ldots, b_n$ from $DB.REL1$, $DB.REL2$ where $DB.REL1.un1 = DB.REL2.un2$ and $DB.REL1.un1 > C_1$ and $DB.REL1.un1 < C_2$ into $VIEW5$;
spj with no indexed attribute	select $a_1, \ldots, a_m, b_1, \ldots, b_n$ from $DB.REL1$, $DB.REL2$ where $DB.REL1.k1 = DB.REL2.un2$ and $DB.REL1.k1 > C_1$ and $DB.REL1.k1 < C_2$ into $VIEW6$;

Table 3: Experiment Queries

	Key-Key	Key-Foreign Key	Non Key-Non Key
Materialized Join View	1.22	1.64	3.24
Materialized OuterJoin View	1.20	1.58	3.18
Join ViewCache	1.02	1.03	1.06
Partial Mater. Join ViewCache	1.04	1.06	1.11
Partial Mater. Projection ViewCaches	1.07	1.04	1.12

Table 4: Storage Overhead

9. Planning Initiative Information Agent

Intelligent Agent Integration Technology

Prepared by:
Donald P. McKay
Principal Investigator
Lockheed Martin
590 Lancaster Avenue, PO Box 4001,
Frazer, PA 19355-1808
Donald.P.McKay@lmco.com, (610)407-3527

This section is a copy of material developed under the Lockheed Martin Planning Initiative contract. It is provided here as additional related information about the use of KQML in an information agent architecture .

An Architecture for Information Agents

Donald P McKay, Jon Pastor and Robin McEntire
Loral Defense Systems

Tim Finin
Computer Science and Electrical Engineering
University of Maryland - Baltimore County

Abstract

Information agents include a significant class of applications which mediate information structures of domain objects to instance representations in a storage manager. Over the past several years, we have been experimenting with an information agent architecture in the context of the ARPI. Our information agent architecture uses the Knowledge Query and Manipulation Language (KQML) to implement access the knowledge services of such an information agent. The information agent itself, which we call the Loom Interface Module (LIM), uses knowledge structures to represent domain objects and contains an explicit mapping of knowledge structures to representations in an external storage manager, a relational database management system. We have developed several performance metrics and features for information agents constructed using this architecture. We described several key component algorithms and performance measurements We have developed the performance metrics, analysis and examples as a part of ARPI TIEs, introduction into the Common Prototyping Environment and, most importantly, under collaboration with the SIMS project at USC ISI and with the CoBASE project at UCLA.

Introduction

Knowledge-based systems can provide a key information processing aid to operational planning, scheduling and monitoring of operations. Specifically, these systems can provide key information support for current deficiencies in crisis action planning for transportation logistics. Requirements for these systems include the ability to access, manipulate, and modify the information stored in existing databases, and, a high level of collaborative and cooperative processing with the other planning agents including people and software components. Within the ARPA/Rome Lab Planning Initiative (ARPI), Loral Defense Systems, in collaboration with USC ISI and UCLA, developed an intelligent information services architecture which integrates cooperative user interaction and information location via domain/user-oriented object representations. This effort, involving participants and software components developed by Loral Defense Systems, USC ISI and UCLA, demonstrated an experimental prototype operating in real-time over the internet capable of providing information satisfying user requests making transparent to the user 1) query relaxation and reformulation despite over-specific queries and lack of data, 2) location and selection of information sources based upon multiple selection criteria, 3) transformation of low-level data source information from databases into domain and user relevant information structures, and 4) the query language utilized. Internal communications over the internet were implemented using KQML, the Knowledge Query and Manipulation Language, an ARPA-sponsored emerging language and protocol for information exchange.

In this paper, we describe technology components to support persistent storage and retrieval of plans and other military transportation relevant entities. This includes the integration of knowledge-based (KB) representation and reasoning systems with standard database (DB) management systems and the development of new standards for interface languages between knowledge-based systems and other software components including knowledge-based systems themselves. The integration of knowledge bases and databases is accomplished by the Loom Interface Module (LIM). LIM allows Loom (MacGregor & Bates 1987) applications to reason efficiently over a large collection of data from a database by utilizing the efficient computational capabilities of a database management system and by avoiding the need to create regular Loom objects to represent intermediate data. In order to enhance the integration of multiple knowledge-based systems, Loral Defense Systems and UMBC are designing and prototyping a new high-level protocol for conveying knowledge between systems. This protocol,

[1] This work was supported by Rome Laboratory and the Advanced Research Projects Agency under USAF contracts F30602-93-C-0177 and F30602-93-C-0028. The views and conclusions are the authors' and should not be interpreted as the official opinion or conclusions of the US Government, the USAF, Rome Laboratory, or ARPA.

KQML (Knowledge Query and Manipulation Language), is being developed in conjunction with a number of university and industry laboratories under the ARPA Intelligent Information Integration program and the Knowledge Sharing Initiative.

These two components, when integrated with other intelligent information system components being developed at USC ISI (SIMS) and at UCLA (CoBASE), provide intelligent access to distributed information sources in a fault tolerant and cooperative manner supporting military planners. The SIMS (Arens 1992) and CoBASE (Chu & Chen 1994) systems are described elsewhere.

ARPI Information Agent

This section describes the basic architecture of an Information Agent --- a knowledge server or source capable of handling all requests for information in a given domain, in this case, the transportation logistics planning domain. We have constructed an Information Agent prototype based on the Loom Interface Module (LIM). Using LIM, we have constructed an Information Agent (see Figure 1) which mediates between knowledge structures defined for use by intelligent system components and database structures. This information agent responds to queries and other commands which operate upon knowledge structures and translates them to the appropriate target system, e.g., SQL queries and data manipulation commands. This information agent has been used to support experimental representations of transportation assets (e.g., planes and ships), geographical locations (e.g., airports and seaports) as well as transportation relevant information about forces and transportation schedules. This LIM information agent is used in conjunction with the CoBASE and SIMS systems described elsewhere to provide a flexible and distributed cooperative intelligent information agent for transportation data which can be accessed at each of these interface points. If only mediation to shared representations is desired, the LIM information agent can be accessed directly; if information access planning is required the SIMS agent can be accessed; finally, if cooperative processing is desired, CoBASE can be used as the point of contact. All three systems can be accessed independently depending on the desired functionality. The Knowledge Query and Manipulation Language is used to support this level of communication transparency.

We have built an Information Agent prototype which involved the integration of the three knowledge-base/database components: LIM, SIMS, and CoBASE and focused upon the data and information collected for the transportation logistics domain. The prototype also tested the robustness of its three component systems in a realistic

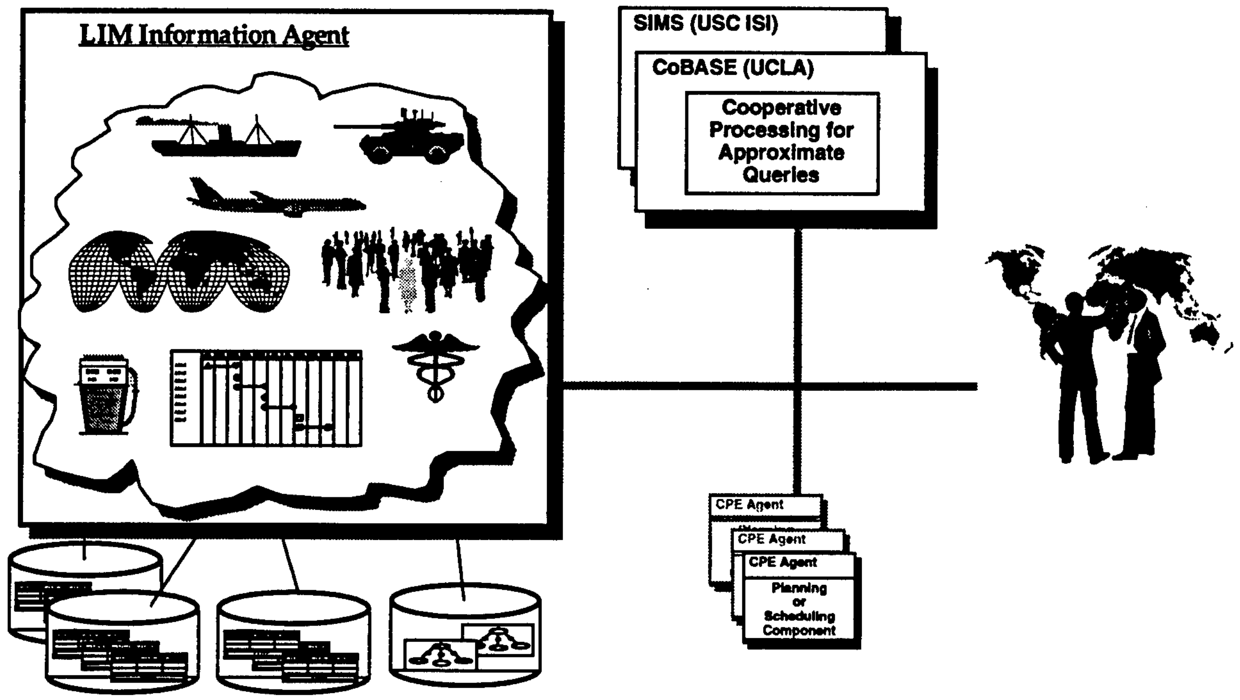

LIM Information Agent
SIMS (USC ISI)
CoBASE (UCLA)
Cooperative Processing for Approximate Queries
CPE Agent
CPE Agent
CPE Agent
Planning or Scheduling Component

Figure 1. LIM Information Agent. LIM provides domain relevant representations of transportation assets and other resources in a high-level representation. It mediates between the storage structures and the representations used by other intelligent agents.

information environment. Performance of tasks in the transportation planning domain typically requires access to data stored in a multiplicity of databases, by people (or computer systems) unfamiliar with their specific structure and contents. It is thus necessary to provide for the possibility of retrieving required data using a uniform language, independently of where the data is actually located and how complicated the actual process of retrieving it may be.

