Evaluation of KQML as an Agent Communication Language *

James Mayfield Yannis Labrou Tim Finin

Computer Science and Electrical Engineering Department
University of Maryland Baltimore County
Baltimore MD 21228–5398 USA

{mayfield,jklabrou,finin}@cs.umbc.edu

Abstract. This chapter discusses the desirable features of languages and protocols for communication among intelligent information agents. These desiderata are divided into seven categories: form, content, semantics, implementation, networking, environment, and reliability. The Knowledge Query and Manipulation Language (KQML) is 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 methodologies for building large–scale knowledge bases that are sharable and reusable. KQML is both a message format and a message–handling protocol to support run–time knowledge sharing among agents. KQML is described and evaluated as an agent communication language relative to the desiderata.

To appear 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.

1 Introduction

The computational environment that is emerging in such programs as the National Information Infrastructure (NII) is highly distributed, heterogeneous, extremely dynamic, and comprises a large number of autonomous nodes. An information system operating in such an environment must handle three basic 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 upon with whom they are interacting.
- Several forms of heterogeneity need to be handled, e.g. different platforms, different data formats, the capabilities of different information services, and the different standards (CORBA, OpenDoc, LINDA, ISIS, ZIRCON, OLE, etc.) used by those services.

* 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.

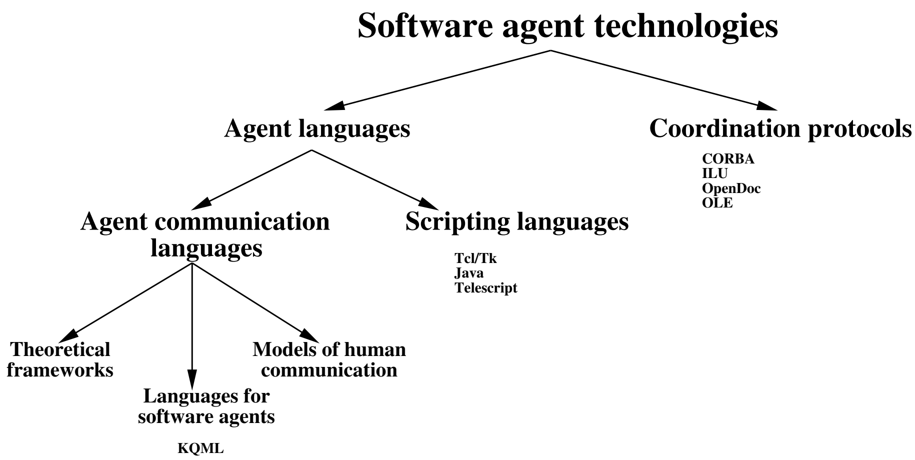

Fig. 1. A taxonomy of agent–related technologies.

- 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 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 software agents can address these problems. When we describe 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. Languages that facilitate high–level communication are thus an essential component of an intelligent software agent architecture. Such languages are the focus of this chapter.

2 What is an Agent Communication Language?

A wide variety of systems, languages, frameworks and standards efforts are associated with software agents; this is due in part to the vagueness of the term 'software agent.' In this section we attempt to tease apart these approaches, and show how KQML relates to each of them. Figure 1 shows a taxonomy of technologies important to software agents. We divide agent technologies into two broad categories: agent languages and coordination protocols.

The agent languages category comprises all languages that can be used to implement software agents. Virtually any programming language can be used for software agent development. One class of languages that has gained much attention lately is the so–called scripting languages, especially those designed for mobile programs. Languages like Tcl/Tk, Java, Telescript, etc., offer the advantage of a level of abstraction that seems particularly attractive for the development of software agents. We place them under the agent languages rubric, because they can be used to program software agents. We distinguish them from agent communication languages though because they are designed primarily to control processes on a single platform. To the extent that these languages contain communication primitives tailored to agent development, they are largely concerned with the transportation of a single agent from one machine to another.

