KQML - Issues and Review
Revised Report
for
Paramax Subcontract No. V20275
Interface Standards for Knowledge Representation Systems

Stuart C. Shapiro and Hans Chalupsky
Department of Computer Science
and Center for Cognitive Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, New York 14260

May 15, 1992

1 Introduction

The main objective of the Knowledge Sharing Initiative [Neches *et al.*, 1991; Genesereth and Fikes, 1991] is to develop languages and protocols that allow knowledge based systems (or agents, applications) interconnected by a computer network to communicate with each other and to exchange knowledge. A typical type of interaction is that of a request for information, i.e., one agent asking another agent whom it assumes to be knowledgable some questions about some subject. The agent who wants to know something (the requester) has to solve the following major problems:

1. Finding out about knowledgable agents that are potential candidates for answering questions about some subject
2. Establishing a reliable communication channel to one of these agents (a potential provider of information)
3. Managing a properly structured discourse with that agent
4. Translating discourse items between its own knowledge representation language (or language of thought) and that of the provider.

As a solution to the first three problems [Finin et al., 1992] propose a model in which agents communicate with each other via interconnected *facilitators*. The fourth problem will be handled by the development of an Interlingua [Genesereth and Fikes, 1991], shared ontologies, etc.

A facilitator acts very much like a secretary. It communicates with its agent (application, "boss") in some private language, and with all the other facilitators in the facilitator language. Here is an example conversation that should illustrate some of the points:

1. Boss1 to F1: Please connect me to somebody who knows recursive function theory.
2. F1 asks all its facilitator friends: Is anybody out there whose boss knows "recursive function theory"?
3. FN to F1: Yes, my boss knows "recursive function theory".
4. F1 to FN: Could you connect me to your boss?
5. FN to F1: All set.
6. F1 to FN: Can my boss ask his questions in English?
7. FN (after making sure that it has a bidirectional German/English translator because its boss is german) to F1: Yes, sure.
8. F1 to FN: Ok, let me talk to my boss now.
9. F1 to Boss1: I got somebody on the line. What would you like to know?
10. Boss1 to F1: Is the Ackermann function primitive recursive?
11. F1 to FN: Question: "Is the Ackermann function primitive recursive"
12. FN to BossN: Ist die Ackermannfunktion primitiv rekursiv?
13. BossN to FN: Nein.
14. FN to F1: Answer: "No".
15. F1 to Boss1: No, it is not.

16. Boss1 to F1: Is it computable?

17. F1 to FN: Question: "Is it computable"

18. FN to BossN: Ist sie berechenbar?

19. BossN to FN: Ja.

20. FN to F1: Answer: "Yes".

21. F1 to Boss1: Yes, it is.

22. ...more recursive function talk...

There are two main parts to this conversation: (1) Establishing a communication channel to a knowledgable agent, and (2) carrying out a conversation across this channel. Note that the facilitators do not have to know anything about recursive function theory or understand sentences discussing it (as is probably true also for most secretaries). All they have to know is how to find an expert in this field, how to carry out a conversation with questions and answers etc., and how to convert the content of a speech act into a different language.

Another important observation to make is that once the communication channel between Boss1 and BossN via their facilitators is established, the discourse has a state. Subsequent questions use the same language. There are follow-up questions that make previous questions more specific etc.

Finally, the task that facilitator F1 has to perform in order to find out about agents knowledgable in recursive function theory is very similar to the request that Boss1 wants to get handled, just with a different domain. Boss1 wants to get answers to issues in recursive function theory, to achieve that, it asks F1 in its private language to find an expert. F1 wants to get answers to who are possible candidates that know about "recursive function theory". To achieve that, it asks the facilitators it directly knows about in their own (facilitator) language.