The Information Agent architecture currently address separate aspects of this problem. This prototype united them into one system that:

- accepted a query in an extension of the Loom language,
- relaxed the query, if appropriate, to enable retrieval of additional information of relevance to the user,
- planned a series of queries to databases and data manipulations that
- brought about the retrieval and/or computation of the
- requested data, and finally
- execute the plan, issuing the necessary queries to the appropriate databases, and returned the resulting data to the user

LIM, SIMS, and CoBase have been combined in various ways, including both a single Common Lisp program which shared one Loom model of the application domain and the databases as well as a distributed Information Agent architecture in which the LIM Information Agent, acting as a server, was at a remote site. Queries were submitted in the Loom language, extended by the approximation operators supported by the CoBASE system. CoBASE translated the user's query into one in the standard Loom language. SIMS broke down the resulting query into a series of LIM queries (again in the Loom language), each restricted to a single databases. The databases were accessed over a network, using the LIM database interface.

KQML Agent Communication Language

This section provides a brief overview of the agent communication language used in the Information Agent architecture. Many computer systems are structured as collections of independent processes, frequently distributed across multiple hosts linked by a network. Database processes, real-time processes and distributed AI systems are a few examples. Furthermore, in modern network systems, it should be possible to build new programs by extending existing systems; a new small process should be conveniently linkable to existing information sources and tools such as filters or rule based systems.

One type of program that would thrive in such an environment is a mediator (Wiederhold 1992), or information agent in this paper. Mediators are processes which situate themselves between "provider" processes and "consumer" processes and perform services on the raw information such as providing standardized interfaces; integrating information from several sources; translating queries or replies. Mediators are becoming increasingly important as they are commonly proposed as an effective method for integrating new information systems with inflexible legacy systems.

Standards and intercommunication approaches such as CORBA, ILU, OpenDoc, OLE, etc., are efforts that are often promulgated as solutions to the agent communication problem. Driving such work is the difficulty of running applications in dynamic, distributed environments. The primary concern of these technologies is to ensure that applications can exchange data structures and invoke remote methods across disparate platforms. Although the results of such standards efforts will be useful in the development of software agents, they do not provide complete answers to the problems of agent communication. After all, software agents are more than collections of data structures and methods on them. Thus, these standards and protocols are best viewed as a substrate on which agent languages might be built.

KQML is a language and a protocol that supports this type of agent communication specifically for knowledge-based systems or information agents. It was developed by the ARPA supported Knowledge Sharing Initiative (Neches et al. 1991, Patil et al. 1992) and separately implemented by several research groups. It has been successfully used to implement a variety of information systems using different software architectures.

KQML is a layered agent communication language (Finin et al. 1994; Finin et al. 1995; Mayfield et al. 1996). The KQML language can be viewed as being divided into two layers: the content layer, and the message layer or the communication layer. The content layer is the actual content of the message, in the agent's representation language; in the Information Agent described in this paper the content language was an extension of the Loom language developed under the Planning Initiative. KQML can carry any representation language, including languages expressed as ASCII strings and those expressed using a binary notation. All of the KQML implementations ignore the content portion of the message except to the extent that they need to determine where it ends.

The communication level encodes a set of features to the message which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication. It also determines the kinds of interactions one can have with a KQML-speaking agent. The primary function of the communication layer is to identify the protocol to be used to deliver the message and to supply a speech act or performative which the sender attaches to the content. The performative signifies that the content is an assertion, a query, a command, or any of a set of known performatives. Because the content is opaque to KQML, this layer also includes optional features which describe the content, e.g., its language.

Conceptually, a KQML message consists of a performative, its associated arguments which include the real content of the message, and a set of optional arguments which describe the content in a manner which is independent of the syntax of the content language. For example, a message representing a query about the location of a particular airport might be encoded as:

```
(ask-one :content (GEOLOC LAX (?long
    ?lat)) :ontology GEO-MODEL3)
```

In this message, the KQML performative is ask-one, the content is (geoloc lax (?long ?lat)) and the assumed ontology is identified by the token :geo-model3. The same general query could be conveyed using standard Prolog as the content language in a form that requests the set of all answers as:

```
(ask-all    :content
            "geoloc(lax,[Long,Lat])"
            :language standard_prolog
            :ontology GEO-MODEL3)
```

Loom Interface Module

LIM acts as an intermediary between a Loom application and one or more DBs. The inter-relationships among the various components of the overall system are illustrated in Figure 2. LIM uses the DB schema, building a Loom representation of the schema based on this information. Subsequently, in response to a query or update request from a Loom application that requires access to the DB, LIM parses the request and generates the appropriate data manipulation language (*DML*) statements for the DBMS; in the case of a query, it then processes the tuples returned to it by the DB into the form requested by the application. The details of the design and implementation appear elsewhere (Pastor & McKay 1994; Pastor, McKay & Finin 1992).

Processing within LIM is directed by a multi-layer KB architecture that is built in a mixed-initiative process. Figure 3 depicts the layers in this architecture. The Semantic Mapping KB (***SMKB***) is an isomorphic representation of the DB schema; it defines one Loom concept for each table and one Loom relation for each column. Application KBs (***AKBs***) define view-concepts which are concepts or objects in the domain and refer to concepts and relations in the SMKB. Within the ARPI Information Agent, concepts such as *Seaport* are defined over underlying SMKB primitive data elements. View-concepts in the AKB do not necessarily map in any simple way to the tables in the DB, and can have arbitrary hierarchical structure. Connections to the DB are implemented via ***DB-mapping*** declarations, in which a concept-role pair in the AKB is mapped to a SMKB role. View-concepts are checked at definition time to assure that they specify an unambiguous database query and, if declared to be updatebale, are unambiguously so. For updates, LIM determines whether the resulting DB action should result in an insert or an update.

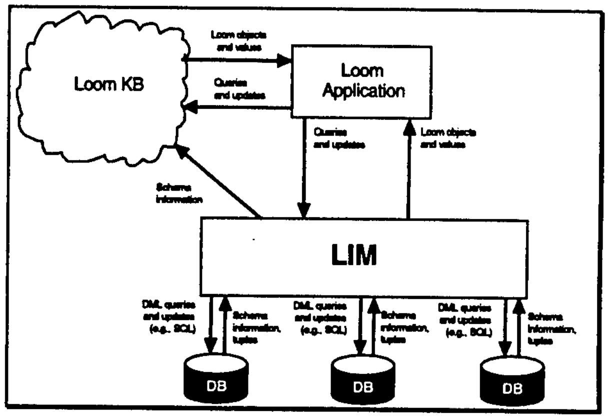

Loom KB
Loom objects and values
Queries and updates
Loom Application
Queries and updates
Loom objects and values
Schema information
LIM
DML queries and updates (e.g., SQL)
Schema information, tuples
DML queries and updates (e.g., SQL)
Schema information, tuples
DML queries and updates (e.g., SQL)
Schema information, tuples
DB
DB
DB

Figure 2. LIM Overview

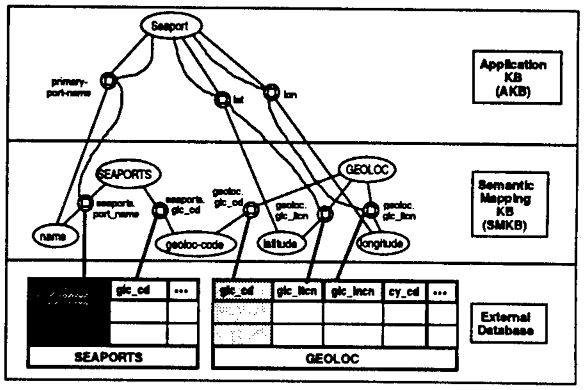


Figure 3. LIM Knowledge Base Architecture

LIM, given a query or update request involving a concept in the SMKB or AKB, first obtains schema mapping information from the SMKB, then translates the request into an equivalent DML statement, submits the statement to the DBMS and assembles the result; and finally (for a query), restructures the returned tuples as necessary, generating any KB structures required to satisfy the query. With regard to the last point, a fundamental principle of LIM is that KB structures are created only on demand: queries are satisfied without creation of KB objects whenever possible, to minimize overhead and bookkeeping. Control over object creation is entirely at the discretion of the application.

A LIM query consists of a list of output variables to be bound, and one or more statements that produce sets of bindings for these variables. It is easily determined from the positions of variables in the output list and the query expressions whether a particular output variable corresponds to a role value or a concept. For a variable corresponding to a role value, the value retrieved from the DB can be returned to the application, possibly with some conversion due to the differences between semantic types used in the KB and simple DB types. For a variable corresponding to a concept, however, the application will expect to have returned to it an instance of that concept; this requires that LIM be capable of creating Loom instances using values retrieved from the DB. LIM's object generation module extracts from the returned tuples all values requested specifically for the purpose of building Loom objects, creates the objects, and returns them to the application. In the Information Agent, the result, either a set of tuples or instance objects is then described in a KQML message using a content expression to desribe each tuple or object instance.

LIM uses a few different caching schemes, for two purposes. The first purpose is the conventional one of improving performance; the second is related to preserving referential integrity. When a user queries LIM for an instance of a view-concept, and then subsequently queries for an instance with the same key values, it is usually the case that one expects the *same* KB object to be returned in both cases. For this reason, LIM checks the Loom instance database (ABox) prior to creating instances. Given an object query, after submitting a query to the IDI and receiving return values, LIM queries the Loom ABox before creating a new instance. If the view-concept that is to be the type for the instance has keys defined, LIM uses these (in conjunction with Loom's indexing capabilities) to speed the search; otherwise, all values are used. This mechanism is also used to support incremental creation of object instances over several LIM queries. Other caching strategies avoid reissuing the same query.

Note that "ABox cache" checking is *not* an efficiency measure: on the contrary, it carries a performance penalty that can become significant on extremely large queries, e.g., many hundred to several thousand objects. For this reason, and because of situations such as dynamic DB contents where ABox cache checking is undesirable, it is controllable both globally and at the individual query level.