In contrast, agent communication languages are designed specifically to describe and facilitate communication among two or more agents. Three broad sub–categories may be identified under the label of agent communication languages: models of human communication, theoretical frameworks, and communication languages for software agents. Human communication is traditionally modeled in terms of speech act theory [2, 28]. Considerable work has been done (e.g., Cohen and Lesveque [5], Singh [32]) to capture the assumptions and conventions of interaction between human agents and subsequently translate them into workable paradigms for the development of their artificial counterparts. Often, such work leads to theoretical frameworks for artificial agents with human–like capabilities [6, 31]. Such frameworks attempt to account for all aspects of the internal state of an artificial autonomous agent, with a particular attention to how this state changes as the agent interacts (and/or communicates) with the world or with other agents. Sometimes, as in the case of Agent Oriented Programming [29], those frameworks may evolve into implemented software systems. In contrast, agent communication languages (ACL) are concerned strictly with the communication between such computational entities. An ACL (the sub–category that includes KQML) is more than a protocol for the exchange of data, because an attitude about what is exchanged by the agents is also communicated. An ACL may be thought as a communication protocol (or a collection of protocols) that supports many message types.

The other main class of software agent technologies is that of standards and coordination protocols. 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 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.

3 Desiderata

In this section we identify requirements for agent communication languages. We divide these requirements into seven categories: form, content, semantics, implementation, networking, environment, and reliability. We believe that an agent communication language will be valuable to the extent that it meets these requirements. At times, these requirements may be in conflict with one another. For example, a language that can be easily read by people might not be as concise as possible. It is the job of the language designer to balance these various needs.

Form

A good agent communication language should be declarative, syntactically simple, and readable by people. It should be concise, yet easy to parse and to generate. To transmit a statement of the language to another agent, the statement must pass through the bit stream of the underlying transport mechanism. Thus, the language should be linear or should be easily translated into a linear stream of characters. Because a communication language will be integrated into a wide variety of systems, its syntax should be extensible. Finally, the form should be stylistically acceptable to a variety of communities, possibly through the use of "syntactic sugar."

Content

A communication language should be layered in a way that fits well with other systems. In particular, a distinction should be made between the communication language, which expresses communicative acts, and the content language, which expresses facts about the domain. Such layering facilitates the successful integration of the language into applications while providing a conceptual framework for the understanding of the language.

The language should commit to a well–defined set of communicative acts (primitives). Although this set could be extensible, a core set of primitives that captures most of our intuitions about what constitutes a communicative act irrespective of application (database, object–oriented system, knowledge base, etc.) will ensure the usability of the language by a variety of systems. The choice of the core set of primitives also affects the decision of whether to commit to a specific content language. A commitment to a content language allows a more restricted set of communicative acts, because it is then possible to carry more information at the content language level. The disadvantage of commitment to a content language is that all applications must then use the same content language; this is a heavy constraint.

Semantics

Semantics is an issue that has often been neglected during the design of communication languages. Such neglect is the direct result of the obscurity that surrounds the purpose and the desired features of communication languages. Although the semantic description of communication languages and their primitives is often limited to natural language descriptions, a well–defined semantic description is anything but a luxury. This is especially true if the communication language is intended for interaction among a diverse range of applications. Applications designers should have a shared understanding of the language, its primitives and the protocols associated with their use, and abide by that shared understanding.

The semantics of a communication language should exhibit those properties expected of the semantics of any other language. It should be grounded in theory, and it should be unambiguous. It should exhibit canonical form (similarity in meaning should lead to similarity in representation). Because a communication language is intended for interaction that extends over time among spatially dispersed applications, location and time should be carefully addressed by the semantics. Finally, the semantic description should provide a model of communication, which will be useful for performance modeling, among other things.

Implementation

The implementation should be efficient, both in terms of speed and bandwidth utilization. It should provide a good fit with existing software technology. The interface should be easy to use; details of the networking layers that lie below the primitive communicative acts should be hidden from the user. It should be easy to integrate or build application program interfaces for a wide variety of programming languages, including procedural languages (e.g., C and Lisp), scripting languages (e.g., Tcl and Perl), object–oriented languages (e.g., Smalltalk and C++), and logic programming languages (e.g., Prolog). Finally, the language should be amenable to partial implementation, because simple agents may only need to handle a subset of the primitive communicative acts.