The main task of this project is to define a language that is rich enough to handle all aspects of conversations such as the one shown above. This language will be called KQML, the Knowledge Query and Manipulation Language, which can be viewed to consist of three layers: A communication layer that deals with proper routing of information, a message layer that deals with speech acts, and a content layer that deals with actual content sentences (cf. [Finin et al., 1992]). In this document we will be mainly concerned with the second or *message layer*.

2 Discourse Establishment

Discourse establishment in itself is a two step process: 1) Finding out who to talk to, and 2) actually starting a discourse with some other agent. Step 1 of this process does not have to be completely defined and specified by KQML. This step is rather handled by these special purpose agents called facilitators. However, KQML will define constructs that allow facilitators to announce and access information to aid this process. For example, [Finin et al., 1992] describes special *declaration* messages with which applications can announce what type of questions they want to import or export. There are various difficult problems associated with this part of KQML, for example, how facilitators would be able to make use of such declarations made by other agents if they do not understand the content language of these expressions and also do not speak the Interlingua.

Step 2 of the discourse establishment process is lower level and more or less just a networking problem. However, this part of the protocol has to be specified by KQML completely, because otherwise agents would not know how to actually start talking to each other over the Internet once they know the address of their communication partner.

In this document we will not be concerned with the discourse establishment process, we rather will assume that a communication channel between two agents has already been established.

3 KQML Performatives

This section discusses the message layer of KQML, i.e., the part of KQML that deals with performatives or speech acts needed in actual one to one discourse. The underlying assumption is that there exists some "almost universal" or generic set of performatives with which conversations with (almost) any knowledge based system (or agent) can be carried out. All performatives described in this section assume that a communication channel between two agents via their facilitators has already been established.

3.1 One to One Discourse

As motivated above one to one discourse is modeled as having a state. This state will reside partly in the facilitators, and partly in the agents/applications they are connected to. Once a discourse channel (or bidirectional communication stream) between two facilitators has been established they can talk to each other by uttering (writing) messages into the channel, and perceiving (reading) messages from it.

The facilitator at one end of the communication channel will be called the *requester* (or client), the facilitator at the other end is the *provider* (or server). Messages sent (uttered) by the requester are called requests, messages sent by the provider are called replies. It is assumed that the provider receives requests in the same order as they are sent by the requester. The requester is in complete control of the discourse, i.e., the provider cannot take initiative and issue requests to the requester.

KQML messages are specified as lists written in Common-Lisp syntax. Messages sent across a communication channel are messages of the communication layer (as described in [Finin et al., 1992]) and are called *packages*. A package is specified as a list whose first element is the atom **PACKAGE** and whose remaining elements are alternating attribute-value pairs using the Common-Lisp keyword argument format. Possible keyword arguments are:

(PACKAGE

:FROM – *<Agent ID>*
The unique identifier of the sending agent (e.g., its Internet address).

:TO – *<Agent ID>*
The unique identifier of the receiving agent.

:ID – *<Package ID>*
A unique identifier for this message. This should be generated at this layer by the facilitator and is used to refer to the message later. A package ID could be generated by concatenating the value of **:FROM** and a time stamp.

:COMM – *<Oneof(*sync,async*)>*
Specifies whether or not the communication is to be carried out in a *synchronous* or *asynchronous* mode.

:CONTENT – a (DCL ...) or (MSG ...) expression.

)

Here is an example package:

```
(PACKAGE
 :FROM "ap001.cs.buffalo.edu"
:TO "ap002.prc.unisys.com"
:ID "ap001.cs.buffalo.edu 10:15:26.023"
:COMM sync
:CONTENT
   (MSG
    :TYPE query
    :CONTENT-LANGUANGE interlingua
    :CONTENT "(color ?X ?Y)"))
```

The following paragraphs will describe the various forms of expressions that are legal as values of the `:CONTENT` slot of a package. These expressions are part of the message layer.

