TR CS-97-03
A Proposal for a new KQML Specification
Yannis Labrou and Tim Finin
February 3, 1997
A Proposal for a new KQML Specification

Yannis Labrou and Tim Finin

Computer Science and Electrical Engineering Department (CSEE)
University of Maryland Baltimore County (UMBC)
Baltimore, Maryland 21250
email: {jklabrou,finin}@cs.umbc.edu

Abstract

We propose a new specification for the Knowledge Query and Manipulation Language (KQML). KQML is a language for the communication between software agents. KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests. Our starting point for the specification of KQML is [1]. Although the differences regarding the syntax of KQML messages and the reserved performative parameters are minimal, there are significant changes regarding the set of reserved performatives, their meaning and intended use.

NOTE: This document is not the official new KQML specification. It is intended as a proposal for a new KQML specification and the authors welcomes any comments.
Contents

1  KQML transport assumptions  1
2  KQML string syntax  2
3  Reserved performative parameters  3
4  The reserved performatives  5
   4.1  Discourse performatives  5
   4.2  Intervention and mechanics of conversation performatives  22
   4.3  Networking and Facilitation performatives  30
List of Tables

1 Summary of reserved parameter keywords and their meanings. .......... 3
2 Summary of reserved performatives for: sender S and: receiver R. .... 7
3 The set of performatives discussed in this document and their properties when used in conversations. .. 8
4 The performatives that may have a <performative>, i.e., a KQML message, as :content ................................. 9
5 The performatives that various kinds of agents may process. ....... 10

List of Figures

1 KQML string syntax in BNF ........................................ 2
2 An ask-all performative and the appropriate response. ............... 6
3 A stream-all performative and the appropriate responses. .......... 12
4 An insert performative following a related advertise, and an example of a proper uninsert............................ 15
5 An achieve performative and the appropriate response, later followed by an unachieve request. ......................... 18
6 An example of an advertise and appropriate follow-ups to that. .... 20
7 An example of an advertise of a subscribe of an ask-all. ........... 22
8 A subscribe request and appropriate responses. ..................... 23
9 Examples of the three situations that may result in an error. ........ 26
10 The exchange of Figure 3 when standby is used. .................. 27
11 The possible scenarios that the exchange of Figure 10 might continue with. 28
12 A conversation involving the forward performative. ............... 33
13 The rest of the exchange of Figure 13. ............................. 34
14 An example of a broker-one performative and the follow-up. ...... 36
15 An example of a recommend-one and a response to it. ............. 38
16 An example of a recruit-one and its follow-up. .................... 40
This document constitutes a proposal for a revision of the current KQML specification document ([1]). Although the differences regarding the syntax of KQML messages and the reserved performative parameters are minimal, there are significant changes regarding the set of reserved performatives, their meaning and intended use. Parts of Sections 1 and 2 appear in the current KQML specification document ([1]) and are included here for reasons of completeness of this presentation.

1 KQML transport assumptions

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

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

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

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

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

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

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

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

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

Figure 1: KQML string syntax in BNF
3 Reserved performative parameters

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

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

:sender <word>
:receiver <word>

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

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

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

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

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

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

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

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

- When referred to in text, performative names are written in italics, e.g., ask-all, tell, etc.

- In text, we use the names of reserved performative parameters to refer to their values, so :sender refers to the particular sender of a performative, :content refers to the content and so on.

- Occasionally, we use parameter_{formative} to refer to the value of a particular performative parameter, i.e., sender_{advertise} to refer to the sender of an advertise in a particular case.

- We use <performative> to refer to a particular instance of a performative.

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

4.1 Discourse performatives

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

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

Agent A sends the following performative to agent B. The to suggest that the ask-all follows a relevant advertise message.

\[
\text{(ask-all)}
\begin{align*}
  :\text{sender} & \quad A \\
  :\text{receiver} & \quad B \\
  :\text{in-reply-to} & \quad \text{id0} \\
  :\text{reply-with} & \quad \text{id1} \\
  :\text{language} & \quad \text{Prolog} \\
  :\text{ontology} & \quad \text{foo} \\
  :\text{content} & \quad "\text{bar(X,Y)"} \\
\end{align*}
\]

and agent B replies with the following KQML message

\[
\text{(tell)}
\begin{align*}
  :\text{sender} & \quad B \\
  :\text{receiver} & \quad A \\
  :\text{in-reply-to} & \quad \text{id1} \\
  :\text{reply-with} & \quad \text{id2} \\
  :\text{language} & \quad \text{Prolog} \\
  :\text{ontology} & \quad \text{foo} \\
  :\text{content} & \quad "\text{[bar(a,b),bar(c,d)]"} \\
\end{align*}
\]