Networking

An agent communication language should fit well with modern networking technology. This is particularly important because some of the communication will be about concepts involving networked communications. The language should support all of the basic connection types—point–to–point, multicast and broadcast. Both synchronous and asynchronous connections should be supported. The language should contain a rich enough set of primitives that it can serve as a substrate upon which higher–level languages and protocols can be built. Moreover, these higher–level protocols should be independent of the transport mechanisms (e.g., TCP/IP, email, http, etc.) used.

Environment

The environment in which software agents will be required to work will be distributed, heterogeneous, and dynamic. To provide a communication channel to the outside world in such an environment, a communication language must provide tools for coping with heterogeneity and dynamism. It must support interoperability with other languages and protocols. It must support knowledge discovery in large networks. Finally, it must be easily attachable to legacy systems.

Reliability

A communication language must support reliable and secure communication among agents. Provisions for secure and private exchanges between two agents should be supported. There should be a way to guarantee authentication of agents. Because neither agents nor the underlying transport mechanisms are infallible, a communication language must be robust to inappropriate or malformed messages. The language should support reasonable mechanisms for identifying and signaling errors and warnings.

4 The Knowledge Sharing Effort

The ARPA Knowledge Sharing Effort (KSE) [23, 26] 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 larger systems with greater functionality 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; Knowledge Representation System Specification; Shared, Reusable Knowledge Bases; 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 [14], which is based on first–order logic with extensions to support non–monotonic reasoning and definitions. KIF provides both a specification for the syntax of a language, and a specification for its semantics. KIF can be used to support translation from one content language to another, or as a common content language between two agents that use different native representation languages. Information about KIF and associated tools is available from http://www.cs.umbc.edu/kse/kif/.

The Knowledge Representation System Specification Group (KRSS) focuses 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 Shared, Reusable Knowledge Bases Group (SRKB) is concerned with facilitating consensus on the content 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 interaction between knowledge–based systems and other modules. 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 [25]. The KQML language is one of the main results to come out of the external interfaces group of the KSE. General information is available from http://www.cs.umbc.edu/kqml.

5 The KQML Language

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

Communication takes place on several levels. The content of a message is only a part of the communication. Locating and engaging the attention of another agent with which an agent wishes to communicate is a part of the process. Packaging a message in a way that makes clear the purpose of an agent's communication 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 can be written in either ASCII strings or one of many binary notations (e.g. network–independent XDR representations). KQML implementations ignore the content portion of a message except 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; 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; it has proven to be quite flexible.

KQML is expected to be supported by a software substrate that 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 facilitators ('facilitators' refers to a family of agents that provide services, such as name servers, brokers, authenticators, etc.). These environments are not a part of the KQML specification; they are not standardized, and most of the current KQML environments will evolve to use one or more of the emerging commercial frameworks, such as OMG's CORBA or Microsoft's OLE2.

The KQML language simplifies its implementation by allowing KQML messages to carry arbitrary useful information, 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 that contain descriptions of the content: its language, the ontology it assumes, and more general descriptions, such as a descriptor naming a topic within the ontology. These optional features make it possible for supporting environments to analyze, route and deliver messages based on their content, despite the inaccessibility of that content.

The form of these message parts may vary, depending on the transport mechanism used to carry the KQML messages. In implementations that 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 their difference from other fields. In other transport mechanisms the syntax and content of these messages may differ. For example, in the email 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 functions of the performatives are to identify the protocol to be used to deliver the message, and to supply a speech act that 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 (i.e., 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 transport arguments (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 original message asks for a single reply; this second request message asks for a set of answers as a reply. If we prefer each response to be sent separately instead of as a single large collection, we can use the stream-all performative (to save space, we will no longer repeat fields that are identical to those 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 stream of replies. To exert control over this set of reply messages, the standby performative can be wrapped around the preceding message:

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

The standby performative expects a KQML expression as its content. It requests that the agent receiving the request hold the stream of messages and release them one at a time; the sending agent requests a reply 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:

