DRAFT
Specification of the KQML Agent-Communication Language

plus example agent policies and architectures

The DARPA Knowledge Sharing Initiative
External Interfaces Working Group

Tim Finin (co-chair)
University of Maryland

Jay Weber (co-chair)
Enterprise Integration Technologies

Gio Wiederhold (former co-chair)
Stanford University

Michael Genesereth
Stanford University

Richard Fritzson
Donald McKay
Paramax Systems

James McGuire
Richard Pelavin
Lockheed AI Center

Stuart Shapiro
SUNY Buffalo

Chris Beck
University of Toronto

June 15, 1993

Abstract

This document is a **draft** of an initial specification for the KQML agent communication language being developed by the external interfaces working group of the DARPA Knowledge Sharing Effort. KQML is intendewd to be a high-level language to be used by knowledge-based system to share knowledge at run time.

Notice of DRAFT status. This document presents the current draft of a specification under consideration by the DARPA Knowledge Sharing Effort. It is provided for information purposes, and should be treated as representing only the current status of discussions. It should not be interpreted as a finished product. This document should not be quoted or cited as representing the official position of DARPA, the Knowledge Sharing Effort, or any other organization. The specifications herein are subject to change. To be placed on the distribution list for future releases of these documents, contact the authors or send electronic mail to Neches@ISI.edu.

General comments. Send general comments on this document by email to the mailing list KQML-USERS@ISI.EDU. The current working group co-chairs can be reached as follows: Tim Finin, Computer Science, University of Maryland Baltimore County, Baltimore MD 21228. phone:410-455-3522, email: finin@cs.umbc.edu. Jaw Weber, Enterprise Integration Technologies Corporation, 459 Hamilton Avenue, Suite 100, Palo Alto, CA 94301 (415)617-8002. weber@eitech.com.

Contents

1 Introduction **4**

1.1 KQML Transport Assumptions 6

2 KQML String Syntax **7**

2.1 KQML string syntax in BNF 7

3 KQML Semantics **8**

4 Reserved Performative Parameters **9**

5 Reserved Performative Names **11**

5.1 Basic informative performatives 12

5.2 Database performatives 13

5.3 Basic responses 15

5.4 Basic query performatives 15

5.5 Multi-response query performatives 17

5.6 Basic effector performatives 18

5.7 Generator performatives 19

5.8 Capability-definition performatives 21

5.9 Notification performatives 22

5.10 Networking performatives 22

5.11 Facilitation performatives 25

6 Proposed Performatives **27**

A Example Agent Policies **28**

B Example Agent Architectures and Implementations **29**

B.1 Content-based routing architecture (ala DRPI) 29

B.2 Agent-Based System Engineering (ABSE) 29

B.3 Palo Alto Collaborative Testbed 30

B.4 Information bus architecture (ala TIB) 32

C KQML APIs and Alternate Syntaxes **33**

C.1 The ABSE Lisp API 33

C.2 The DRPI TCP/IP API 33

C.3 KQeMaiL 33

C.4 CORBA dynamic invocation interface 33

D Future Work **34**

Preface

The External Interfaces Working Group is a collection of artificial intelligence and distributed systems researchers interested in software systems of communicating agents.

The group was formed in 1990 as a part of the DARPA Knowledge Sharing Effort, with the charter to develop protocols for the exchange of represented knowledge among autonomous information systems. The principal result of this effort is KQML, the Knowledge Query and Manipulation Language. Other working groups include the Knowledge Interchange Format (KIF) working group, the Ontologies working group, and the Knowledge Representations Systems Standards (KRSS) working group.

The Knowledge Sharing Effort has received some direct funding from DARPA, the NSF and AFOSR for organization and coordination. In addition, many of the members of the External Interfaces Working Group are funded through research contracts from these and other agencies.

The development of KQML has been influenced, in particular, by two prototypical agent-based systems. The first is part of the DARPA/Rome Planning Initiative, and involves wide-area communication among planning, scheduling, resource control, and temporal reasoning programs [reference?]. These programs were written in combinations of LISP, Prolog, and C++, they run on a variety of workstation platforms, and communicate over TCP/IP connections.

The second is the Palo Alto Collaborative Testbed (PACT), which demonstrated collaborative distributed design, validation, and prototyping of an electromechanical device. PACT involves metropolitan-area communication among software, circuit, power, sensor, and mechanical CAD systems [PACT-ref]. These systems were written in combinations of LISP and C/C++, they run on a variety of workstation and personal computer platforms, and communicate using either TCP/IP or SMTP (electronic mail) connections (cf. Appendix ??).

1 Introduction

Modern computing systems often involve multiple intergenerating computations/nodes. Distinct, and often autonomous nodes can be viewed as agents performing within the overall system, in response to messages from other nodes. There are several levels at which agent-based systems must agree, at least in their interfaces, in order to successfully interoperate:

Transport: how agents send and receive messages;

Language: what the individual messages mean;

Policy: how agents structure conversations;

Architecture: how to connect systems in accordance with constituent protocols.

This document is mostly about the language level. This document specifies the syntactic and semantic fundamentals of the Knowledge Query and Manipulation Language (KQML).

KQML is complementary to work on representation languages for domain content, including the DARPA Knowledge Sharing Initiative's Knowledge Interchange Format (KIF). KQML has also been used to transmit object-oriented data, and a wide range of information can be accumulated. KQML is a language for programs to use to communicate attitudes about information, such as querying, stating, believing, requiring, achieving, subscribing, and offering. KQML is indifferent to the format of the information itself, thus KQML expressions will often contain subexpressions in other so-called "content languages."