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

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

\[
\text{(ask-all)}
\begin{align*}
  :\text{sender} & \quad \text{<word>} \\
  :\text{receiver} & \quad \text{<word>} \\
  * & \quad :\text{in-reply-to} \quad \text{<word>} \\
  & \quad :\text{reply-with} \quad \text{<word>} \\
  & \quad :\text{language} \quad \text{<word>} \\
  & \quad :\text{ontology} \quad \text{<word>} \\
  & \quad :\text{content} \quad \text{<expression>)} \\
\end{align*}
\]

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

\[
\text{(ask-one)}
\begin{align*}
\end{align*}
\]

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

Table 2: Summary of reserved performatives for :sender S and :receiver R.
<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Response Required</th>
<th>Response Only</th>
<th>No Response</th>
<th>:content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discourse</td>
<td>ask-if</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>ask-all</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>ask-one</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>stream-all</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>eos</td>
<td>X</td>
<td></td>
<td></td>
<td>empty</td>
</tr>
<tr>
<td></td>
<td>tell</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>untell</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>deny</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>uninsert</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>delete-one</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>delete-all</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>undelete</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>achieve</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>unachieve</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>advertise</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>unadvertise</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>subscribe</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td>Intervention</td>
<td>error</td>
<td>X</td>
<td></td>
<td></td>
<td>empty</td>
</tr>
<tr>
<td>and Mechanics</td>
<td>sorry</td>
<td>X</td>
<td></td>
<td></td>
<td>empty</td>
</tr>
<tr>
<td></td>
<td>standby</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>&lt;performative&gt;</td>
</tr>
<tr>
<td></td>
<td>ready</td>
<td>n/a</td>
<td>n/a</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>next</td>
<td>n/a</td>
<td>n/a</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rest</td>
<td>n/a</td>
<td>n/a</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>discard</td>
<td>n/a</td>
<td>n/a</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>Facilitation</td>
<td>register</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td>and Networking</td>
<td>unregister</td>
<td>X</td>
<td></td>
<td></td>
<td>empty</td>
</tr>
<tr>
<td></td>
<td>forward</td>
<td>:content</td>
<td>:performative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broadcast</td>
<td>:content</td>
<td>:performative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>transport-address</td>
<td>X</td>
<td></td>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
<tr>
<td></td>
<td>broker-one</td>
<td>:content</td>
<td>:performative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broker-all</td>
<td>:content</td>
<td>:performative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recommend-one</td>
<td>X</td>
<td></td>
<td></td>
<td>:performative</td>
</tr>
<tr>
<td></td>
<td>recommend-all</td>
<td>X</td>
<td></td>
<td></td>
<td>:performative</td>
</tr>
<tr>
<td></td>
<td>recruit-one</td>
<td>:content</td>
<td>:performative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recruit-all</td>
<td>:content</td>
<td>:performative</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: This is the set of performatives discussed in this document and their properties when used in conversations. The properties have the following meaning: "response required" means that the :receiver processes the performative and generates the response on its own; "response only" means that the performative can only be used in the context of responding to some other performative; "no response" means that those performatives do not require (but might allow) a response (there is also the possibility of a follow-up message); and :content refers to the type of the :content ("n/a" stands for not applicable; see Section 4.2 for an explanation). Forward, broadcast, broker-one, broker-all, recruit-one and recruit-all, do not require a response by default. Whether there is a response or a follow-up to them, depends solely on the :content, i.e., on the <performative> that appears in the :content and its properties in conversations.
<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>advertise</th>
<th>subscribe</th>
<th>standby</th>
<th>forward</th>
<th>broadcast</th>
<th>Facilitation performatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discourse</td>
<td>ask-if</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ask-all</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ask-one</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stream-all</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cos</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tell</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>untell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>deny</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>uninsert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete-one</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete-all</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>undelete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>achieve</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unachieve</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>advertise</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unadvertise</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>subscribe</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intervention and Mechanics</td>
<td>error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sorry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>standby</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ready</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>next</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rest</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>discard</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facilitation and Networking</td>
<td>register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unregister</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>forward</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broadcast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>transport-address</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broker-one</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broker-all</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recommend-one</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recommend-all</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recruit-one</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>recruit-all</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: *Advertise, subscribe, standby, forward, broadcast* and the *facilitation performatives* are the only performatives that may have a `<performatives>`, i.e., a KQML message, as :content ("facilitation performatives" refers to broker-one, broker-all, recruit-one, recruit-all, recommend-one and recommend-all). Note that the facilitation performatives allow exactly the same performatives as advertise, which makes sense since the processing of the facilitation performatives depends on advertisements. The facilitation performatives may appear in the :content of advertise messages if and only if a non-facilitator is the :sender of the advertise.
<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>All agents</th>
<th>Facilitators only</th>
<th>Only if advertised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discourse</td>
<td>ask-if</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ask-all</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ask-one</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>stream-all</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>eos</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tell</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>until</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>deny</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>insert</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>uninsert</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete-one</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete-all</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>undelete</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>achieve</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unachieve</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>advertise</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unadvertise</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>subscribe</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intervention and Mechanics</td>
<td>error</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sorry</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>standby</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ready</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>next</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rest</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>discard</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facilitation and Networking</td>
<td>register</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unregister</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>forward</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broadcast</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>transport-address</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>broker-one</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>broker-all</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recommend-one</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recommend-all</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recruit-one</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>recruit-all</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