LIM Example

Let us presume that an application requires information about the location of various seaports. In the databases, information about seaports is stored in a table called SEAPORTS, and information about geographic locations in a table called GEOLOC. The various KB layers representing the mapping from application to DB are shown in Figure 3. The bottom panel shows a simplified tabular representation of the schema definitions for the two tables, SEAPORTS and GEOLOC. The middle panel shows the SMKB concepts representing the two tables. The SMKB definition is:

```
(defconcept Geoloc
  :is-primitive
  (:and db-concept
      (:the Geoloc.Glc_cd Geoloc_Code)
      (:the Geoloc.glc_lncn Longitude)
      (:the Geoloc.glc_ltcn Latitude)))
```

The top panel shows a simple application-level concept derived from information in both DB tables. The following is the Loom concept definition for the AKB concept seaport:

```
(defconcept seaport
  :is-primitive
  (:and View-Concept
      (:the primary-port-name string)
      (:the lat latitude)
      (:the lon longitude)))
```

This is mapped to the DB by making additional declarations, which are stored as assertions in the Loom KB. Queries can be posed referencing either the SMKB or the AKB. For example, the query:

```
(db-retrieve (?name)
  (:and
      (Seaports ?port)
      (Geoloc ?geoloc)
      (Seaports.Glc_cd ?port ?geocode)
      (Geoloc.Port_Code ?geoloc
?geocode)
      (Seaports.port_name ?port ?name)
      (Geoloc.Country_State_Code
?geoloc "DP")
      (Seaports.Clearance_Rail_Flag
?port "Y")))
```

("What are the names of seaports in Dogpatch that have railroad capabilities at the port?") can be posed using the SMKB. The SQL generated by LIM for this query is:

```
SELECT DISTINCT RV1.name
FROM SEAPORTS RV1, GEOLOC RV2
WHERE RV2.glc_cd = RV1.glc_cd
  AND RV2.country_state_code = 'DP'
  AND RV1.clearance_rail_flag = 'Y'
```

The values returned are a set of tuples:

```
("Cair Paravel" "Minas Tirith"
"Coheeries Town" "Lake Woebegon" "Oz")
```

The query:

```
(db-retrieve ?port
  (:and
    (seaport ?port)
    (primary-port-name ?port "Oz")))
```

("Return a seaport object for the port whose name is 'Oz'") can be posed using the AKB. The SQL generated for this query is:

```
SELECT DISTINCT RV1.name,
       RV2.latitude,
       RV2.longitude
FROM SEAPORTS RV1, GEOLOC RV2
WHERE RV2.glc_cd = RV1.glc_cd
  AND RV1.name = 'Oz'
```

The value returned by this query is an object whose Loom definition is:

```
(TELL
  (:ABOUT SEAPORT59253
    SEAPORT
    (LON 98.6)
    (LAT 3.14159)
    (PRIMARY-PORT-NAME "Oz")))
```

The Information Agent uses a slightly different form of the above s-expressions for sets of tuples and instances to return answers to other agents. The particulars are outside the scope of this paper.

Information Agent Performance

We have defined metrics for performance evaluation and have been using them continuously throughout the development of the Information Agent for both KQML and LIM. The performance model for KQML is described elsewhere. The LIM Information Agent metrics include components of total execution time:

- *Augmentation*: CPU time required to add concept-derived restrictions to the query
- *Translation*: CPU time required to translate LIM query into internal canonical form
- *Query Generation*: CPU time required to translate internal canonical form into DML
- *Connection*: Real time required to establish connection with DBMS server
- *Execution*: Real time required to execute the query on a DBMS server
- *Collection*: CPU time required to accumulate results of the query
- *Object Generation*: CPU time required to post-process results including creation of Loom instances if appropriate
- *ABOX Cache*: CPU time required to search Loom instance database (Abox) to prevent creation of duplicate instance (included in Object Generation time)
- *Total Execution Time*: Sum of all the above excluding the ABOX Cache time

Using benchmarks derived from queries collected during early uses of the LIM Information Agent under ARPI technology integration experiments, we have developed a performance profile. The queries vary from small functional tests to the retrieval of large view-concepts for force modules for combat services and combat services support; each force module is retrieved independently. The benchmark consists of executing the LIM query 25 times with all caching turned off, i.e., queries are sent to Oracle each time. Figure 4 below compares performance from initial baseline performance in November 1992, LIM 1.1 performance in May 1993, LIM 1.2 performance in May 1994, and LIM 1.4 performance in May 1995.

It should be noted that these queries retrieve and create a significant number of object instances with relatively large numbers of slot value sets; performance is now well below one second for total execution time. One test results in over 700 force module instances created and about 60,000 attribute value sets within those instances. The total execution time for this query set as of May 1994 was on the order of ten minutes; current execution time (LIM 1.4) is approximately 1 minute 25 seconds.

We have improved LIM performance dramatically over the course of the Planning Initiative. The most notable improvements are due to the following factors:

- We now use of faster Loom primitives where available, or adopted them they became available, which has dramatically improved basic execution speed.
- In cases where repeated use of the same Loom inferencing chain might otherwise result, we cache information retrieved from Loom knowledge base to "memoize" knowledge base access
- We use improved fundamental data structures within the LIM database interface
- We improved algorithms; for example, it is now possible to specify that results be returned from the Oracle interface in batches, rather than tuple-at-a-time.
- We tuned fundamental data structures extensively for speed and space.

All data has been collected on SUN SPARC 2 CPU with 96MB memory. Since a component of LIM processing is due to actual execution of queries on a remote Oracle database, the times measured below are dependent on the actual system used to support Oracle as well.

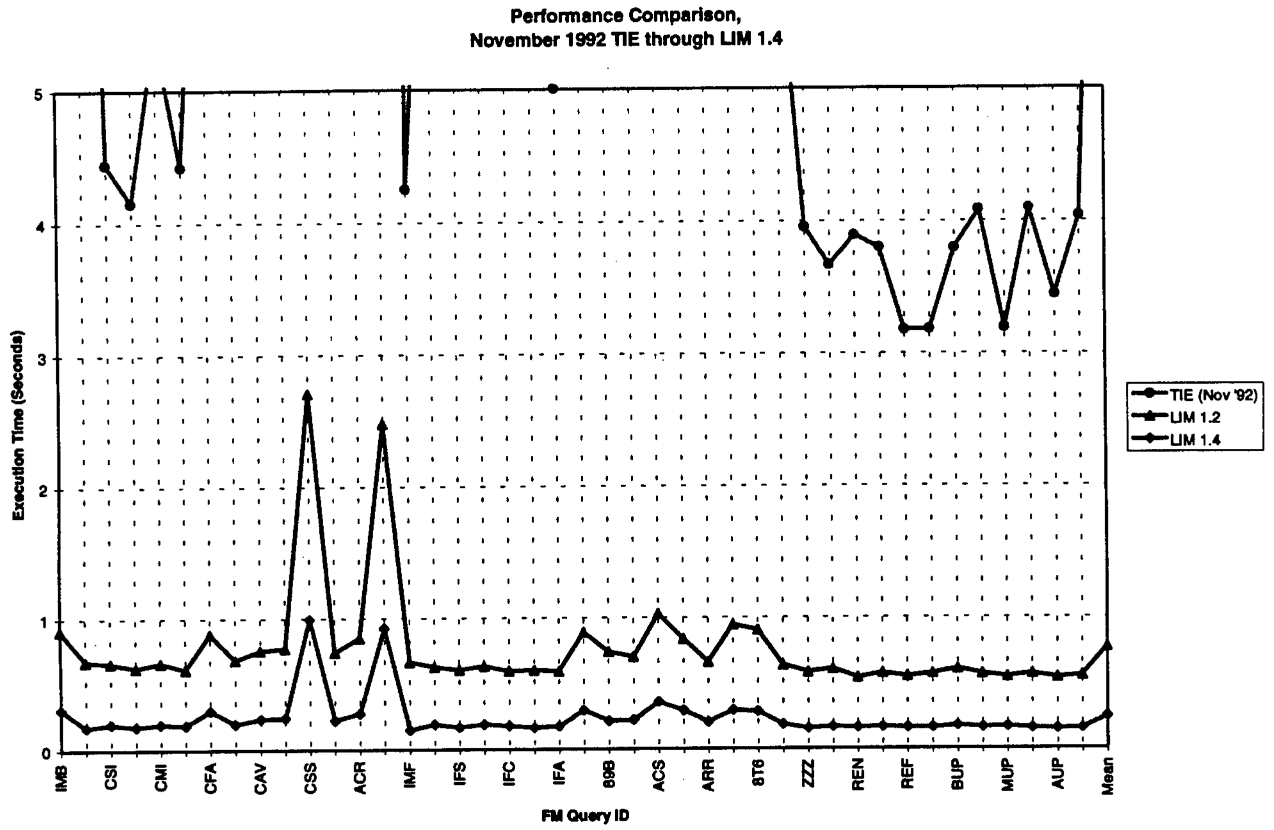


Figure 4 LIM Information Agent performance summary.

We have begun to compare critical portions of the LIM execution profile, specifically, object creation and slot filling algorithms, with those available in a commercial product expert system shell (ES). Initial results are preliminary, but, illustrate some of the performance issues for Information Agents. LIM, implemented in Common Lisp and Loom, outperforms the commercial ES, one written in C. We have measured the performance of LIM with that of the expert system shell (ES) on two query sets, showing both DB execution and Object Creation. The database and queries were selected from a set developed in another project. We have observed about a 10:1 ratio for object creations per second of DB execution time in favor of LIM. In addition, at least a 10:1 ratio for object creations per second of object creation time, again in favor of LIM. The most accurate metric is based upon slot-value sets, since this accurately reflects the total amount of data being transmitted and processed; in our initial measurements this is significantly over 10:1.

Conclusion

We have described an Information Agent architecture in which two key components are an agent communication language and a collection of information agents. Specifically, we have described KQML, the communication portion of the agent communication language. In addition, we have described the LIM Information Agent which interfaces the Loom knowledge representation and reasoning system with relational databases. We have described some of the performance measures we have developed for the LIM Information Agent and reviewed some of our current performance results. One set of preliminary measurements indicates that the performance of object creation and manipulation components for information agents is a key measure, and, that LIM outperforms at lest one widely available expert system shell. We intend to follow up on this result and investigate this measurement approach further.

The LIM Information Agent relies on a view-concept model which uses a knowledge representation language, Loom, to define the semantic schema of a database. This definition has two levels, each of which is of utility to a knowledge-based application. The semantic mapping layer defines the relevant concepts supported by the database domain; in our current knowledge bases, the semantic mapping layer adds semantic types to the automatically-generated schema model. We envision additional information in the semantic mapping layer, including composites of database objects which form larger conceptual structures. The view-concept model includes an application-specific layer that defines the mapping between an application domain's conceptual structures and the semantic definition of database concepts. We believe that the structured approach embodied in the view-concept model significantly elucidates the knowledge-base-to-database interface problem.