KQML is most useful for communication among agent-based programs, in the sense that the programs are autonomous and asynchronous. Autonomy entails that agents may have different and even conflicting agendas; thus the meaning of a KQML message is defined in terms of constraints on the message sender rather than the message receiver. This allows the message receiver to choose a course of action that is compatible with other aspects of its function. Of course, most useful agent architectures strive for maximal cooperation among agents, but just as with human organizations, complete cooperation is not always possible.

KQML is complementary to new approaches to distributed computing, which focus on the transport level. For example, the new and popular Object Request Broker [OMG ORB] specification defines distributed services for interprocess and interplatform messaging, data type translation, and name registration. It does not specify a rich set of message types and their meanings, as does KQML.

A KQML message is called a *performative*, in that the message is intended to perform some action by virtue of being sent. (The term is from speech act theory.) This document defines a substantial number of performatives in terms of what they connote about the sender's knowledge.

However, we recognize that the performatives defined herein are neither necessary nor sufficient for all agent-based applications. Therefore, agents need not support the entire set of defined performatives (indeed, we expect that agents will usually support a small subset), and agents may use performatives that do not appear in this specification. New performatives should be defined precisely, and in the style of this specification.

The performative names in this specification are **reserved**; an application is not KQML-compliant if it uses these performatives in ways that are inconsistent with the definitions in this specification. We encourage implementors to use these reserved performatives when possible, to increase overall interoperability.

The primary dimension of KQML extension is through the definition of new performatives. The definitions of new performatives must explicitly describe all permissible parameters, and when applicable, default values for parameters that do not appear in particular messages. A performative definition may coin new parameter names; however, we encourage the use of the parameter names in this specification when they apply.

Besides KQML, at the language level of interoperation, this document touches on issues at the other three levels. Appendix A describes work-in-progress on KQML APIs that provide a definition of and code for the transport level, Appendix B defines some useful terms for describing messaging policies (e.g., timely responses, pertinent communications), and Appendix C describes the architectures of some existing agent-based systems. Our intent is for these terms and examples to make other agent-based systems easier to characterize and compare for the task of achieving high-level interoperation.

This specification is written in the style of an Internet RFC. That is, the main thread of this document is a dry description of what KQML *is*; comments regarding motivation for particular aspects are relegated to inset NOTEs. Also, the latter portion of this document describes several example systems that use (or in one case, could use) KQML messages.

1.1 KQML Transport Assumptions

It is not the intent of this document to standardize a programming interface, much less a system infrastructure, for message transport. Such issues are usually dominated by implementation considerations, including programming language choice, network services and security.

Nevertheless, this document does intend to make prescriptions regarding an agent communication language, and this requires a model of message transport. So for these purposes, we define the following abstraction of the transport level:

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

> NOTE: The latter property is less useful than it may appear, unless there is a guarantee of *agent reliability* as well. Such a guarantee is a policy issue, and may vary among systems.

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

Conversely, higher levels can implement a variety of different communication abstractions. For example, a star architecture (cf. Section ??) where the hub handles `broadcast` (cf. Section ??) messages provides a virtual broadcast communication abstraction. A hierarchical architecture may provide a virtual content-based multicast abstraction (cf. Section ??). The use of the `pipe` message produces a virtual connection-oriented approach to message transport.

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

2 KQML String Syntax

A KQML message is also called a *performative*. A performative is expressed as an ASCII string using the syntax defined by this section. This syntax is a restriction on the ASCII representation of Common Lisp Polish-prefix notation.

> NOTE: We chose the ASCII-string LISP list notation because it is readable by humans, simple for programs to parse (particularly for many knowledge-based programs), and transportable by many inter-application messaging platforms. However, no choice of message syntax will be both convenient and efficient for all messaging APIs; Appendix ?? describes some alternate syntaxes for particular applications.

Unlike Lisp function invocations, parameters in performatives are indexed by keywords and therefore order independent. These keywords, called *parameter names*, must begin with a colon (:) and must precede the corresponding *parameter value*.

> NOTE: Performative parameters are identified by keywords rather than by their position due to a large number of optional parameters to performatives.

Several examples of the syntax appear in Section 5 of this document.

2.1 KQML string syntax in BNF

The BNF given in Figure ?? assumes definitions for <ascii>, <alphabetic>, <numeric>, <double-quote>, <backslash>, and <whitespace>. "*" means any number of occurrences, and "-" indicates set difference. Note that <performative> is a specialization of <expression>.

> NOTE: In length-delimited strings, e.g. "#3"abc", the whole number before the double-quote specifies the length of the string after the double-quote.

```
<performative> ::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)

<expression> ::= <word> | <quotation> | <string> |
                 (<word> {<whitespace> <expression>}*)

<word> ::= <character><character>*

<character> ::= <alphabetic> | <numeric> | <special>

<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
              @ | $ | % | : | . | ! | ?

<quotation> ::= '<expression> | `<comma-expression>

<comma-expression> ::= <word> | <quotation> | <string> | ,<comma-expression>
                       (<word> {<whitespace> <comma-expression>}*)

<string> ::= "<stringchar>*" | #<digit><digit>*"<ascii>*

<stringchar> ::= \<ascii> | <ascii>-\-<double-quote>
```

Figure 1: KQML string syntax in BNF

3 KQML Semantics

The semantic model underlying KQML is a simple, uniform context for agents to view each others' capabilities. Each agent appears, on the outside, as if it manages a knowledge base (KB). That is, communication with the agent is with regard to this KB base, e.g., questions about what a KB contains, statements about what a KB contains, requests to add or delete statements from the KB, or requests to use knowledge in the KB to route messages to appropriate other agents.

The implementation of an agent is not necessarily structured as a knowledge base. The implementation may use a simpler database system, or a program using a special datastructure, as long as wrapper code translates that representation into a knowledge-based abstraction for the benefit of other agents. Thus we say that each agent manages a *virtual* knowledge base (VKB).