3.2 Content Expressions

Content expressions have a similar syntax to packages. They are lists whose first atom indicates the class of the expression. The two classes are *messages* identified by `MSG` as the first atom of the list, and *declarations* with `DCL` as the first atom. The body of a content expression is a list of attribute-value pairs using the Common-Lisp keyword format. The keywords and their values determine the semantics of a content expression.

3.2.1 Content Language

The definition of the various KQML performatives described below is based on the following model of a knowledge base: A *knowledge base* (KB) is a set of sentences in a language L. L can be the object language of the knowledge base, or a set of sentences of another language for which a computable mapping into L exists. Candidates for languages other than the object language of a KB are, for example, the Interlingua, or, if the object language for a KB are graphs, a linear notation describing these graphs.

Since KQML is not assumed to be a superset of the Interlingua, it has to identify sentences of the KB by way of quoted sentences of a language that can be translated into the object language of the KB. This language is called the *content language* (CL).

All sentence arguments of the KQML performatives described below are CL sentences which are assumed to be (finite) Common-Lisp strings. These sentences are not part of KQML, however, KQML must be able to talk about content languages.

The languages for the contents of requests and replies can be declared with **`declare-content-languages`**:

(`MSG`

`:TYPE` – **`declare-content-languages`**

`:REQUEST-CONTENT-LANGUAGE` – *<content-language>*
Declares what the content language for requests will be. From then on all content language sentences of requests received by the provider should be assumed to be in that language. The language for requests can be `interlingua` (the default), `local` to use the object language of the local KB of the provider, or a string that specifies some other content language known by the provider. This request can only be handled successfully if the provider knows how to translate sentences of the request CL into sentences of the object language of its local KB.

`:REPLY-CONTENT-LANGUAGE` – *<content-language>*
Request what the content language for replies should be. Default is the content language used for requests. This request can only be handled successfully if the provider knows how to translate sentences of the object language of its local KB into sentences of the reply CL.

)

3.3 Discourse Contexts

Requests and replies should be made relative to a current discourse context. A discourse context is a subset of the sentences that define the local KB of the provider. There is a design decision whether the requester or the provider should do most of the work in context construction. A rich context specification language would put most of the burden on the provider, a simple one puts most of it on the requester (i.e., contexts have to be established by compiling a request context specification into a sequence of simple context manipulation commands that can be handled by the provider). Many systems might not be able to provide partitioned views of the KB at all, in which case the discourse context will always be the whole KB.

The following messages allow a requester to establish a discourse context that contains some subset of the sentences of the local KB of the provider. With **set-discourse-context** it can explicitly set the current discourse context to such a subset of sentences of the local KB:

(MSG

:TYPE – **set-discourse-context**

:REQUEST-CONTENT-LANGUAGE – *<content-language>*
Content language to be used for this particular message. Default is the language set by a **declare--content-languages** message, or `interlingua`.

:CONTENT – *<sentence-pattern>*
Request the current discourse context to be set to the set of all sentences which match the supplied sentence pattern in the local KB of the provider. We will assume that every knowledge representation language will have a notion of a pattern or an open sentence and a matching or unification operation associated with it. If the value of pattern is **empty** the current discourse context will be set to the empty set, if its value is `all` the whole KB of the provider will be used (the default).

)

add-to-discourse-context allows a requester to add additional sentences to the current discourse context:

(MSG

:TYPE – **add-to-discourse-context**

:REQUEST-CONTENT-LANGUAGE – *<content-language>* (see above)

:CONTENT – *<sentence-pattern>*
Request that all sentences in the local KB of the provider which match the supplied sentence pattern are added to the current discourse context. The default value for this slot is `all`.

)

assert allows a requester to add a new sentence which is not already a member of the local KB to the current discourse context:

(MSG

:TYPE – **assert**

:REQUEST-CONTENT-LANGUAGE – *<content-language>* (see above)

:CONTENT – *<sentence>*
Request the supplied sentence to be added as an assertion to the current discourse context.