The system described above has operated in single module form where each of SIMS and CoBASE have LIM loaded into the same Common Lisp program, independently using LIM as a server remotely over the network (local or internet) and together in an architecture where SIMS acts both as a server to CoBASE and as a client to multiple LIM-based knowledge agents available on the network in a mixture of local and internet configurations. The basic issue addressed in all of the above work is the actual running of demonstrations in a reliable and repeatable manner. This goal forces one to pay attention to details of normal operations including performance and interpretation. Further, without the attempt to integrate, some of the issues described above would have not been identified as well as other integration issues.

Acknowledgments

The authors wish to acknowledge the contributions of the LIM development team, specifically Rebecca Davis, Robin McEntire, Rich Fritzson, Tim Finin, and Barry Silk (US Government). Earlier contributors to the first version of the Intelligent Database Interface include Tony O'Hare, currently at IBM, Research Triangle, and Larry Travis of the University of Wisconsin. Also, we acknowledge the useful comments and suggestions of users of LIM specifically Yigal Arens and Craig Knoblock of USC ISI, Wes Chu, Berthier Ribiero and Galdys Chow of UCLA, Scott Fouse, Nancy Lehrer, Mark Hoffman and Louis Rumanes of ISX, and, Glenn Abrett and Mark Burnstein of BBN. Finally, we have appreciated participating in the ARPA Rome Lab Planning Initiative and acknowledge the stimulating interactions with the participants in the program, Steve Cross and Tom Garvey the APPA Program Managers as well as the staff and program management at Rome Laboratory specifically Ray Liuzzi, Lou Hoebel, Don Roberts and Nort Fowler.

References

Yigal Arens 1992. Planning and Reformulating Queries for Semantically-Modeled Multidatabase Systems, In Proceedings of the First International Conference on Information and Knowledge Management.

Wesley W. Chu and Q. Chen 1994. A structured approach for cooperative query answering. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):738--749.

Tim Finin, Richard Fritzson Don McKay and Robin McEntire 1994. KQML as an Agent Communication Language, In Proceedings of the Third International Conference on Information and Knowledge Management, ACM Press.

Tim Finin, Yannis Labrou, and James Mayfield 1995. KQML as an agent communication language, In Jeff Bradshaw (Ed.), ``Software Agents", MIT Press, Forthcoming.

Robert MacGregor and Raymond Bates 1987 The Loom Knowledge Representation Language, Proceedings of the Knowledge-Based Systems Workshop, St. Louis, Missouri.

James Mayfield, Yannis Labrou, and Tim Finin 1996. Evaluation of KQML as an Agent Communication Language. In Intelligent Agents Volume II -- Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages. M Wooldridge, J. P. Muller and M. Tambe (eds). Lecture Notes in Artificial Intelligence, Springer-Verlag.

R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout 1991. Enabling technology for knowledge sharing. AI Magazine, 12(3):36 -- 56.

Jon Pastor and Don McKay 1994. View Concepts - Persistent Storage for Planning and Scheduling, Proceedings of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop, Tucson, AZ.

Jon Pastor, Don McKay and Tim Finin 1992. View-Concepts: Knowledge Based Access to Databases. In Proceedings of the First Conference on Information and Knowledge Management.

R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches 1992. The DARPA Knowledge Sharing Effort: Progress Report. In B. Nebel, C. Rich, and W. Swartout, editors, Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference.

Gio Wiederhold, 1986 Views, Objects, and Databases, *IEEE Computer*, 19(12):37–44.

Gio Wiederhold, 1992 Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3):38-49.

10. INTELLIGENT RESOURCE AGENT ARCHITECTURE

Intelligent Agent Integration Technology

Prepared by:
Donald P. McKay
Principal Investigator
Lockheed Martin
590 Lancaster Avenue, Frzer, PA 19355-1808
Donald.P.McKay@lmco.com, (610)407-3527

This information was developed under a separate Lockheed Martin contract. It is provided here as related material demonstrating the utility of an architecture for intelligent agents using KQML. The paper will appear in the Proceedings of the Cooperative Information Systems '97.

10.1 Introduction

The proliferation of large network systems in general, and the exponential growth of the World Wide Web (WWW) in particular, have resulted in nearly unlimited opportunities for the large scale gathering, creation, and sharing of information. One unfortunate aspect of such systems, and especially of the WWW, is that they are still relatively unstructured collections of heterogeneous resources.

Imagine a third-grade teacher who wants to find current online resources to supplement a planned space science unit. This teacher is not sophisticated technically, is unfamiliar with the subtleties of using search engines like Lycos and Infoseek, and has little time available for class preparation. A keyword search for "space and planets" submitted to Infoseek produces a list of literally hundreds of possibilities. Since standard WWW search interfaces are purely keyword-driven, and have no notion of the purpose of the search ("material suitable for a third-grade science class") or any particulars about the user ("collaborative activities ideal; prefer images to text; special interest in Jupiter"), the list contains mostly irrelevant or uninteresting information, and—even worse—may miss items that would be nearly ideal. For example, twelve pages down in the list is the perfect resource—a collaborative activity sponsored by NASA that would really get her students excited and involved—but it is buried in the sheer volume of data. A recent set of spectacular Jupiter images doesn't even appear in the list, because the captions happen not to contain the search terms ("space" and "planets"). Very few valuable, interesting, or relevant resources are found in the 15 minutes that the teacher has before her next class, and she dismisses the WWW as a waste of time and energy.

To assist users like this teacher in obtaining useful, structured information—rather than a chaotic mass of largely irrelevant data—we have developed a set of specialized intelligent resource agents that serve as mediators between the user and Internet resources. These agents interact cooperatively in a distributed environment, and are accessible from within a WWW infrastructure. This system will eventually employ over a dozen agents to perform a variety of tasks related to supporting and enhancing the use of the Internet as an educational tool. Agents in the current implementation perform tasks ranging from remote database access, to dynamic composition of HTML pages for the display of appropriate resources, to customization of both the form and content of those pages to an individual user's preferred styles. An early prototype in the domain of K-12 education is in use with teachers through the Defense Advanced Research Projects Agency (DARPA) Computer Aided Education and Training Initiative (CAETI) program.

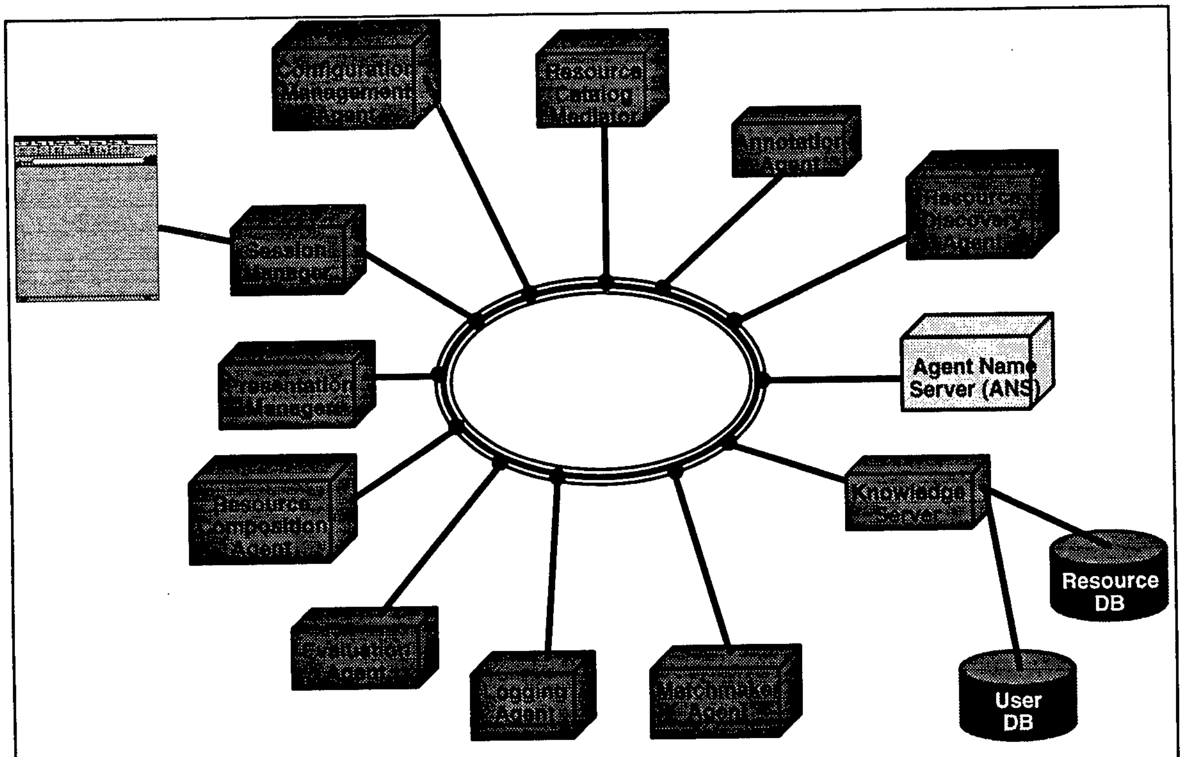


Figure 1: Intelligent Agent Architecture

In this paper, we describe the components of the overall system architecture and its implementation, including the agents themselves, the communication protocols, and the shared domain model. This application is just one example of the power of well-designed distributed agent architectures to maximize "plug-and-play" compatibility and reuse.

10.2 System Architecture

10.2.1 System Components

The first step in constructing a large-scale system is identifying the key functionality required, and ascribing each functional unit to a separate module. In the case of an agent architecture, these modules are embodied as independent applications, interacting cooperatively in the distributed environment. The overall system requires the coordinated application of a wide range of conventional technologies, including information retrieval, knowledge bases, databases, as well as domain-dependent technologies such as intelligent custom generation of HTML pages to suit purpose and preferences of the individual user. Our analysis of the problem domain suggested that the requisite functionality was best factored into approximately a dozen agents, as depicted in Figure 1. The resulting system is accessible from within a WWW infrastructure, interfacing with the users via standard WWW languages and protocols—HTTP, HTML, JavaScript, and the Common Gateway Interface (CGI).