When defining performatives, it is useful to classify the statements in a VKB into two categories: beliefs and goals. An agent's beliefs encode information it has about itself and its external environment, including the VKBs of other agents. An agent's goals encode states of its external environment that the agent will act to achieve. Performative definitions make reference to either or both of an agent's goals and beliefs, e.g., that the agent wants another agent to send it a certain class of information. The English-prose performatives in this document make reference to these terms, but this view of the VKB is especially important in the formal semantics of KQML [SEMANTICS].

Agents talk about the contents of theirs and other's VKBs using KQML, but the encoding of statements in VKBs can use a variety of representation languages. That is, the KQML performative `tell` is used to specify that a particular string is contained in an agent's belief store, but the encoding of that string can be a representation language other than KQML.

The only restrictions on such a representation is that it be sentential, so that expressions using that representation can be viewed as entries in a VKB, and that sentences have an encoding as an

Keyword	*Meaning*
`:content`	the information about which the performative expresses an attitude
`:force`	whether the sender will ever deny the meaning of the performative
`:in-reply-to`	the expected label in a reply
`:language`	the name of represenation language of the `:content` parameter
`:ontology`	the name of the ontology (e.g., set of term definitions) used in the `:content` parameter
`:receiver`	the actual receiver of the performative
`:reply-with`	whether the sender expects a reply, and if so, a label for the reply
`:sender`	the actual sender of the performative

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

ascii string, so that sentences can be embedded in KQML messages. Fortunately, these restrictions appear to hold for the representations of interest to KQML users, including AI languages, database languages, object-oriented representations, and many CAD formats.

4 Reserved Performative Parameters

As described in Section 2, performatives take parameters identified by keywords. This section defines the meaning of some common performative parameters, by coining their keywords and describing the meaning of the accompanying values. This will facilitate brevity in the performative definitions of Section 5, since the following parameters are used heavily.

The following parameters are *reserved* in the sense that any performative's use of parameters with those keywords must be consistent with the definitions below. These keywords and information parameter meanings are summarized in Table 1.

> NOTE: The specification of reserved parameter keywords is useful in at least two ways: 1) to mandate some degree of uniformity on the semantics of common parameters, and thereby reduce programmer confusion, and 2) to support some level of understanding, by programs, of performatives with unknown names but with known parameter keywords.

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

These parameters convey the actual sender and receiver of a performative, as opposed to the virtual sender and receiver in the `:from` and `:to` parameters of networking performatives (cf. Section 5.10).

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

If the `<expression>` is the word `nil` or this parameter is absent from the performative, then the sender does not expect a reply. If the `<expression>` is the word `t` then the sender expects a reply. Otherwise, the sender expects a reply containing a `:in-reply-to` parameter with a value identical to `<expression>`.

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

The `:content` parameter indicates the "direct object" (in the linguistic sense) of the performative. For example, if the performative name is `tell` then the `:content` will be sentence being told. The `<expression>` in the `:content` parameter must be a valid expression in the representation language specified by the `:language` parameter, or KQML if the `:language` parameter does not appear. Furthermore, the constants used in the `expression` must be a subset of those defined by the ontology named by the `:ontology` parameter, or the standard ontology for the representation language if the `:ontology` parameter does not appear.

> NOTE: Both `:language` and `:ontology` are restricted to only take `<word>`s as values, and therefore complex terms, e.g., denoting unions of ontologies, are not allowed. We do believe that it will be important to support a calculus of ontologies and languages, but we feel that its proper place is in performatives that define new KQML names. This way, only those agents that can process extensional performatives are expected to understand such a calculus.

`:force <word>`

If the value of this parameter is the word `permanent`, then the sender guarantees that it will never deny the meaning of the performative. Any other value indicates that the sender may deny the meaning in the future. (This parameter exists to help agents avoid unnecessary truth-maintenance overhead.) The default value is `tentative`.

Name	*Section*	*Meaning*
`achieve`	5.6	S wants R to do make something true of their environment
`advertise`	5.8	S is particulary-suited to processing a performative
`ask-about`	5.4	S wants all relevant sentences in R's VKB
`ask-all`	5.4	S wants all of R's answers to a question
`ask-if`	5.4	S wants to know if the sentence is in R's VKB
`ask-one`	5.4	S wants one of R's answers to a question
`break`	5.10	S wants R to break an established pipe
`broadcast`	5.10	S wants R to send a performative over all connections
`broker-all`	5.11	S wants R to collect all responses to a performative
`broker-one`	5.11	S wants R to get help in responding to a performative
`deny`	5.1	the embedded performative does not apply to S (anymore)
`delete`	5.2	S wants R to remove a ground sentence from its VKB
`delete-all`	5.2	S wants R to remove all matching sentences from its VKB
`delete-one`	5.2	S wants R to remove om matching sentence from its VKB
`discard`	5.7	S will not want R's remaining responses to a previous performative
`eos`	5.5	end of a stream of responses to an earlier query
`error`	5.3	S considers R's ealier message to be mal-formed
`evaluate`	5.4	S wants R to simplify the sentence
`forward`	5.10	S wants R to route a performative
`generator`	5.7	same as a `standby` of a `stream-all`
`insert`	5.2	S asks R to add content to its VKB
`monitor`	5.9	S wants updates to R's response to a `stream-all`
`next`	5.7	S wants R's next response to a previously-mentioned performative
`pipe`	5.10	S wants R to route all further performatives to a another agent
`ready`	5.7	S is ready to respond to R's previously-mentioned performative
`recommend-all`	5.11	S wants all names of agents who can respond to a performative
`recommend-one`	5.11	S wants the name of an agent who can respond to a performative
`recruit-all`	5.11	S wants R to get all suitable agents to respond to a performative
`recruit-one`	5.11	S wants R to get another agent to respond to a performative
`register`	5.10	S can deliver performatives to some named agent
`reply`	5.4	communicates an expected reply
`rest`	5.7	S wants R's remaining responses to a previously-mentioned performative
`sorry`	5.3	S cannot provide a more informative reply
`standby`	5.7	S wants R to be ready to respond to a performative
`stream-about`	5.5	multiple-response version of `ask-about`
`stream-all`	5.5	multiple-response version of `ask-all`
`subscribe`	5.9	S wants updates to R's response to a performative
`tell`	5.1	the sentence in S's VKB
`transport-address`	5.10	S associates symbolic name with transport address
`unregister`	5.10	a `deny` of a `register`
`untell`	5.1	the sentence is not in S's VKB