)

With **remove-from-discourse-context** a requester can remove a set of sentences from the current discourse context:

(MSG

:TYPE – **remove-from-discourse-context**

:**REQUEST-CONTENT-LANGUAGE** – <*content-language*> (see above)

:**CONTENT** – <*sentence-pattern*>
Request that all sentences in the current discourse context which match the supplied sentence pattern are removed from the current discourse context. The default value for this slot is **empty**.

)

assign-truth-value allows the requester to change truth values associated with sentences in the current discourse context:

(**MSG**

:TYPE – **assign-truth-value**

:**REQUEST-CONTENT-LANGUAGE** – <*content-language*> (see above)

:**TRUTH-VALUE** – <*phrase*>
A CL phrase that describes a valid truth value which should get assigned to the sentences identified by the :**CONTENT** slot (what a valid truth value is is defined by the local KB of the provider). Some KBs might not deal with truth values explicitly, but rather implicitly by assuming a sentence to be true if it is an element of the KB (or the current discourse context). A truth value does not necessarily have to be one of true or false, it could be a belief status, an assertion flag, or a numerical value representing some kind of certainty.

:**CONTENT** – <*sentence-pattern*>
Request that all sentences in the current discourse context that match the supplied pattern get assigned the value of the :**TRUTH-VALUE** slot.

)

3.4 Definitions

At the moment we will treat definitions as special cases of assertions which assert sentences that express definitions. However, this approach might be too simplistic and special performatives for definition and un-definition might be necessary.

3.5 Question Answering

Once we have established a discourse context we want to ask questions. One type of question asks about the truth value of sentences. If the question is a closed sentence of the CL we want to know whether it has a certain truth value. If the question is an open sentence we are interested in a number of instances of the question that have a certain truth value. Some questions might be easy to answer, others might be very difficult or impossible to answer. To tell the provider how much work it should invest to find an answer we introduce the concept of a *work level* which is basically a specification of how much resources should be spent at the most to answer a question. Depending on the supplied work level the provider might choose a particular inference strategy suited for that level.

Unless otherwise indicated, for all the following messages it is assumed that derived answers will be automatically added to the current discourse context.

query-sentence-status handles queries about the truth value (or belief status) of sentences:

(**MSG**

:TYPE – **query-sentence-status**

:REQUEST-CONTENT-LANGUAGE – *<content-language>* (see above)

:REPLY-CONTENT-LANGUAGE – *<content-language>* (see above)

:WORKLEVEL – *<worklevel type>*
Answers to the query in the **:CONTENT** slot will be found by performing some kind of inference. The total amount of inference (or work) to be invested to find all requested answers is controlled by the value of **:WORKLEVEL**. Its value can be **minimal** (the default) to request quick but probably incomplete answers, **maximal** to request the provider to derive answers without any (or maximal) resource bounds, or a number that specifies a maximal number of work units to be invested by the provider. What a work unit is is defined by the local KB of the provider.

:HOW-MANY – *<natural number>*
The provider should report new answers at the earliest possible point after it has derived at least **:HOW-MANY** new answers since the last report (default is 1). If the number of allotted resources got exhausted all answers derived so far will be reported. This kind of control is important if there is more than one answer, e.g., if the query is an open sentence or a pattern.

:REPORT-MODE – *<Oneof(***suspend,continuous***)>*
Controls what the provider should do after it has reported a number of answers as specified by **:HOW-MANY**. If the value is **suspend** the provider will suspend its answering activity until it receives a continuation message. If the value is **continuous** (the default) the provider will continuously try to find new answers until either no more answers can be found, the allotted resources are exhausted, or it receives a control message that tells it to stop. New answers will be reported whenever at least **:HOW-MANY** new answers are available.