The overall Intelligent Resource Agent architecture consists of the agents themselves, common message exchange protocols and syntaxes, and a common domain description or *ontology*. Information about the users, and meta-information about Internet resources, is maintained in external persistent stores, and accessed via specialized agents ("knowledge servers"). This information is used by the agents to customize interfaces, personalize searches for web resources, and generate appropriate customized displays of search results. The knowledge server architecture itself is described in [Pastor 92, Pastor 94, McKay 96]; it is based on the concept of a mediator between information consuming agents and information providing agents [Wiederhold 95, Wiederhold 92].

10.2.2 Agent Descriptions

The agents of Figure 1 fall loosely into three categories:

intelligent resource agents,

- generic intelligent systems agents, and

core infrastructure agents.

Intelligent resource agents implement functionality that is appropriate for cooperative information systems, applied to the problem of information resource identification and use in the WWW and similar areas (e.g., digital libraries), but not necessarily generalizable outside the information resource domain. Generic intelligent systems agents are components that are generally reusable across instances of the intelligent resource agent architecture, and are designed to be easily customizable for a given application. Examples of this class of agent include:

evaluation agents, which are used to measure and report on system and resource utilization, and information mediators, which provide persistent storage in terms of the common ontology.

Core infrastructure agents are components that are required to support inter-agent communication and system operation.

Intelligent Resource Agents work together to enable appropriate, timely and customized access to distributed information sources. In our particular case, these agents target multi-media Internet resources. This includes creating customized interfaces to assist the user in finding potential sources of information, identifying relevant information, and forming new resources based on this information which are presented to the user in an appropriate manner (e.g., a summary of findings customized for the user).

The **Resource Discovery Agent** retrieves and collects information from a variety of dispersed, multimedia information resources. This agent is responsible to keeping track of resources in this dynamic environment—i.e., resources may change, move to different locations, or even disappear at any time. This collected information is a primary source for the resource catalog. In our implementation, the resource discovery agent is akin to a *web- crawler*, searching the Internet based on known educationally-relevant seed sites, and indexing resources. The resource discovery agent contributes some meta-data about the resource, along with content-based indexing, to help populate the catalog.

Resource cataloging technology then provides the means for representing, integrating, and acquiring knowledge about the collection of information resources through an analysis of the objects and relationships among them. The **Resource Catalog** provides a conceptualization and description of information resources which adds semantic structure for a domain-specific use. The **Resource Catalog Mediator** allows users to quickly, intelligently, and productively access information sources. Where most search engines available on the WWW search strictly on content, via Boolean combinations of keywords, the resource catalog provides a database of structured meta-information on relevant resources. This allows the users to target their searches in useful ways, and does not rely solely on matching key words in the text. For example, a third-grade teacher can ask for resources specifically targeted for the appropriate grade-level. What distinguishes this resource catalog from conventional digital libraries is that it is a highly-dynamic mix of "automated"—e.g., captured by resource discovery—and "user driven" collections, a live and evolving entity to which users contribute new resources. User contributions may be may be original works. interesting resources found on the net that are not already in the collection, or even comments on existing resources via annotations.

Support for user-annotation of resources in the collection is one of the most unique and interesting aspects of this catalog. Annotations become a searchable portion of the catalog, permitting one teacher to benefit from the experience of others, and even to search directly on aspects of the annotations. Users are encouraged to contribute, and are provided tools for producing, both structured (e.g., rankings) and unstructured (e.g., free-text) annotations

Matchmaking extends the concept of a resource beyond conventional online multimedia resources to include human resources, people who are available as mentors, experts, or potential collaborators. A teacher can now contact an expert on comets who will answer questions posed by her inquisitive third graders. Users can register their interests and their willingness to act in various mentoring and/or collaborative relationships; other users can then search for "people" resources to assist them in planning or executing classroom activities. Matchmakers come in two flavors: *reactive* and *proactive. Reactive* matchmakers respond to a specific user request, while *proactive* matchmakers observe the online behavior or interest of users and try to arrange matches. For example, a proactive matchmaker might notice that ten users are teaching Space in their third grade class, and "suggest" via e-mail that they start a discussion group or consider participating in a collaborative activity on this topic.

Resource Composition provides customized presentation of web resources, interfaces to the intelligent agents, or results from search queries. When the user queries the resource catalog, the results are presented in an appropriate fashion based on user preferences. The object is to organize the resources into a form that suits the functional and aesthetic requirements of the individual user, rather than to the typical output of search engines: simple and uninformative hit lists based on titles, keywords, or source location. The teacher interested in images of Jupiter, finds resources which are image-rich, as well as short summaries of what each resource provides, and comments from other users on how they used that resource.

In a similar fashion, interfaces to the intelligent agents (e.g., catalog search) are customized for ease of use and comfort-level to the particular user, based on profiles that include both information about the user (e.g., *teacher* or *student* status) and expressed preferences (e.g., brief or detailed summary results). For example, teacher and student interfaces might have a different "look," K-3 interfaces might differ from high school level interfaces, and so on.

The **Session Manager** and **Presentation Manager** are works in progress, designed to address the challenging problem of persistence within a WWW "session," and are discussed briefly in Section 10.4 on future directions.

The **Collection Agent** serves as a repository for information on system performance and utilization, fed by messages from all appropriate agents in the system, and even from HTTP server logs. Once raw data has been collected by the collection agent, this data is persistently stored for later use by other evaluation agents in the system. The **Evaluation Agent** operates on this data, providing a wide range of usage and performance statistics; these can then be used to track and evaluate relevant aspects of system and user behavior, and to tune both performance and appropriateness.

The **Knowledge Server** represents a class of agents that provides the other agents with persistent information; in the system described here, persistent information is needed about users, and about resources in the catalog. A Knowledge Server mediates between the applications and external persistent stores, providing an object-based view of the raw external data, in terms of the domain ontology. The current Knowledge Server is implemented in Common Lisp using the Lockheed Martin Interface Module (LIM) and the Loom knowledge representation language; a replacement, to be written in the Java language, is currently in the design stage.

The other agents represented in the architecture are components of the underlying infrastructure. The **Agent Name Server (ANS)** keeps track of agent registration, and permits agents to locate and address other agents by name (rather than, say, IP addresses and ports). The **Configuration Management Agent (CM Agent)** monitors the status of all agents in a defined agent configuration/system, and provides the ability to reconfigure that system and start, stop, suspend, or resume agents in the system.

10.2.3 Inter-Agent Communication Language

This system, like many other software systems, is structured as a collection of independent processes, distributed across multiple hosts linked by a network. The Knowledge Query and Manipulation Language (KQML) is used as the communication protocol among the agents [Finin 95, Finin 94a, Finin 94b, Mayfield 96]. KQML is a language and a protocol that supports extensible network programming, specifically for knowledge-based systems and intelligent agents [Neches 91, Patil 92].

Lockheed Martin's implementation of KQML, used in this current work, contains two primary layers. The outer layer, called a ***router***, provides message routing functionality to its associated application agent, and handles the establishment of all communication links to other agents in the system. The inner layer, called a ***KRIL*** (KQML Router Interface Library), provides a library of application-level interface routines. The ***router*** module makes use of the Agent Name Server (ANS—cf. Section 10.2.2) to determine addresses for agents and services within the system. Routers function *independent* of message content. Each agent has a separate router process; he router handles all incoming and outgoing message traffic for its associated agent.

The router is a separate module from the application, and it is necessary to provide a programming interface between the application and the router. This application program interface is called the ***KRIL***. While the ***router*** is a separate process with no understanding of the content of the KQML message, the ***KRIL*** is embedded in the application and has access to the applications methods for understanding the content of the KQML message—in this specific case, the objects that are passed among the agents. In addition, the ***KRIL*** has access to the application objects that must be transmitted from one agent to another. The ***KRIL*** is responsible for encoding these application objects into the communication objects that are carried in the body of KQML messages, and, conversely, for recomposing these communication objects into appropriate domain objects when a message arrives at the receiving agent. The ***KRIL*** will also compose the complete KQML message, ready for transmission. Using a set of verb-like tags, called ***performatives***, such as `ask-one`, `ask-all`, `register`, and `advertise`, these syntactically correct messages are created in the ***KRIL*** and are then handed to the ***router*** to be dispatched to appropriate agents in the system.

10.2.4 Shared Communication Ontology

In order for an agent to be able to "understand" the content of a message, it must be stated in terms that make sense to the agent: the nomenclature used by the sending agent must be comprehensible to the receiver. In human speech, we take for granted that references to objects, conditions, and other terminological entities are comprehensible to all parties in a conversation—that all parties share a set of concepts that have essentially the same characteristics in the minds of all participants. A ***shared communication ontology*** is a formalization of this notion, ensuring that when one agent refers to, say, a "resource", this evokes the same feature set, in terms of both structure and behavior, to the receiver as to the sender.

A key feature of our architecture is that it specifies the common ontology in a representation that subsumes the models of the individual agents. The practical implications of this are

any model required by any agent can be expressed in the common ontology;

- translations from the common ontology's formal representation are lossless with respect to the agents' models;

consequently, the agents are not constrained to employ the same modeling language—or, in fact, the same programming language.

As a result, it is possible to build hybrid systems consisting of modules developed in different languages, on different platforms, using different representations or *views* of a common set of objects; the Intelligent

Resource Agent system includes modules in C, C++,Common Lisp, Perl, and HTML, all interacting freely with one another without regard for representation details, language, or platform.

In our current system, while the internal program data structure representations used by different agents may be encoded differently due to language differences, it is typically the case that the agents' models are either isomorphic to, or proper subsets of, the communication ontology. This is neither necessary nor necessarily desirable, and in this application we have begun to introduce a separation between the *application* ontology manipulated by the agents themselves and the *communication* ontology used for inter-agent communication.

We view the ability to translate bi-directionally between the communication ontology and a collection of application ontologies as being critical to the long-term success of distributed agent architectures, since it places few—if any—restrictions on the kinds of applications that can participate as agents. An example of this principle is embodied in the Knowledge Server which is a *mediator* agent that provides access to databases in which persistent data for the catalog, among other things, is stored. The Knowledge Server is able not only to translate the flat structure of the relational model into the hierarchical structures of the communication ontology, but also to perform translations of encodings and modalities. For example, a database field that contained a string naming a specific MIME type (e.g., "image/gif") could be mapped to a conceptual type in the ontology that represented that MIME type (e.g., an object of type **GIFimage**. An encoding of this object, described in terms the communication ontology, would then be sent via a KQML message to the requesting agent. The significance of this observation is that other agents can be written in terms of the underlying ontology—a **GIFimage** object—rather than a specific text string—"image/gif".