Table 2: Summary of reserved performatives, for sender S and recipient R.

5 Reserved Performative Names

In this section, we define several **reserved** performatives. That is, they are reserved in the sense that if an implementation uses any of the following performative names in a way that is inconsistent with the following performative definitions, then that implementation is not compliant with KQML. The reserved performatives and their meanings are summarized in Table 2.

In this section, we describe performative semantics in English prose. Since English prose is often ambiguous and sometimes self-contradictory, we have developed a framework for formal definition of performatives. A full description appears in a separate paper [Genesereth et al.].

Definitions of new performatives should follow the style of the definitions in this section. That is, a definition should convey the following:

- the performative name;
- all parameters keywords that the performative may contain;
- syntactic categories and semantics for all values of parameters with non-reserved keywords;
- any additional syntactic and semantic constraints for values of parameters with reserved keywords;
- the default values of all absent parameters;
- the semantics, in terms of a statement the sender is making of itself, of the performative name applied to the parameters.

5.1 Basic informative performatives

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

Performatives of this type indicate that the `:content` sentence is in the sender's virtual knowledge base (VKB) (cf. Section 3).

```
deny
     :content <performative>
     :language KQML
     :ontology <word>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the meaning of the embedded `<performative>` is *not* true of the sender. A `deny` of a `deny` cancels out.

```
untell
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
```

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

A performative of this type is equivalent to a `deny` of a `tell`.

> NOTE: `untell` weaker than telling the negation of the sentence; the sender may not have the negation in its VKB either.

> NOTE: Inclusion of `untell` performative is obviously redundant; in this document, perspecuity takes precedence over minimality.

5.2 Database performatives

These performatives, INSERT, DELETE, etc. provide an ability for one agent to request another agent to insert or delete sentences in its VKB.

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

The sender requests the receiver to add the `:content` sentence to its VKB. The performative can either fail or succeed. Possible errors and warning conditions are:

- Content duplicates sentence already in VKB.
- Content contradicts sentence already in VKB.
- Sender is not authorized to INSERT content.
- ...

```
delete
     :content <performative>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

The sender requests the receiver to delete the `:content` sentence from its VKB. The sentence must be ground. The performative can either fail or succeed. Possible errors and warning conditions are:

- Content not ground.
- Content not in VKB.
- Content necessarily true in VKB.
- Sender is not authorized to DELETE content.
- ...

```
delete-one
     :content <performative>
     :aspect <expression>
     :order { first | last | undefined }
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

The sender requests the receiver to delete one sentence from its VKB which matches `:content`. Note that performatives of this type make most sense with languages that define schema variables.

The `:aspect` parameter describes the form of the desired reply; for the match of the deleted `:content` in the recipient's VKB, the reply will be the `:aspect` with all of its schema variables replaced by the values bound to the corresponding schema variables in deleted sentence. The value of the `:aspect` parameter defaults to the value of the `:content` parameter. if the `:aspect` is NIL, then no response will be given for a successful deletion.

The optional `:order` parameter specifies whether the sentence to be deleted should be the first or last one found in the VKB (this will only make sense to some agents (e.g. Prolog based ones)). The default value for the `:order` parameter is `undefined`.

The performative can either fail or succeed. Possible errors and warning conditions are:

- No sentence matching content in VKB.
- Content necessarily true in VKB.
- Sender is not authorized to DELETE content.
- ...

```
delete-all
     :content <performative>
     :aspect <expression>
     :language KQML
     :ontology <word>
     :reply-with <expression>
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

This performative is like `delete-one`, except that the reply should be a collection of instantiated aspects corresponding to all deleted sentences matching the `:content`.

The performative can either fail or succeed. Possible errors and warning conditions are:

- No sentence matching content in VKB.
- All Content necessarily true in VKB.
- Sender is not authorized to DELETE content.
- ...

5.3 Basic responses

```
error :in-reply-to <expression>
      :sender <word>
      :receiver <word>
      :comment <string>
      :code <integer>
```

A performative of this type indicates that the sender can not understand or considers to be illegal the message referenced by the `:in-reply-to` parameter. The :CODE parameter gives a numeric code to classify the error type. The :COMMENT parameter can be used to return a string further describing how the sender considers the message to be ill formed.

```
sorry :in-reply-to <expression>
      :sender <word>
      :receiver <word>
      :comment <string>
```

A performative of this type indicates that the sender understands, but is not able to provide any (more) response(s) to the message referenced by the `:in-reply-to` parameter. A performative of this type may be used in response to an `evaluate` or `ask-one` query, when no other reply is appropriate. The optional :COMMENT parameter can be used to pass a string which describes the specifics of situation leading to refusal to provide a response or additional responses.

5.4 Basic query performatives

```
evaluate
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the sender would like the recipient to simplify the expression in the `:content` parameter, and reply with the result. (Simplification is a language specific concept, but it should subsume "believed equal".)