Category	Name
Basic query	evaluate, ask–if, ask–about, ask–one, ask–all
Multi–response query	stream–about, stream–all, eos
Response	reply, sorry
Generic informational	tell, achieve, cancel, untell, unachieve
Generator	standby, ready, next, rest, discard, generator
Capability–definition	advertise, subscribe, monitor, import, export
Networking	register, unregister, forward, broadcast, route

Table 1. KQML has about two dozen reserved performative names, which fall into seven basic categories.

```
(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 requests that a stream of messages be generated to reflect changes in the trading price of IBM stock). An abbreviation for subscribe/stream combination is known a monitor:

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

Although KQML defines a 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.

Some of the reserved performatives are shown in Table 1. 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 announces what kinds of asynchronous messages an agent is willing to handle) and recruit (which can be used to find suitable agents to respond to 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)))
```

This 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.

6 How KQML Stacks Up

In this section, we evaluate the KQML language as it stands today, relative to our desiderata for agent communication languages.

Form

The only primitives of the language, the performatives, convey the communicative act and the actions to be taken as a result. Thus the form of KQML should be deemed to be declarative. The default format for a KQML message is a linear stream of characters with a Lisp–like syntax. Although this formatting is irrelevant to the function of the language, it makes messages easy to read, easy to parse, and easy to convert to other formats. The inclusion of named parameter–value pairs has several advantages: optional parameters need not be included; KQML messages are easily converted to an object–oriented or frame–based representation; and extensions using additional parameters are easily added. On the negative side, some potential users find Lisp–like syntax to be undesirable.

Content

The KQML language can be viewed as being divided into three layers: the content layer, the message layer and the communication layer. KQML messages are oblivious to the content they carry. Although in current implementations of the language there is no support for non–ASCII content, there is nothing in the language that would prevent such support. The language offers a minimum set of performatives that covers a basic repertoire of communicative acts. They constitute the message layer of the language and are to be interpreted as speech acts. Although there is no "right" necessary and sufficient set of communicative acts, KQML designers tried to find the middle ground between two extremes: 1) providing a small set of primitives thereby requiring overloading at the content level; and 2) providing an extensive set of acts, where inevitably acts will overlap one another and/or embody fine distinctions. The communication layer encodes in the message a set of features that describe lower–level communication parameters (such as the identity of the sender and recipient, and a unique identifier associated with the communication).

Semantics

KQML semantics is still an open issue. For now there are only natural language descriptions of the intended meaning of the performatives and their use (i.e., protocols). An approach that emphasizes the speech act flavor of the communicative acts is a thread of ongoing research [18, 33, 6].

Labrou and Finin [18] have proposed a specific framework for KQML that defines cognitive states for agents and uses them to describe the performative and associated preconditions, postconditions and satisfiability conditions for felicitous use. In addition, conversation policies are provided in the form of dialogue grammars specifying additional constraints for coherent discourse.

Implementation

There are currently a number of KQML software suites that have been implemented and are in use.[2] These implemented systems differ in how they measure up to our implementation desiderata. However, taken as a group, it does appear that implementations are possible that can do well with respect to each of our criteria.

Both the Lockheed KAPI system and the Loral/UMBC KATS suite, for example, provide a content–independent message router and a facilitator. Facilitators are specialized KQML agents that maintain information about other agents in their domain, and about those agents' query–answering capabilities (existing versions of facilitators supply only simple registration services). The application must provide a handler function for each performative that is to be processed by the application.

In general, it is not necessary that an application be able to handle all performatives, since not all KQML–speaking applications will be equally powerful. Creating a KQML–speaking interface to an existing application is a matter of providing the appropriate handler functions.

The efficiency of KQML communication has been investigated. Various compression enhancements have been added that cut communication costs by reducing message size, and by eliminating a substantial fraction of symbol lookup and string duplication [11].