This performative only serves the purpose of marking the end-of-stream of the multi-response to a stream-all (see Figure 3).
Agent A sends a message to agent B

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

and agent B replies with the following KQML message

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

and later agent B sends

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

and finally concludes the response with

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

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

Figure 3: A stream-all performative and the appropriate responses.
(tell
   :sender <word>
   :receiver <word>
   :in-reply-to <word>
   :reply-with <word>
   :language <word>
   :ontology <word>
   :content <expression>)

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

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

This performative indicates that the :content expression is not true of the sender, i.e., it is not part of the sender’s VKB. This does not necessarily mean that the expression’s negation is true of the sender. In other words, untell<expression> is not the same as tell-<expression>.

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

This performative indicates that the negation of the :content is true of the sender, i.e., it is in the sender’s VKB. In other words, deny<expression> is the same as tell-<expression>.

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

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

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

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

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

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

(delete-one
  :sender <word>
  :receiver <word>
  * :in-reply-to <word>
  :reply-with <word>
  :language <word>
  :ontology <word>
  :content <expression>)
Agent A sends the following performative to agent B

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

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

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

and some time later B sends the following message to A

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

which is followed a bit later by

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

Figure 4: An `insert` performative following a related `advertise`, and an example of a proper `uninsert`. Note that `reply - with_insert` is not preset by the `sender` of the `advertise`. 
This performative is a request to delete one sentence from the receiver’s KB. The sentence to be deleted is the one that would have been the :content of the response if an identical ask-one KQML message had been sent and a tell performative had been used in the response.

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

**delete-all**

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

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

**undele**

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

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

**Note:** An undele can only be used after a corresponding delete-one or delete-all. In either case, it undeltes whatever was deleted in the first place, assuming of course that the original delete action was executed successfully (no error or sorry was sent as a response).
A proposal for a new KQML specification

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

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

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

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

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

(advertise
  :sender       <word>
  :receiver     <word>
  :reply-with   <word>
  :language     <word>
  :ontology     <word>
  :content      (performative_name
                            :sender       <word>
                            :receiver     <word>
                            :in-reply-to  <word>
Agent A sends the following performative to agent B (the :in-reply-to value suggests that B has sent an advertise for such an achieve message), requesting that B set a new value for the motor torque of motor1

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

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

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

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

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

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

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

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

Figure 5: An achieve performative and the appropriate response, later followed by an unachieve request.
This performative indicates that the :sender commits to process the whole embedded message if the sender\textsubscript{advertise} receives it (presumably from receiver\textsubscript{advertise} in the future). The subsequent KQML message ought to be identical to whatever the content\textsubscript{advertise} is, except for the :reply-with value that is going to be set by the :receiver of the advertise. There are constraints that apply to such a message:

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

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

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

(unadvertise
 :sender <word>
 :receiver <word>
 :reply-with <word>
 :language <word>
 :ontology <word>
 :content (performative\_name
 :sender <word>
 :receiver <word>
 :in-reply-to <word>
 :language <word>
 :ontology <word>
 :content <expression> )
)
Agent A sends the following performative to agent B

**(advertise)**

:sender A
:receiver B
:reply-with id1
:language KQML
:ontology kqml-ontology
:content (ask-if

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

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

**(ask-if)**

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

and agent A responds accordingly, as committed to do

**(tell)**

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

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

---

Figure 6: An example of an **advertise** and appropriate follow-ups to that.
This performative essentially cancels an advertise. Its :content has to be the same with the :content of the advertise that it cancels.