```
reply
     :content <expression>
     :language <word>
     :ontology <word>
     :in-reply-to <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

Performatives of this type indicate that the sender believes that `:content` is an appropriate reply to the query in the `:in-reply-to` message.

```
ask-if
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is the same as `evaluate`, except that the `:content` must be a *sentence schema* in the `:language`. In other words, the sender wishes to know if the `:content` matches any sentence in the recipient's VKB.

```
ask-about
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is like `ask-if`, except that the reply should be the collection of all sentences in the recipient's VKB that contain a sentence or term that matches the sentence or term schema in the `:content`. Note that the reply `:language` and `:ontology` must include a "collection" construct (e.g., sets, lists, bags, etc.).

```
ask-one
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is like an `ask-if`, except that the `:aspect` parameter describes the form of the desired reply; for some match of the `:content` in the recipient's VKB, the reply will

Agent A sends the following performative to agent B:

```
(evaluate :language KIF :ontology motors :reply-with q1
       :content (val (torque motor1) (sim-time 5)))
```

and agent B replies with:

```
(reply :language KIF :ontology motors :in-reply-to q1
       :content (scalar 12 kgf))
```

Figure 2: In this example of basic query performatives, agent A asks agent B a simple query and receives a response via a tell.

be the `:aspect` with all of its schema variables replaced by the values bound to the corresponding schema variables in `:content`. The value of the `:aspect` parameter defaults to the value of the `:content` parameter. Note that performatives of this type make most sense with languages that define schema variables.

```
ask-all
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type is like `ask-one`, except that the reply should be a collection of instantiated aspects corresponding to all matches of the `:content` sentences on the recipient's VKB.

```
sorry
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type indicates that the sender understands, but is not able to provide any (more) response(s) to the message referenced by the `:in-reply-to` parameter. A performative of this type may be used in response to an `evaluate` or `ask-one` query, when no other reply is appropriate. It may also be used as the last response to a multi-response query performative (e.g., the performatives in the next section).

5.5 Multi-response query performatives

```
stream-about
     :content <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
```

Agent `A` sends the following performative to agent `B`:

```
(stream-about :language KIF :ontology motors :reply-with q1
              :content motor1)
```

and agent `B` replies with a series of performatives:

```
(tell :language KIF :ontology motors :in-reply-to q1
      :content (= (val (torque motor1) (sim-time 5)) (scalar 12 kgf))
(tell :language KIF :ontology structures :in-reply-to q1
      :content (fastens frame12 motor1))
(eos :in-reply-to q1)
```

Figure 3: Agent A asks B to tell all it knows about motor1. B replys with a sequenct of tells terminated with a sorry.

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

This type is like `ask-about`, except that rather than replying with the collection of matches, the responder should send a series of performatives that when taken together identify the members of that collection.

```
stream-all
     :content <expression>
     :aspect <expression>
     :language <word>
     :ontology <word>
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type is like `ask-all`, except that rather than replying with the collection of instantiated aspects, the responder should send a series of performatives that when taken together identify the members of that collection.

```
eos    :in-reply-to <expression>
       :sender <word>
       :receiver <word>
```

The "End Of Stream" performative indicates that the sequence of responses to an earlier multi-response message (e.g., stream-all) :IN-REPLY-TO has terminated successfully. No more responses will be sent.

5.6 Basic effector performatives

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

Agent A sends the following performative to agent B:

```
(achieve :language KIF :ontology motors :reply-with q1
         :content (= (val (torque motor1) (sim-time 5))
                     (scalar 2 kgf))
```

and after achieving the requested motor torque, agent B might send the following (though it is not expected):

```
(tell :language KIF :ontology motors
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 2 kgf))
```

Figure 4: Agent A tells B to achieve a state in which the the torque of motor1 is a particular value.

```
    :ontology <word>
    :force <word>
    :sender <word>
    :receiver <word>
```

Performatives of this type are requests that the recipient try to make the sentence in `:content` true of the system (technically, that the sender wants the recipient to want to make the sentence true of the system).

```
unachieve
    :content <expression>
    :language <word>
    :ontology <word>
    :sender <word>
    :receiver <word>
```

A performative of this type is the same as a `deny` of an `achieve`.

5.7 Generator performatives

The following performatives comprise a *generator* mechanism for the delivery of responses to a KQML performative. That is, this mechanism allows an agent to explicitly retrieve responses in a series; this is especially useful when there are a large number of responses, and/or the agent is not able to efficiently buffer incoming responses.

```
standby
    :content <performative>
    :language KQML
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wants the recipient to take the would-be response(s) from the performative in `:content`, and announce its readiness to accept requests for the responses.

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

This type indicates that the sender will answer requests for the responses to the performative contained in some performative with the `:in-reply-to` label. The `:reply-with` parameter is, in function, the returned generator.

```
next
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wishes to receive the next response from those promised by the performative identified by the `:in-reply-to` parameter.

> NOTE: The `next` performative does not have a `:reply-with` parameter because the `:in-reply-to` parameter of the next response should match the `:reply-with` parameter of the performative embedded in the original `standby` message.

```
rest
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wishes to receive the remaining responses, in a stream, from those promised by the `ready` performative identified by the `:in-reply-to` parameter.