:TRUTH-VALUES – *<(phrase, phrase)>*
A pair of CL phrases that describe valid truth values. All derived answers are required to have a truth value that is within the range of truth values defined by the two supplied values. If the local KB of the provider does not have a notion of an ordering of truth values then the range is just the set of the two values. If the two values are identical this set will be a singleton set. The default is a special value **any** which indicates that answers of any truth value are acceptable.

:CONTENT – *<sentence-pattern>*
This slot contains the actual query whose truth value should be found. If the query is an open sentence or a pattern then all instances of it that have the specified truth value are potential answers.

)

Another type of question is topical in nature, i.e., it requests information related to a certain topic, for example, as in the question "tell me (everything you know) about dogs". Here we are not interested in the truth value of particular sentences, rather we want sentences that are related to the topic expressed by the question. Depending on the different levels of expertise of the requester and the provider there might be answers that the requester will not be able to understand.

(MSG

:TYPE – **query-about-topic**

:REQUEST-CONTENT-LANGUAGE – *<content-language>* (see above)

:REPLY-CONTENT-LANGUAGE – *<content-language>* (see above)

:WORKLEVEL – *<worklevel type>* (see above)

:HOW-MANY – *<natural number>* (see above)

:REPORT-MODE – *<Oneof(***suspend,continuous***)>* (see above)

:**TRUTH-VALUES** – <*(phrase, phrase)*> (see above)

:**CONTENT** – <*phrase or pattern*>
A CL phrase denoting some entity about which relevant information should be found. What is considered as relevant is defined by the local KB of the provider. If the supplied value is a pattern then it is to be viewed as something like a predicate that describes a class of entities about which answers should be found.

)

The next message handles queries of the kind "what can you infer from X", that is some kind of forward inference. There are two variations to this kind of query: One where the answerer actually assumes the initial assertions as its own, and another one in which these assertions are only hypothetically assumed to answer the question:

(**MSG**

:TYPE – **assert-and-infer**

:**REQUEST-CONTENT-LANGUAGE** – <*content-language*> (see above)

:**REPLY-CONTENT-LANGUAGE** – <*content-language*> (see above)

:**WORKLEVEL** – <*worklevel type*> (see above)

:**HOW-MANY** – <*natural number*> (see above)

:**REPORT-MODE** – <*Oneof(*`suspend,continuous`*)*> (see above)

:**TRUTH-VALUES** – <*(phrase, phrase)*> (see above)

:**ASSERTION-MODE** – <*Oneof(*`actual,hypothetical`*)*>
If the value of this slot is `actual` (the default) then the sentences will be added as normal assertions to the current discourse context of the provider. If the value is `hypothetical` the sentences will be hypothetically assumed in the current discourse context (added to it) until all the answers are reported. Then all these assumptions and the answers depending on them will be removed again. Note, that even if one of the content sentences was already a member of the current discourse context before this request was issued, it will be removed with all the other hypothetical assumptions of this request.

:**CONTENT** – <*sentence list*>
Answers should be found by starting inference from the sentences supplied in this slot.

)

3.6 Control Messages

`control` messages allow the requester to exert some control on ongoing work performed by the provider.

(**MSG**

:TYPE – `control`

:**CONTROL-TYPE** – <*Oneof(*`suspend,continue,stop`*)*>
If the value of this slot is `suspend` then at the next possible point the provider should suspend working on the request identified by :**REQUEST-ID** and allow for a continuation if requested. If the request has already been completed this is a noop. If the value is `continue` then the provider should finish whatever it is doing right now and then continue to work on the request identified by :**REQUEST-ID**. If that request has already been finished this is a noop. It cannot be assumed that a

previously suspended task will find the exact same state that existed when the interrupt occurred. If the value is `stop` then at the next possible point the provider should terminate to work on the request identified by `:REQUEST-ID`. If that request has already been finished this is a noop.

`:REQUEST-ID` – *<Package ID>*
Holds the ID of the package that contained the request to which this message refers. Defaults to the ID of the most recently sent request.