10.3 Implementation

10.3.1 Intelligent Resource Agent Architecture

The current system consists of the following components:

- a Common Lisp-based knowledge server mapping from a Loom knowledge base to an Oracle database
- a collection of intelligent resource agents, primarily implemented in C++, that communicate with both the knowledge server and each other
- a communication infrastructure that supports inter-agent communication, which includes the router and KRIL functionality used by each agent, a number of utility agents, such as the ANS, that provide assistance for message traffic, and encoding/decoding routines that allow the transmission of application-level objects from one agent to another

Agents wishing to communicate with other agents are thus presented with a true object-oriented model of the communication acts, and are insulated from the details of KQML message composition and encoding.

The programmer API for database access consists of a handful of methods, for those agents requiring external data; these methods further encapsulate the KQML protocol so that requests for data are stated in terms of actions on objects of the desired type. All of the KQML and database functionality is defined at the level of abstract classes; objects requiring KQML transmission, or both KQML transmission and database access, are derived from these abstract classes and inherit the requisite behavior.

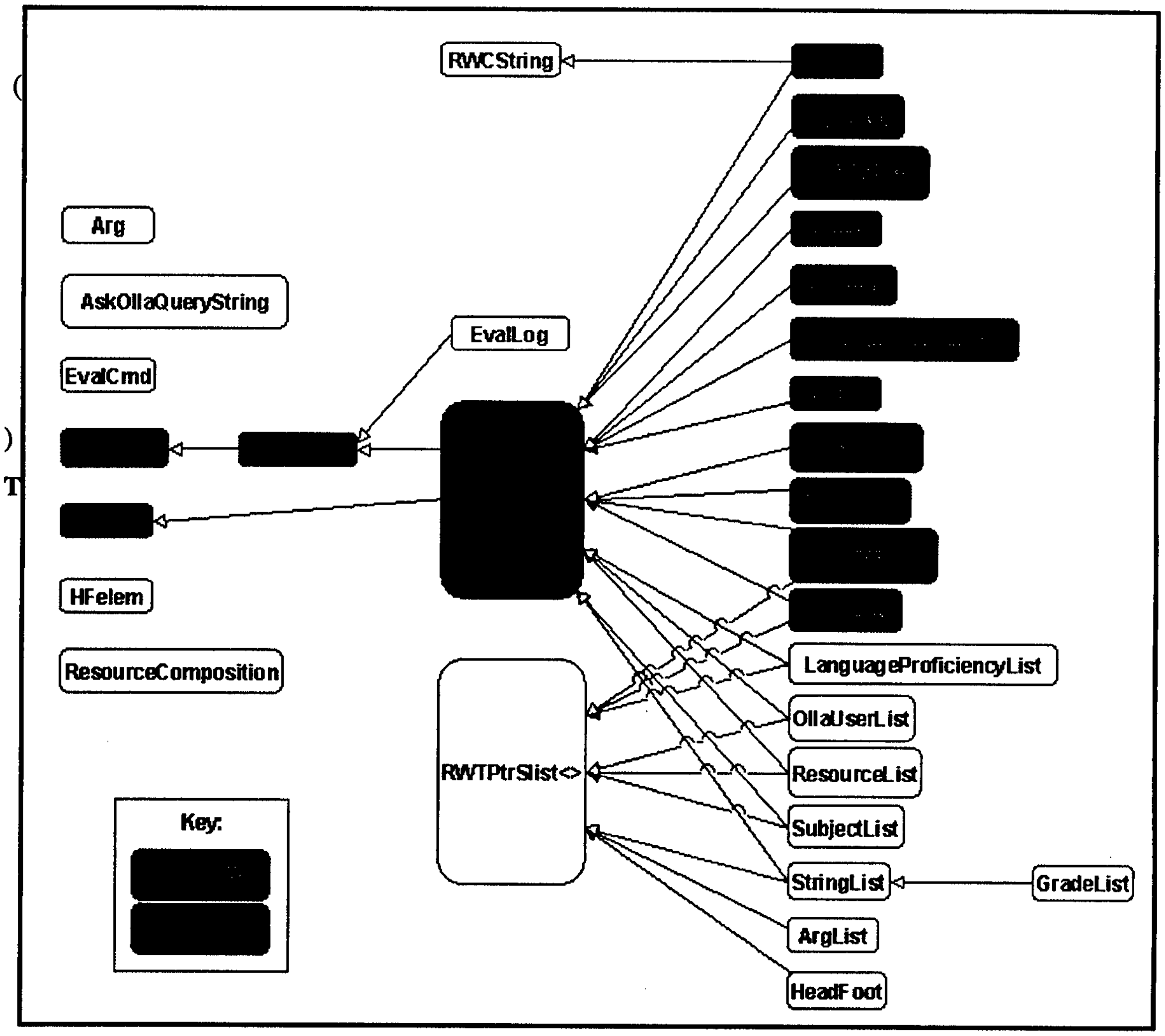


Figure 2: Shared domain ontology

In all present cases, as illustrated in Table 1, the C++ classes are structurally similar to the Loom concepts defined in the shared domain model—the application model is nearly isomorphic to the communication ontology. The C++ class definitions include not only the structural elements and the inherited KQML and/or database behavior, but also a standard set of methods for *encoding* and *decoding* C++ objects to and from a canonical ASCII representation used to represent objects in transmitted messages. While the form of this representation is actually arbitrary, as long as it can encode the application data structures unambiguously, it must be standardized among all agents in order for them to be "plug-compatible".

Given the close correspondence between the Loom domain model and the C++ object definitions for the application model, we chose to provide for automatic generation of the C++ classes from the Loom KB: both the data members and a working subset of the required function members (methods) are generated by a Common Lisp function that walks the Loom KB and processes concepts corresponding to externally-useful objects.

The C++ class architecture for our Intelligent Resource Agent system is depicted in Figure 2. Communication acts, and methods on application-level objects, are mixed to create hybrid methods that incorporate the power of both communication-based and object-based acts. Using these classes, a domain-relevant class such as `Resource`, which implements a resource object in a resource catalog, can be defined in terms of its unique defining attributes, and inherit its communication and data base aspects.

A critical point to observe about Figure 2 is that it depicts not only the structure of the knowledge-base objects used to represent the abstract ontology, but also the structure of the application-level objects in an object-oriented language, in this case C++. That is, the agents manipulate objects that are isomorphic, in terms of structure and taxonomy, to the underlying domain ontology. Table 1 shows a knowledge base definition, in the Loom language, side-by-side with the corresponding portion of the C++ class definition used when exchanging information about educational standards.

Each C++ class derived from `kqmlClass` defines a pair of C++ methods—`Encode()` and `Decode()`—to create and interpret (respectively) a canonical text representation that is used as the content language for KQML transmission; it also inherits a set of methods corresponding to a subset of the KQML performatives (communication primitives). In addition, some agents define higher-level methods that encapsulate the appropriate translation and encoding of application-level objects, KQML message formation, transmission, and reception; and decoding and creation of C++ objects. In fact, from an agent's perspective, it is manipulating instances of the application ontology classes, in some cases unaware that some of the objects it is manipulating have been retrieved from remote locations.

10.3.2 Application to K-12 Education

Figure 3 shows the interplay of key intelligent resource agents to assist our teacher in finding the

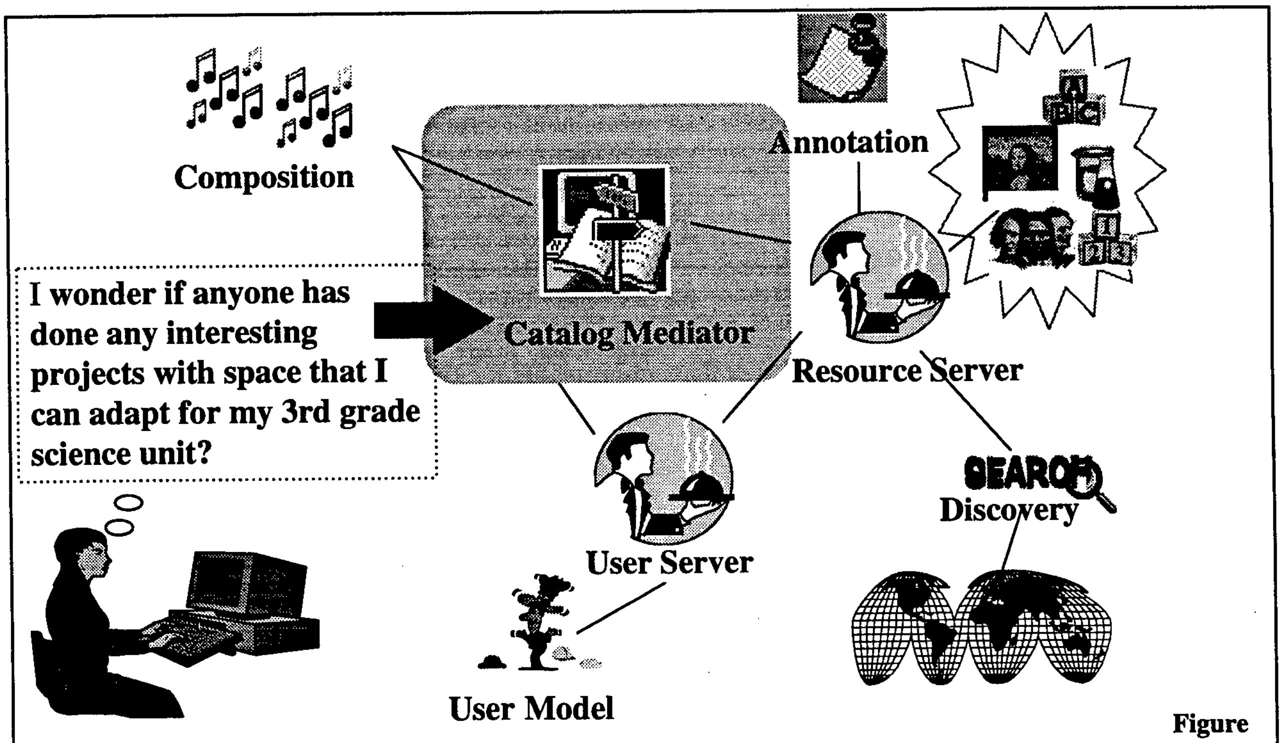

Composition
Annotation
I wonder if anyone has done any interesting projects with space that I can adapt for my 3rd grade science unit?
Catalog Mediator
Resource Server
SEARCH
Discovery
User Server
User Model

Figure 3: Resource agents work together to gather customized resources for the user and present them in an appropriate manner.

appropriate resources for her third-grade science class. The teacher makes a specific request through the form catalog form interface on her WWW browser, insulating her from the details of the underlying query language. The catalog mediator—implemented as a Common Gateway Interface (CGI) program—interprets the form submission, initiates the search based on her selections. The catalog mediator checks the collections created by the discovery agent, filters the results through the catalog, and then passes the resulting resource information to the composition agent. The composition agent queries the user server for information relevant to presentation of results, and returns a customized HTML page—based on the original query, the resulting resources, and the user information—to the catalog mediator. Finally, the page is displayed to the teacher, via the catalog mediator, in the client area of her WWW browser. Along with some excellent photo archives from the Hubble telescope, the teacher finds a wonderful *Mission to Mars* activity sponsored by NASA.