```
discard
    :in-reply-to <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender will issue no more replies to the `ready` performative identified by the `:in-reply-to` parameter. (This is a courtesy to the owner of the generator, so it can reclaim resources needed to maintain the generator.)

```
generator
    :content <expression>
    :aspect <expression>
    :language <word>
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

Agent A sends the following performative to agent B:

```
(standby :language KQML :ontology K10 :reply-with g1
         :content (stream-about :language KIF
                                :ontology motors
                                :reply-with q3
                                :content motor1))
```

and agent B replies with:

```
(ready :reply-with 2F0B :in-reply-to g1)
```

then agent A follows with:

```
   (next :in-reply-to 2F0B)
```

to which B replies with:

```
(tell :language KIF :ontology motors :in-reply-to q3
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 12 kgf))
```

and so on, until A sends:

```
(discard :in-reply-to 2F0B)
```

Figure 5: In this example, agent A asks B to prepare to generate a stream of all of the information it knows about motor1. Agen B replys that it is ready and returns an identifier for A to use in requesting the individual facts. Agent A asks for a number of facts and finally indicates that no more are required.

This type is the same as:

```
(standby
     :content (stream-all :content <expression>
          :aspect <expression>
          :language <word>
          :ontology <word>
          :sender <word>
          :receiver <word>)
     :language KQML
     :reply-with <expression>)
```

5.8 Capability-definition performatives

```
advertise
     :content <performative>
     :language KQML
     :ontology <word>
     :force <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender is particularly suited to process the class of KQML performatives described by the `:content` parameter. If the embedded performative is missing any parameters (defined for the embedded performative), then those parameters may take any otherwise legal values.

5.9 Notification performatives

```
subscribe
     :content <performative>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wishes the recipient to tell it about future changes to what would be the response(s) to the KQML performative in the `:content` parameter.

```
monitor
     :content <expression>
     :ontology <word>
     :language <word>
     :reply-with <expression>
     :force <word>
     :sender <word>
     :receiver <word>
```

This type is the same as:

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

5.10 Networking performatives

```
register
     :name <word>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender can deliver performatives to the agent named by the `:name` parameter (this subsumes the case when the sender calls itself by this name).

Agent B sends the following performative to agent A:

```
(advertis :language KQML :ontology K10
           :content (subscribe :language KQML
                               :ontology K10
           :content (stream-about :language KIF
                                  :ontology motors
                                  :content motor1)))
```

to which agent B responds with:

```
(subscribe :reply-with s1
           :language KQML :ontology K10
           :content (stream-about :language KIF
                                  :ontology motors
                                  :content motor1))
```

then agent A follows with this stream of performatives over time:

```
(tell :language KIF :ontology motors :in-reply-to s1
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 12 kgf))
(tell :language KIF :ontology structures :in-reply-to s1
      :content (fastens frame12 motor1))
(untell :language KIF :ontology motors :in-reply-to s1
        :content (= (val (torque motor1) (sim-time 5))
                    (scalar 12 kgf))
(tell :language KIF :ontology motors :in-reply-to s1
      :content (= (val (torque motor1) (sim-time 5))
                  (scalar 13 kgf))
...
```

Figure 6: In this example, agent A announces that it is willing to accept subscriptions from other agents who would like to find out about motor1. Agent B tells A that it would indeed like to receive a stream of information about motor1. A then supplies the stream.

```
unregister
      :name <word>
      :sender <word>
      :receiver <word>
```

This type is the same as a `deny` of a `register`.

```
forward
      :to <word>
      :from <word>
      :content <performative>
      :language KQML
      :ontology <word>
      :sender <word>
      :receiver <word>
```

This type indicates that the sender wants the `:to` agent to process the performative in the `:content` parameter as if it came from the `:from` agent directly. It is important that the `:to` agent receive the package, not just the performative, or it will think that the performative is from the next-to-last step in the path.

> NOTE: This will normally entail that the response(s) are also wrapped in forward(s), since the responder will want to deliver the response(s) to the requesting agent, and achieving this may involve the use of package or other networking performatives. However, it is possible that agent `A` must use a package to send a performative to `B`, but `B` can send a performative to `A` directly.

> NOTE: Previous versions of KQML defined three levels of KQML syntax – the communication (package) layer, the message layer, and the content layer. The current approach is a proper generalization, since the layers arise from the embedding of performatives.

```
broadcast
      :from <word>
      :content <expression>
      :ontology <word>
      :language <word>
      :sender <word>
      :receiver <word>
```

This type indicates that the sender would like the recipient to route the `broadcast` performative to each of its outgoing connections, unless the recipient has already received a `broadcast` performative with this `:reply-with` (for cycle detection).

```
pipe
      :to <word>
      :from <word>
      :reply-with <expression>
      :sender <word>
      :receiver <word>
```

This type indicates that future traffic on this channel should be routed to the `:to` agent, as if `:to` and `:from` were directly connected. Furthermore, the recipient is expected to pass the `pipe` performative toward the `:to` agent. Like `forward`, it is important that the destination receive the `pipe` performative, so that it knows that performatives from the next-to-last agent on the path come from the `:from` agent.

```
break
     :in-reply-to <expression>
     :sender <word>
     :receiver <word>
```

A performative of this type breaks a pipe. The `:in-reply-to` parameter value must match the `:reply-with` value of a previous `pipe` performative. Not only is the recipient of a `break` expected to cease piped routing, but it is also expected to pass the `break` up the pipe. This will have the effect of dismantling the pipe in the opposite direction in which it was built.

```
transport-address
     :name <word>
     :content <expression>
     :language <word>
     :ontology <word>
```

The `transport-address` performative is a way to define an association between a symbolic name for a KQML agent and a transport address. For example, the following expression asserts that the sender uses the name `A` to refer to the agent who can be contacted at `eitech.com 4000`

```
     (transport-address :agent A :content (eitech.com 4000)
                        :language s-expressions :ontology tcp-host-port)
```

such peformatives can be TELLed, ASKed, MONITORed, etc.