Networking

KQML does address most of the networking desiderata and provides, we believe, a good fit with current networking technology. KQML has been designed to work with multiple transport mechanisms, and implementations have been done that use TCP/IP, SMTP (email), HTTP and CORBA objects to carry messages. KQML agents can be addressed using symbolic names, which are resolved into transport–level addresses by agent name servers. KQML messages can be sent point–to–point; multicasting and broadcasting are possible in any of the transport mechanisms through the use of facilitator class agents.

---

[2] These include KATS (Loral and UMBC), KAPI (Lockheed, EIT and Stanford), Magenta (Stanford), COOL (University of Toronto), and LogicWare (Crystaliz Inc.). Details on these systems can be found at http://www.cs.umbc.edu/kqml/software/.

KQML allows both synchronous/asynchronous interactions and blocking/non–blocking message sending on behalf of an application, through assignment of appropriate values for those parameters in a KQML message. We have found the basic primitives to be sufficient to create agents that offer network–oriented services, such as name servers, proxy agents and brokers. Although some work has been done to build higher–level protocols on top of KQML [3, 4, 17], this remains as a rich area to be explored.

Environment

KQML can use any transport protocol as its transport mechanism (HTTP, SMTP, TCP/IP etc.). Since KQML messages are oblivious to content, there are no restrictions on the content language (beyond the obvious requirement that a handler can be written to process the content of each type of performative). Interoperability with other communication languages remains to be addressed as such languages appear. One such attempt has been made by Davis, whose Agent–K [8] attempts to join KQML with Shoham's Agent Oriented Programming [29]. The existence of facilitators in the KQML environment can provide the means for knowledge discovery in large networks, especially if facilitators can cooperate with other knowledge discovery applications available in the World Wide Web.

Reliability

Since KQML speaking agents might be imperfect, there are performatives (error and sorry) that can be used to respond to messages that an application cannot process or comprehend; this provides a crude way for an agent to respond to ill–formed, inappropriate or unwanted incoming messages. A more general approach being considered is the development of an ontology of warnings, errors and infelicities that would be appropriate for agents and agent communications languages. Terms in this ontology could be used in the :content field of the rudimentary error and sorry performatives to describe the problem encountered.

The issues of security and authentication are only beginning to be addressed by the KQML community. A security architecture model [34] based on data encryption techniques has been proposed for KQML. In tune with KQML's asynchronous nature, 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 sender authentication, message integrity and data privacy. The enhanced security model additionally 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 guard against cipher attacks.

7 Conclusion

A good agent communication language has many needs, some of which are in competition. KQML is a new communication language that addresses many (although not all) of these needs. The development of KQML has been marked by an effort to balance the theoretical requirements for a sound and complete framework against the technical demands for efficient, easy–to–use implementations.

The inevitable compromise between the two may not result in a communication language that suits everyone, but we believe that KQML will prove useful in a wide range of intelligent software agent architectures. Additional information about KQML, including papers, language specifications, access to APIs, information on email discussion lists, etc., can be found at http://www.cs.umbc.edu/kqml/.

References

1. ARPA Knowledge Sharing Initiative. Specification of the KQML agent–communication language. ARPA Knowledge Sharing Initiative, External Interfaces Working Group working paper. Available as http://www.cs.umbc.edu/kqml/papers/kqml-spec.ps, December 1992.
2. J. L. Austin. How To Do Things With Words. Harvard University Press, second edition, 1962, 1975.
3. M. Barbuceanu and M. S. Fox. COOL: a language for describing coordination in multi-agent systems. In Proceedings of the First International Conference on Multi-agent systems, pages 17–24. AAAI/MIT Press, 1995.
4. M. Barbuceanu and M. S. Fox. The architecture of an agent building shell. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, Intelligent Agents Volume II— Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL–95), Lecture Notes in Artificial Intelligence. Springer–Verlag, 1996. (In this volume).
5. Philip R. Cohen and H.J. Levesque. Intention is choice with commitment. Artificial Intelligence, 42(2–3):213–361, 1990.
6. Philip R. Cohen and H.J. Levesque. Communicative actions for artificial agents. In Proceedings of the International Conference on Multi–Agent Systems. AAAI Press, June 1995.
7. Sun Computers. The JAVA language: a white paper. 1994.
8. Winton Davies. Agent–K: An integration of AOP and KQML. Available as http://www.-csd.abdn.ac.uk/w̃davies/Publications/CIKM94/agentk.html, 1994.
9. Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeffery M. Bradshaw, editor, Software Agents. MIT Press, 1995.
10. 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 (Eds.), Knowledge Building and Knowledge Sharing, Ohmsha and IOS Press, 1994. Available as http://www.cs.umbc.edu/kqml/papers/kbks.ps.
11. 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.
12. 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).
13. Michael Genesereth. An agent–based approach to software interoperability. Technical Report Logic–91–6, Logic Group, CSD, Stanford University, February 1993.