Early prototypes of this technology are currently in use with teachers in the DARPA sponsored CAETI (Computer Aided Education and Training Initiative) Department of Defense Educational Activity (DoDEA) test bed via the Online Learning Academy (OLLA), a World Wide Web (WWW) environment, which supports the use of telecomputing in the classroom [OLLA 96]. OLLA is currently deployed in four school complexes (akin to school districts) and in pilot classroom use with about twenty elementary school teachers. The intelligent resource agent architecture, and scenarios such as the one above, have been successfully demonstrated from within the OLLA infrastructure. The proactive matchmaker intelligent resource agent is the first of the intelligent resource agents to be deployed with OLLA (housed at the Franklin Institute Science Museum), connected by HTTP access, and in use by educators world-wide.

10.4 Conclusions and Future Directions

The intelligent resource agent architecture provides the basis for a scaleable and robust agent infrastructure. The key ideas include appropriate modularization of agents into reasoned software processes. In the area of information resource utilization on the WWW, we have demonstrated that the intelligent resource agent architecture is one effective way of providing tailored information services for specific applications, i.e., K-12 education and the effective use of internet resources.

Future development of this system includes refinement of the existing intelligent resource agents, as well as the addition of several new agents. Since we are working within the confines of the WWW, customization within a *session* has been challenging. To this end, **Session Manager** and **Presentation Manager** agents are under development. The **Session Manager** maintains persistence within a session, and the **Presentation Manager** controls customization of interfaces within a session. Both of these components will involve communication among processes on the client, as well as distributed client/server interactions.

In addition, we plan to continue with the general agent infrastructure that we demonstrated successfully within this application. We have begun to define aspects of communication ontologies under this project. Communication ontologies hide from application-level processes both the communication acts and the information exchange distribution necessary to achieve application-level goals. This is accomplished by including in an object's definition both the methods to be invoked on that object and the communication act necessary to satisfy the successful execution of each method. For CORBA-based implementations, this can be directly implemented via remote method invocation supported in CORBA. However, we envision significantly more stylized information exchange, including status and data item updates, as well as volunteering of newly developed information. A convenient and flexible information exchange mechanism is a key to the continued success of these systems.

We are just beginning to exploit this approach in an architecture for Information Resource Agents. We will further explore this approach and use both a formal definition of an ontology and a practical software implementation approach.

10.5 References

[Finin 95] T. Finin, C. Thirunavukkarasu, A. Potluri, D. McKay, and R. McEntire, On Agent Domains, Agent Names and Proxy Agents, *Proceedings of the ACM CIKM Intelligent Information Agents Workshop*, Baltimore, December 1995.

[Finin 94a] T. Finin, D. McKay, R. Fritzson, and R. McEntire. “The Knowledge Query and Manipulation Language for Information and Knowledge Exchange”, *Proceedings of the Third International Conference on Information and Knowledge Management* (CIKM’94), November 1994.

[Finin 94b] T. Finin, D. McKay, R. Fritzson and R. McEntire, “KQML - A Language and Protocol for Knowledge and Information Exchange.” *Proceedings of the 13th International Distributed Artificial Intelligence Workshop,* July 1994.

[McKay 96] D. McKay, J. Pastor, R. McEntire and T. Finin, An architecture for information agents, in "*Advanced Planning Technology*", (ed. Tate, A.), The AAAI Press, Menlo Park, CA., USA, May 1996, ISBN 0-929280-98-0.

[Mayfield 96] J. Mayfield, Y. Labrou, and T. Finin, Evaluation of KQML as an Agent Communication Language, in Intelligent Agents Volume II -- Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages. M. Wooldridge, J. P. Muller and M. Tambe (eds). Lecture Notes in Artificial Intelligence, Springer-Verlag, 1996.

[Neches 91] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. “Enabling Technology for Knowledge Sharing”. AI Magazine, 12(3):36-56, Fall 1991.

[OLLA 96] OnLine Learning Academy (OLLA) Users Manual. Lockheed Martin. 1996.

[Pastor 94] J. Pastor and D. McKay, "View-Concepts - Persistent Storage for Planning and Scheduling", *Proceedings of the ARPA/Rome Lab 1994 Knowledge-Based Planning and Scheduling Initiative Workshop*, Tucson, February, 1994.

[Pastor 92] J. Pastor, D. McKay and T. Finin, "View-Concepts: Knowledge-Based Access to Databases". *First International Conference on Information and Knowledge Management*, Baltimore, November 1992.

[Patil 92] R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, R. Neches. “The DARPA Knowledge Sharing Effort: Progress Report”. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference* (KR’92), San Mateo, CA, November 1992. Morgan Kaufmann.

[Wiederhold 95] G. Wiederhold "Mediation in Information Systems; in Research Directions in Software Engineering", *ACM Comp.Surveys*, Vol.27 No.2, June 1995, pages 265-267.

[Wiederhold 92] G. Wiederhold, “Mediators in the architecture of Future Information systems”, *IEEE Computer*, March 1992, pages 38-49.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61209

DISTRIBUTION LIST

addresses	number of copies
RAYMOND A. LIUZZI PHD	10
AFRL/IFTB	
525 BROOKS ROAD	
ROME, NY 13441-4505	
LOCKHEED MARTIN	5
590 LANCASTER AVENUE	
P.O. BOX 4001	
FRAZER, PA 19355-1808	
AFRL/IFOIL	1
TECHNICAL LIBRARY	
26 ELECTRONIC PKY	
ROME NY 13441-4514	
ATTENTION: DTIC-OCC	2
DEFENSE TECHNICAL INFO CENTER	
8725 JOHN J. KINGMAN ROAD, STE 0944	
FT. BELVOIR, VA 22060-6218	
ADVANCED RESEARCH PROJECTS AGENCY	1
3701 NORTH FAIRFAX DRIVE	
ARLINGTON VA 22203-1714	
RELIABILITY ANALYSIS CENTER	1
201 MILL ST.	
ROME NY 13440-8200	
ATTN: GWEN NGUYEN	1
GIDEP	
P.O. BOX 8000	
CORONA CA 91718-8000	

AFIT ACADEMIC LIBRARY/LDEE 1
2950 P STREET
AREA B, BLDG 642
WRIGHT-PATTERSON AFB OH 45433-7765

ATTN: GILBERT G. KUPERMAN 1
AL/CFHI, BLDG. 248
2255 H STREET
WRIGHT-PATTERSON AFB OH 45433-7022

ATTN: TECHNICAL DOCUMENTS CENTER 1
OL AL HSC/HRG
2698 G STREET
WRIGHT-PATTERSON AFB OH 45433-7604

AIR UNIVERSITY LIBRARY (AUL/LSAD) 1
600 CHENNAULT CIRCLE
MAXWELL AFB AL 36112-6424

US ARMY SSDC 1
P.O. BOX 1500
ATTN: CSSD-IM-PA
HUNTSVILLE AL 35807-3801

TECHNICAL LIBRARY D0274(PL-TS) 1
SPAWARSYSCEN
53560 HULL STREET
SAN DIEGO CA 92152-5001

NAVAL AIR WARFARE CENTER 1
WEAPONS DIVISION
CODE 4BL000D
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

SPACE & NAVAL WARFARE SYSTEMS CMD 2
ATTN: PMW163-1 (R. SKIANO)RM 1044A
53560 HULL ST.
SAN DIEGO, CA 92152-5002

SPACE & NAVAL WARFARE SYSTEMS COMMAND, EXECUTIVE DIRECTOR (PD13A) ATTN: MR. CARL ANDRIANI 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
COMMANDER, SPACE & NAVAL WARFARE SYSTEMS COMMAND (CODE 32) 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
CDR, US ARMY MISSILE COMMAND REDSTONE SCIENTIFIC INFORMATION CTR ATTN: AMSMI-RD-CS-R, DOCS REDSTONE ARSENAL AL 35898-5241	2
ADVISORY GROUP ON ELECTRON DEVICES SUITE 500 1745 JEFFERSON DAVIS HIGHWAY ARLINGTON VA 22202	1
REPORT COLLECTION, CIC-14 MS P364 LOS ALAMOS NATIONAL LABORATORY LOS ALAMOS NM 87545	1
AEDC LIBRARY TECHNICAL REPORTS FILE 100 KINDEL DRIVE, SUITE C211 ARNOLD AFB TN 37389-3211	1
COMMANDER USAISC ASHC-IMD-L, BLDG 61801 FT HUACHUCA AZ 85613-5000	1
US DEPT OF TRANSPORTATION LIBRARY FB10A, M-457, RM 930 800 INDEPENDENCE AVE, SW WASH DC 22591	1
AWS TECHNICAL LIBRARY 859 BUCHANAN STREET, RM. 427 SCOTT AFB IL 62225-5118	1