5.11 Facilitation performatives

```
broker-one
     :content <expression>
     :ontology <word>
     :language KQML
     :reply-with <expression>
     :sender <word>
     :receiver <word>
```

This type indicates that the sender wants the recipient to process the embedded performative through the help of a single agent that is particularly suited to processing the embedded performative. (Presumably, such suitability was established using `:advertise` performatives.)

```
broker-all
```

```
    :content <expression>
    :ontology <word>
    :language KQML
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

This type is similar to `broker-one` except that the sender wants the recipient to enlist the help of *all* agents particularly suited to processing the embedded performative. The recipient of the `broker-all` replies with a list of all responses.

```
recommend-one
    :content <expression>
    :ontology <word>
    :language KQML
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wants the recipient to reply with the name of a single agent that is particularly suited to processing the embedded performative.

```
recommend-all
    :content <expression>
    :language KQML
    :ontology <word>
    :reply-with <expression>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wants the recipient to reply with a list of names of agents that are particularly suited to processing the embedded performative.

```
recruit-one
    :from <word>
    :content <expression>
    :language KQML
    :ontology <word>
    :sender <word>
    :receiver <word>
```

This type indicates that the sender wants the recipient to forward the embedded performative to a single agent that is particularly suited to processing the embedded performative. This differs from `broker-one` because the recruited agent will forward its response directly to the original sender.