)

3.7 Replies

Similar to requests we need a set of performatives for replies. Every reply has to refer to a particular request by the identifier of the package that contained the request. There are basically two kinds of answers:

Success/Failure replies tell the requester whether a certain request could get handled successfully or not. Some requests only expect that kind of reply, e.g., the messages that deal with setting up a discourse context.

Content replies contain a set of sentences that are answers to queries. An empty set indicates that no answers could be found. Yes/no type queries will get a singleton set as a reply if the answer was yes, an empty set otherwise. The possibility to supply a range of acceptable truth values to a query makes it necessary to indicate the truth value of a particular answer. Instead of associating sets of answers with sets of truth values we will assume that the truth value of an answer is expressed as part of the content language sentence.

Success/Failure replies can be sent with the following message:

(MSG

`:TYPE` – `success-reply`

`:VALUE` – *<Oneof(*`success,failure`*)>*

`:REQUEST-ID` – *<Package ID>*
Holds the ID of the package that contained the request to which this message refers.

`:EXPLANATION` – *<string>*
If the value of `:VALUE` was `failure` this slot can be used to hold an english sentence that explains to any humans involved why a certain request could not be handled.

)

`content-reply` messages transfer actual answers to queries back to the requester.

(MSG

`:TYPE` – `content-reply`

`:REQUEST-ID` – *<Package ID>* (see above)

`:REPLY-NUMBER` – *<natural number>*
The different report modes for queries allow for multiple replies to a particular query. The value of this slot indicates the number of this particular reply. Default is 1.

`:CONTENT` – *<sentence set>*
Contains a set of sentences that constitute replies to the query identified by `:REQUEST-ID`.

)

3.8 Literal Discourse

There will always be some kinds of speech acts that cannot be handled by any of the performatives above. As a way out of this dilemma we define a performative that allows the requester to issue a command which will be literally interpreted and executed by the provider. Of course, this performative can only be used by a requester who knows a lot about the internal structure of the provider.

(MSG

  :TYPE – **interpret-literally**

  :CONTENT – *<command>*
    Literally interpret and execute the command held in :CONTENT.

)

3.9 To Do

So far the various performatives do not account for ontologies. Future versions will have to.

4 Discourse Protocol

The definitions of the various performatives above only vaguely hint at a discourse protocol used by requester and provider to communicate successfully. In a prototype implementation of a KQML interface for SNePS-2.1 applications [Shapiro and Chalupsky, 1992] we adopted the following simple protocol for the exchange of KQML messages between a KQML server and a client application:

Client Application	*KQML Server*
	Wait (read) for next request
Send (write) a request Wait for next reply	
	Interpret request Send 0 or more content replies without waiting inbetween
Read and interpret 0 or more content replies	
	Send a success reply
Read and interpret success reply and regain discourse control	
	Wait for next request
	.....

After a connection between a client and a server has been established, the KQML server stays in a read-request/interpret-request/send-replies loop until the connection gets closed by the remote process, or some unrecoverable error occurs. Because a certain request can trigger multiple content reply messages, there must be a way to tell the client when it cannot expect any more replies. For this purpose the following policy has been adopted:

> Every KQML package handled by the KQML server has to be acknowledged by sending exactly one **success-reply** message back to the client once its handling has been completed, regardless of whether the request did trigger any content replies or not.

Using this policy, after issuing a request the client can simply read and interpret successive incoming packages until it finally receives a success reply package which will result in the regaining of discourse control. For asynchronous, non-blocking communication, more elaborate protocols will have to be developed.

References

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

[Genesereth and Fikes, 1991] Michael R. Genesereth and Richard Fikes. *Knowledge Interchange Format*. Computer Science Department, Stanford University, Stanford, CA 94305, version 2.2 edition, March 1991.

[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.

[Shapiro and Chalupsky, 1992] Stuart C. Shapiro and Hans Chalupsky. KQML prototype interface. Final report, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY, May 1992.