AFIWC/MSY 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-7016	1
SOFTWARE ENGINEERING INSTITUTE CARNEGIE MELLON UNIVERSITY 4500 FIFTH AVENUE PITTSBURGH PA 15213	1
NSA/CSS K1 FT MEADE MD 20755-6000	1
ATTN: OM CHAUHAN DCMC WICHITA 271 WEST THIRD STREET NORTH SUITE 6000 WICHITA KS 67202-1212	1
PHILLIPS LABORATORY PL/TL (LIBRARY) 5 WRIGHT STREET HANSCOM AFB MA 01731-3004	1
ATTN: EILEEN LADUKE/D460 MITRE CORPORATION 202 BURLINGTON RD BEDFORD MA 01730	1
OUSD(P)/DTSA/DUTD ATTN: PATRICK G. SULLIVAN, JR. 400 ARMY NAVY DRIVE SUITE 300 ARLINGTON VA 22202	2
SOFTWARE ENGR'G INST TECH LIBRARY ATTN: MR DENNIS SMITH CARNEGIE MELLON UNIVERSITY PITTSBURGH PA 15213-3890	1
USC-ISI ATTN: DR ROBERT M. BALZER 4676 ADMIRALTY WAY MARINA DEL REY CA 90292-6695	1

KESTREL INSTITUTE 1
ATTN: DR CORDELL GREEN
1801 PAGE MILL ROAD
PALO ALTO CA 94304

ROCHESTER INSTITUTE OF TECHNOLOGY 1
ATTN: PROF J. A. LASKY
1 LOMB MEMORIAL DRIVE
P.O. BOX 9887
ROCHESTER NY 14613-5700

WESTINGHOUSE ELECTRONICS CORP 1
ATTN: MR DENNIS BIELAK
ELECTRONICS SYSTEMS GROUP
P.O. BOX 746, MAIL STOP 432
BALTIMORE MD 21203

AFIT/ENG 1
ATTN:TOM HARTRUM
WPAFB OH 45433-6583

THE MITRE CORPORATION 1
ATTN: MR EDWARD H. BENSLEY
BURLINGTON RD/MAIL STOP A350
BEDFORD MA 01730

UNIV OF ILLINOIS, URBANA-CHAMPAIGN 1
ATTN: DR MEHDI HARANDI
DEPT OF COMPUTER SCIENCES
1304 W. SPRINGFIELD/240 DIGITAL LAB
URBANA IL 61801

HONEYWELL, INC. 1
ATTN: MR BERT HARRIS
FEDERAL SYSTEMS
7900 WESTPARK DRIVE
MCLEAN VA 22102

SOFTWARE ENGINEERING INSTITUTE 1
ATTN: MR WILLIAM E. HEFLEY
CARNEGIE-MELLON UNIVERSITY
SEI 2218
PITTSBURGH PA 15213-38990

UNIVERSITY OF SOUTHERN CALIFORNIA 1
ATTN: DR W. LEWIS JOHNSON
INFORMATION SCIENCES INSTITUTE
4676 ADMIRALTY WAY/SUITE 1001
MARINA DEL REY CA 90292-6695

COLUMBIA UNIV/DEPT COMPUTER SCIENCE ATTN: DR GAIL E. KAISER 450 COMPUTER SCIENCE BLDG 500 WEST 120TH STREET NEW YORK NY 10027	1
SOFTWARE PRODUCTIVITY CONSORTIUM ATTN: MR ROBERT LAI 2214 ROCK HILL ROAD HERNDON VA 22070	1
AFIT/ENG ATTN: DR GARY B. LAMONT SCHOOL OF ENGINEERING DEPT ELECTRICAL & COMPUTER ENGRG WPAFB OH 45433-6583	1
NSA/OFC OF RESEARCH ATTN: MS MARY ANNE OVERMAN 9800 SAVAGE ROAD FT GEORGE G. MEADE MD 20755-6000	1
AT&T BELL LABORATORIES ATTN: MR PETER G. SELFRIDGE ROOM 3C-441 600 MOUNTAIN AVE MURRAY HILL NJ 07974	1
ODYSSEY RESEARCH ASSOCIATES, INC. ATTN: MS MAUREEN STILLMAN 301A HARRIS B. DATES DRIVE ITHACA NY 14850-1313	1
TEXAS INSTRUMENTS INCORPORATED ATTN: DR DAVID L. WELLS P.O. BOX 655474, MS 238 DALLAS TX 75265	1
TEXAS A & M UNIVERSITY ATTN: DR PAULA MAYER KNOWLEDGE BASED SYSTEMS LABORATORY DEPT OF INDUSTRIAL ENGINEERING COLLEGE STATION TX 77843	1
KESTREL DEVELOPMENT CORPORATION ATTN: DR RICHARD JULLIG 3260 HILLVIEW AVENUE PALO ALTO CA 94304	1

DARPA/ITO 1
ATTN: DR KIRSTIE BELLMAN
3701 N FAIRFAX DRIVE
ARLINGTON VA 22203-1714

NASA/JOHNSON SPACE CENTER 1
ATTN: CHRIS CULBERT
MAIL CODE PT4
HOUSTON TX 77058

SAIC 1
ATTN: LANCE MILLER
MS T1-6-3
PO BOX 1303 (OR 1710 GOODRIDGE DR)
MCLEAN VA 22102

STERLING IMD INC. 1
KSC OPERATIONS
ATTN: MARK MAGINN
BEECHES TECHNICAL CAMPUS/RT 26 N.
ROME NY 13440

NAVAL POSTGRADUATE SCHOOL 1
ATTN: BALA RAMESH
CODE AS/RS
ADMINISTRATIVE SCIENCES DEPT
MONTEREY CA 93943

HUGHES AIRCRAFT COMPANY 1
ATTN: GERRY BARKSDALE
P. O. BOX 3310
BLDG 618 MS E215
FULLERTON CA 92634

SCHLUMBERGER LABORATORY FOR 1
COMPUTER SCIENCE
ATTN: DR. GUILLERMO ARANGO
8311 NORTH FM620
AUSTIN, TX 78720

MOTOROLA, INC. 1
ATTN: MR. ARNOLD PITTLER
3701 ALGONQUIN ROAD, SUTE 601
ROLLING MEADOWS, IL 60008

DECISION SYSTEMS DEPARTMENT 1
ATTN: PROF WALT SCACCHI
SCHOOL OF BUSINESS
UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90089-1421

SOUTHWEST RESEARCH INSTITUTE ATTN: BRUCE REYNOLDS 6220 CULEBRA ROAD SAN ANTONIO, TX 78228-0510	1
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY ATTN: CHRIS DABROWSKI ROOM A266, BLDG 225 GAITHSBURG MD 20899	1
EXPERT SYSTEMS LABORATORY ATTN: STEVEN H. SCHWARTZ NYNEX SCIENCE & TECHNOLOGY 500 WESTCHESTER AVENUE WHITE PLANS NY 20604	1
NAVAL TRAINING SYSTEMS CENTER ATTN: ROBERT BREAUX/CODE 252 12350 RESEARCH PARKWAY ORLANDO FL 32826-3224	1
CENTER FOR EXCELLENCE IN COMPUTER- AIDED SYSTEMS ENGINEERING ATTN: PERRY ALEXANDER 2291 IRVING HILL ROAD LAWRENCE KS 66049	1
DR JOHN SALASIN DARPA/ITO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
DR BARRY BOEHM DIR, USC CENTER FOR SW ENGINEERING COMPUTER SCIENCE DEPT UNIV OF SOUTHERN CALIFORNIA LOS ANGELES CA 90089-0781	1
DR STEVE CROSS CARNEGIE MELLON UNIVERSITY SCHOOL OF COMPUTER SCIENCE PITTSBURGH PA 15213-3891	1
DR MARK MAYBURY MITRE CORPORATION ADVANCED INFO SYS TECH; G041 BURLINTON ROAD, M/S K-329 BEDFORD MA 01730	1

MR SCOTT FOUSE ISX 4353 PARK TERRACE DRIVE WESTLAKE VILLAGE C 91361	1
MR GARY EDWARDS ISX 433 PARK TERRACE DRIVE WESTLAKE VILLAGE CA 91361	1
DR ED WALKER BBN SYSTEMS & TECH CORPORATION 10 MOULTON STREET CAMBRIDGE MA 02238	1
LEE ERMAN CIMFLEX TEKNOWLEDGE 1810 EMBACADERO ROAD P.O. BOX 10119 PALO ALTO CA 94303	1
DR. DAVE GUNNING DARPA/ISO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
DR. GARY KOOB DARPA/ITO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
DR. ROBERT LUCAS DARPA/ITO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
DR. DAVID GUNNING DARPA/ISO 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
AFIT ACADEMIC LIBRARY/LDEE 2950 P STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1

US ARMY STRATEGIC DEFENSE COMMAND CSSD-IM-PA P.O. BOX 1500 HUNTSVILLE AL 35807-3801	1
NAVAL AIR WARFARE CENTER 6000 E. 21ST STREET INDIANAPOLIS IN 46219-2189	1
COMMANDER, TECHNICAL LIBRARY 474700D/C0223 NAVAIRWARCENWPNDIV 1 ADMINISTRATION CIRCLE CHINA LAKE CA 93555-6001	1
CDR, US ARMY MISSILE COMMAND RSIC, BLDG. 4484 AMSMI-RD-CS-R, DOCS REDSTONE ARSENAL AL 35898-5241	1
REPORT COLLECTION, CIC-14 MS P364 LOS ALAMOS NATIONAL LIBRARY LOS ALAMOS NM 87545	1
AIR WEATHER SERVICE TECHNICAL LIBRARY (FL 4414) 859 BUCHANAN STREET SCOTT AFB IL 62225-5118	1
AFIWC/MSO 102 HALL BLVD, STE 315 SAN ANTONIO TX 78243-6016	1
PHILLIPS LABORATORY PL/TL (LIBRARY) 5 WRIGHT STREET HANSCOM AFB MA 01731-3004	1
AEDC LIBRARY TECHNICAL REPORTS FILE 100 KINDEL DRIVE, SUSITE C211 ARNOLD AFT TN 37389-3211	1

SPACE & NAVAL WARFARE SYSTEMS COMMAND (PMW 178-1) 2451 CRYSTAL DRIVE ARLINGTON VA 22245-5200	1
DR. DOUGLAS DYER DARPA/ISO 3701 NORTH FAIRFAX DRIVE ARLINGTON, VA 22203-1714	1