```
recruit-all
```

```
:from <word>
:content <expression>
:ontology <word>
:language KQML
:sender <word>
:receiver <word>
```

This type is similar to `recruit-one` except that the sender wants the recipient to forward the embedded performative to all agents particularly suited to processing the embedded performative. The recruited agents individually forward their responses to the original sender.

6 Proposed Performatives

This section documents some proposed performatives which are currently being discussed and/or reviewed. As the group reaches a consensus on these proposed performatives, they will be included in other section or deleted. They are included in this document to give the reader an accurate picture of the evolving specification and to encourage discussion of these proposals.

References

[1] Genesereth, M. R., Fikes, R. E., et al.: "Knowledge Interchange Format Version 3.0 Reference Manual", Logic-92-1, Stanford University Logic Group, 1991.

[2] Genesereth, M. R.: "An Agent-Based Approach to Software Interoperation", Logic-91-6, Stanford University Logic Group, 1991.

A Example Agent Policies

Agent-based software needs more than just a language for agents to describe their belief and wants. Agents need motivation for performing these communicative acts in terms of expectations about a helpful response. The shared expectations about message-passing behavior, e.g., helpfulness, responsiveness, commitment, etc., comprise the agents' protocols.

There is no single collection of protocols necessary for agenthood. The protocols of a particular system should be optimized for the constituent programs and the task at hand. In this specification, we merely list several protocols that may be useful in many applications. Other protocols, say for skepticism, bidding, reimbursement, and security, should be defined in this manner.

honesty a message's KQML semantics apply to the sender.

gullibility agents adopt the beliefs of others that are consistent with their own.

helpfulness agents adopt the goals of others that are consistent with their own

responsiveness agents will eventually respond to every received performative for which a response is expected

> NOTE: this protocol folds in two important constraints: that an agent will eventually process every performative, and that it will generate some sort of response whenever responses are expected. The purpose of the latter constraint is to force a response like "sorry" to performatives that just happen to not produce any other responses. Of course, the meaning of this is totally wrapped-up in the word "expected"; the intent is that response(s) are expected from a performative like "ask", but not "tell". "advertise" is trickier, but even though responses are possible, or even commonplace, they are not "expected".

empathy agents have a built-in way of determining what performatives are needed by others (i.e., without needing an explicit performative to which to respond)

pertinence agents will not send performatives that they believe will not benefit others

identity agents will never register a networking name that is identical with the name of another agent on the same network

Figure 7: The ABSE federation architecture

B Example Agent Architectures and Implementations

B.1 Content-based routing architecture (ala DRPI)

(to be completed by a representative of DRPI)

applications talk to router interface libraries, communicating content, however they want. We can call it KQML if we want, but it is language- specific intraprocess communication so no need to overregulate it. Router interface libraries communicate with router agents using advertise/publish/subscribe (so-called declarations). Routers talk to each other using advertise/publish/subscribe, maybe broadcast, for the exchange of declarations, and using packages for delivery to end-agents.

B.2 Agent-Based System Engineering (ABSE)

The ABSE project is a collaboration between the Stanford University Logic Group and Hewlett-Packard Palo Alto Research Laboratories. The ABSE architecture is a network of application agents (referred to simply as *agents*, below) connected through *facilitator* agents.

Agents and facilitators are linked together in what is often called a *federation architecture*. Figure **??** illustrates this architecture for the simple case in which there are just three machines, one with three agents and two with two agents apiece. As suggested by the diagram, agents do not communicate directly with each other. Instead, they communicate only with their local facilitators, and facilitators communicate with each other. In effect, the agents form a "federation" in which they surrender their communication autonomy to the facilitators; hence, the name of the architecture.

Messages from agents to facilitators may be directed or undirected. Undirected messages have content but no addresses. It is the responsibility of the facilitators to route such messages to agents able to handle them. In performing this task, facilitators can go beyond simple pattern match – they can translate messages, they can decompose problems into subproblems, and they can schedule the work on those subproblems. In some cases, this can be done interpretively (with messages going through the facilitator); in other cases, it can be done in one-shot fashion (with the facilitator setting up specialized links between individual agents and then stepping out of the

Figure 8: The PACT architecture

picture).

To accomplish the above. facilitators handle the reserved KQML performatives `forward`, `broker-one`, `broker-all`, and `register`. In addition, facilitators exploit the definitions of the reserved parameters `:content`, `:language`, and `:ontology` to perform representation language translation. When taken together, agents handle a wide variety of reserved KQML performatives, including `evaluate`, `ask-about`, `reply`, `deny`, and `generator`.

B.3 Palo Alto Collaborative Testbed

The PACT experiments show how pre-existing engineering software systems can be combined to constitute a distributed system of integrated design information and services. The PACT architecture encapsulates each component system with an *information* agent, which serves to bridge the idiosynchrosies of access to that system's knowledge and abilities (see Figure 8). Information agents use KQML as their agent communication language, with KIF as the exclusive representation language. Information agents are connected as needed, in part through an ABSE post-office agent (cf. Section **??**).

The experiments involved four geographically distributed engineering teams, collaborating on scenarios of design, fabrication, and redesign of a robotic manipulator. Each of the four design environments in PACT was used to model a different aspect of the manipulator (controller software, rigid body dynamics, encoder circuitry, sensors, and power system) and to reason about it from the standpoint of a different engineering discipline. Collaborative design tasks were performed including dynamics model exchange between the controls agent and dynamics agent, fine-grained cooperative distributed simulation exercising each aspect supported by the four tools, and finally design modifications suggested by the simulation. Each team was supported by its own computational environment linked via the PACT framework [Singh and Genesereth][Genesereth92] over the Internet.

The challenge in PACT was to take four existing systems, each already a specialized framework, and to integrate them via a flexible, higher-level framework. Framework building requires commitments from each party desiring participation in the shared environment to establish interface agreements and protocols of interaction. To drive the experiments with concrete goals, scenarios of interoperation among the various concurrent engineering tools, initially thwarted by tool isolation, were proposed. Next a series of interpersonal interactions were conducted among the developers of the various tools to identify the necessary information that bridged tool perspectives and enabled the execution of the driving design scenarios. Once the types of enabling knowledge had been identified (components, connectivity, attribute features, time varying values, equational functional models, etc.), agreements were reached on the form of the shareable knowledge. As a result of these interactions, an implicit ontology was created reflecting offline agreements. Over the course of the PACT experiments, the ontologies were explicitly encoded in KIF and Ontolingua Gruber92].After the form and semantics of the knowledge content had been agreed upon, an KQML-like language Agent Communication Language was specified to allow expressions of attitude toward knowledge content such as belief, disbelief, and interest.

Each tool has been wrapped up as an information agent available as a service to other agents. Tool-specific wrappers were constructed for each tool to translate into and out of the shared ontology and to manage the tool's application programmer interface for reacting to requests and updates expressed within the KQML-like Agent Communication Language.

PACT employs a rich suite of performatives, indicating the diversity of the PACT architecture.

Networking The PACT framework provides an infrastructure postal service to allow agents to delegate all message delivery responsibilities. To utilize the postal service, individual agents employ the `register` performative to make the postal service aware of its presence. The postal service is capable of handling `forward` messages addressed to any registered agent. Other message traffic is point-to-point between agents to reduce the overhead of a centralized bottleneck (e.g. during a distributed simulation).

Notification The PACT experiments exercised concurrent engineering design scenarios. Embodied within the concept of concurrent engineering environment is the tenet that all affected parties of recent design changes will be notified of the change so they may assess the impact. Notification is triggered by detection of change in information of interest. The traditional query oriented approach for requesting existing known information does not support notification, since the interest is assumed to expire after the answer is returned. What is needed is a performative whose semantics convey the monitoring nature of a notification request. Consequently, heavy use is made of the `subscribe` performative to convey the conditions triggering a notification. As a simple example of the utility of `subscribe` within PACT, one agent (NextCut) posts a persistent interest in the type of motor applying torque to the manipulator arms. This way if the motor changes, the consequences of the change in the motor's features can be evaluated.

Facilitation PACT is currently building sophistication into the infrastructure to provide mechanisms for locating registered agents with capabilities suited to fulfilling specific information interests. This way an agent with an information need would allow the infrastructure to broker the service request to agents who have stated capabilities matching the request. To enable this process, service provider agents would be forced to advertise their capabilities via the `advertise` performative. Followup to these advertisements occurs using the `broker-one` and `recommend-one` performatives.

Generator, Multi-response To provide flexibility on the packaging of transmitted knowledge and support a local tool's paradigm, provide a variety of mechanisms to specify the form of interesting information. (For architectures which cannot handle asynchrony, provide mechanism to get all answers back at once, allow asynchronous incremental transmission for forward-chaining agents, or support generators).

B.4 Information bus architecture (ala TIB)

(to be completed by Jay Weber)

based on publish & subscribe, advertising of label names (with a hierarchical naming scheme) assumed a priori, clients subscribe to these names and servers publish. [Does TIB partition clients and servers? No real reason to, except it does consolidate recipients of subscriptions.]

C KQML APIs and Alternate Syntaxes

Architectures that use pre-existing message-passing platforms and agent implementations may find it easier and/or more efficient to use an alternative to the list syntax described in this specification.

C.1 The ABSE Lisp API

C.2 The DRPI TCP/IP API

C.3 KQeMaiL

There have been implementations that use e-mail as a transport mechanism between agents.

Is is reasonable to construe the UNIX sendmail process as an agent that processes package performatives.

We have a mapping between package performatives and Internet e-mail messages.

C.4 CORBA dynamic invocation interface

The Common Object Request Broker Architecture is a new standard for distributed object-oriented processes that has broad support from software/hardware vendors, standards organizations, and potential users. The CORBA is a reasonable and effective platform for agent-oriented software as well.

There is a simple mapping from the list syntax of this spec to the CORBA dynamic invocation interface.

D Future Work

This section notes some of the capabilities which are recognized as being needed or desired in KQML and some thoughts on how they might be realized. Any material here is highly speculative. It is included to provide the reader with an accurate vision of the complete language.