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.

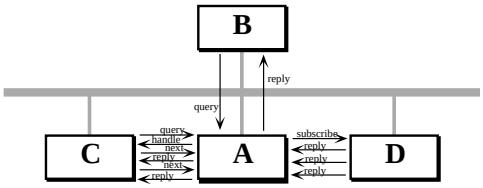

Figure 1: **Several basic communication protocols are supported in KQML.**

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.

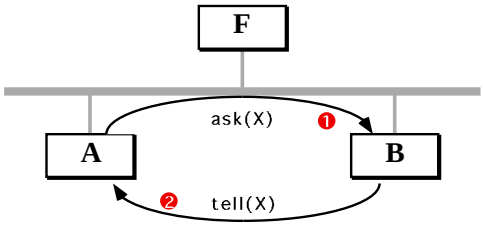

Figure 2: **When A is aware of B and of the appropriateness of querying B about X, a simple point-to-point protocol can be used.**

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

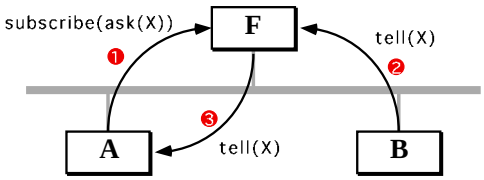

Figure 3: **Agent A can ask facilitator agent F to monitor for changes in its knowledge-base. Facilitators are agents that deal in knowledge about the information services and requirements of other agents and offer such services as forwarding, brokering, recruiting and content-based routing.**

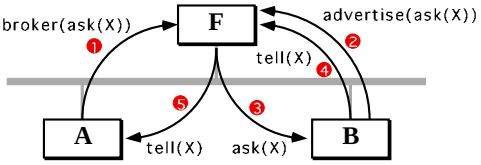

Figure 4: **The broker performative is used to ask a facilitator agent to find another agent which can process a given performative and to receive and forward the reply.**

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

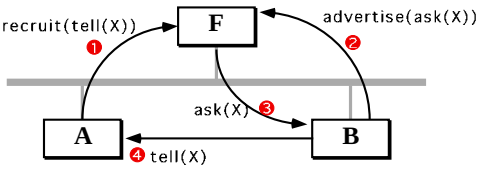

Figure 5: **The recruit performative is used to ask a facilitator agent to find an appropriate agent to which an embedded performative can be forwarded. Any reply is returned directly to the original agent.**

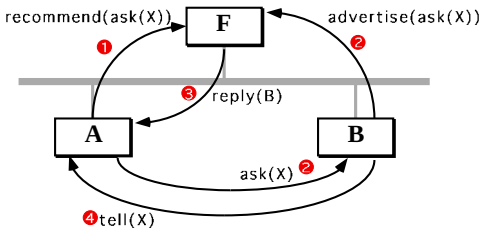

Figure 6: **The recommend performative is used to ask a facilitator agent to respond with the "name" of another agent which is appropriate for sending a particular performative.**

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

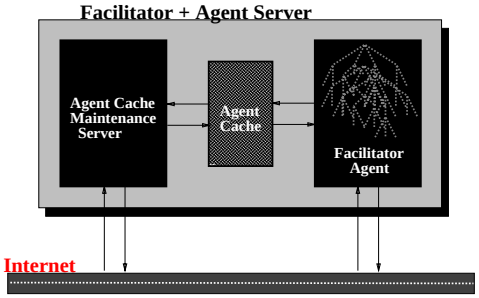

Figure 7: **Some facilitator agents will specialize in knowing how to contact other agents (among other things) and can thus act as "agent-servers".**

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 con-

tent 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

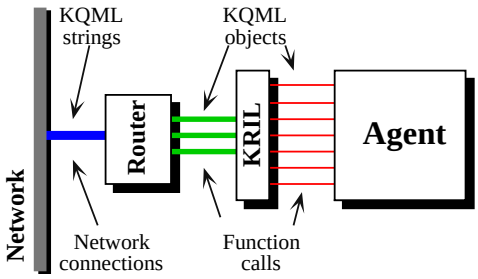

Figure 9: **A router gives an application a single interface to the network, providing both client and server capabilities, managing multiple simultaneous connections, and handling some KQML interactions autonomously. The KRIL is the interface to the agent application and provides internal access points 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. Viewconcepts: 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.