14. Michael Genesereth and Richard Fikes. Knowledge Interchange Format, version 3.0 reference manual. Technical report, Computer Science Department, Stanford University, June 1992.
15. Robert S. Gray. agent Tcl: a transportable agent system. In Proceedings of the ACM CIKM Intelligent Information Agents Workshop, December 1995.
16. Daniel R. Kuokka, James G. McGuire, Jay C. Weber, Jay M. Tenenbaum, Thomas R. Gruber, and Gregory R. Olsen. SHADE: Technology for knowledge–based collaborative engineering. In Proceedings of the AAAI Workshop on AI in Collaborative Design, 1993.
17. K. Kuwabara. AgenTalk: coordination protocol description for multi-agent systems. In Proceedings of the First International Conference on Multi-agent systems. AAAI/MIT Press, 1995.
18. 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.
19. James Mayfield, Yannis Labrou, and Tim Finin. Desiderata for agent communication languages. In Proceedings of the 1995 AAAI Spring Symposium on Information Gathering in Distributed Environments, March 1995.
20. John McCarthy. elephant 2000: a programming language based on speech acts.
21. James G. McGuire, Daniel R. Kuokka, Jay C. Weber, Jay M. Tenenbaum, Thomas R. Gruber, and Gregory R. Olsen. SHADE: Technology for knowledge–based collaborative engineering. Journal of Concurrent Engineering: Applications and Research (CERA), 1(2), September 1993.
22. M.Tenenbaum, J. Weber, and T. Gruber. Enterprise integration: Lessons from SHADE and PACT. In C. Petrie, editor, Enterprise Integration Modeling. MIT Press, 1993.
23. 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.
24. 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).
25. Jon Pastor, Don McKay, and Tim Finin. View–concepts: Knowledge–based access to databases. In Proceedings of the First International Conference on Information and Knowledge Management, October 1992.
26. 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.
27. TIm Ritchey. Java! New Riders Publishing, 1995.
28. John R. Searle. Speech Acts: An Essay in the Philosophy of Language. Cambridge University Press, 1969.
29. Yoav Shoham. Agent–oriented programming. Artificial Intelligence, 60:51–92, 1993.
30. Candy L. Sidner. An artificial discourse language for collaborative negotiation. In Proceedings of the 1994 National Conference on Artificial Intelligence (AAAI–94), August 1994.
31. M. P. Singh. Semantical considerations on some primitives for agent specification. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, Intelligent Agents Volume II—Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL–95), Lecture Notes in Artificial Intelligence. Springer–Verlag, 1996. (In this volume).
32. M.P. Singh. Towards a formal theory of communication for multiagent systems. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '91), 1991.

33. Ira A. Smith and Philip R. Cohen. Toward a semantics for a speech act based agent communications language. In Proceedings of the ACM CIKM Intelligent Information Agents Workshop, December 1995.
34. Chelliah Thirunavukkarasu, Tim Finin, and James Mayfield. Secret agents—a security architecture for the KQML agent communication language. In Proceedings of the ACM CIKM Intelligent Information Agents Workshop, December 1995.
35. James White. Mobile agents. In Jeffery M. Bradshaw, editor, Software Agents. MIT Press, 1995.
36. Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. Communications of the ACM, 33(11):89–99, November 1992.
37. Darrell Woelk. Developing InfoSleuth agents using Rosette: an agent based language. In Proceedings of the ACM CIKM Intelligent Information Agents Workshop, December 1995.