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

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

NOTE: The performative_name in the content subscribe might be any of the performatives that require a response (see Table 3).
(advertise
  :sender B
  :receiver A
  :reply-with id0
  :language KQML
  :ontology kqml-ontology
  :content (subscribe
    :sender A
    :receiver B
    :in-reply-to id0
    :language KQML
    :ontology kqml-ontology
    :content (ask-all
      :sender A
      :receiver B
      :in-reply-to id0
      :language Prolog
      :ontology foo
      :content "bar(X,Y)" )))

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

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

4.2 Intervention and mechanics of conversation performatives

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

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

This performative suggests that the :sender received a message, indicated by the value of :in-reply-to, that it does not comprehend. The cause for an error might be: 1) syntactically ill-formed message, 2) the message has wrong performative parameter values, or 3) it does not comply with the conversation protocols. This performative can appear as a response to any performative, if necessary. See Figure 9 for examples of cases that may lead to an error performative being sent.
Agent A sends to agent B the following KQML message, whose :in-reply-to tag suggests that is a follow-up to an advertise (see Figure 7 for this advertise; it is an example of a really long KQML message)

\[
\text{(subscribe)}
\begin{align*}
\text{:sender} & \quad A \\
\text{:receiver} & \quad B \\
\text{:in-reply-to} & \quad \text{id0} \\
\text{:reply-with} & \quad \text{id1} \\
\text{:language} & \quad \text{KQML} \\
\text{:ontology} & \quad \text{kqml-ontology} \\
\text{:content} & \quad \text{(ask-all)}
\end{align*}
\begin{align*}
\text{:sender} & \quad A \\
\text{:receiver} & \quad B \\
\text{:in-reply-to} & \quad \text{id0} \\
\text{:reply-with} & \quad \text{id2} \\
\text{:language} & \quad \text{Prolog} \\
\text{:ontology} & \quad \text{foo} \\
\text{:content} & \quad "\text{bar}(X,Y)" \ )
\end{align*}
\]

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

\[
\text{(tell)}
\begin{align*}
\text{:sender} & \quad B \\
\text{:receiver} & \quad A \\
\text{:in-reply-to} & \quad \text{id2} \\
\text{:reply-with} & \quad \text{id3} \\
\text{:language} & \quad \text{Prolog} \\
\text{:ontology} & \quad \text{foo} \\
\text{:content} & \quad "\text{[bar}(a,b)\text{,bar}(a,c)]" \ )
\end{align*}
\]

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

\[
\text{(tell)}
\begin{align*}
\text{:sender} & \quad B \\
\text{:receiver} & \quad A \\
\text{:in-reply-to} & \quad \text{id2} \\
\text{:reply-with} & \quad \text{id4} \\
\text{:language} & \quad \text{Prolog} \\
\text{:ontology} & \quad \text{foo} \\
\text{:content} & \quad "\text{[bar}(a,b)]" \ )
\end{align*}
\]

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

Figure 8: A subscribe request and appropriate responses.
(sorry
    :sender   <word>
    :receiver <word>
    :in-reply-to <word>
    :reply-with <word>)

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

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

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

NOTE: In short, standby transfers control of the timing of the responses to the :sender of the original query, thus reversing the default protocol, according to which the :receiver delivers its responses at will.
See Figure 10 for an example of an exchange that involves the *standby* performative.

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

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

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

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

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

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

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

**Example 1**

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

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

**Example 2**

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

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

**Example 3**

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

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

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

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

Figure 9: Examples of the three situations that may result in an *error*. 
Agent A sends a message identical to the *stream-all* of Figure 3 but this time a *standby* is used.

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

and agent B this time responds with

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

Then, agent A requests the first response with

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

and finally A delivers

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

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

Figure 10: The exchange of Figure 3 when *standby* is used.
**Scenario 1:** Agent A requests the second response and B delivers it

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

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

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

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

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

in which case the exchange ends with B delivering

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

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

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

Figure 11: The possible scenarios that the exchange of Figure 10 might continue with (Figure 10 shows the exchange of Figure 3 when *standby* is used).
This performative indicates to the :receiver that the :sender is not interested in any more responses (presumably to a multi-response performative). See Figure 10 and Figure 11 for an example of an exchange that involves the discard performative.

NOTE: Performatives that may result to a multi-response are: stream-all, subscribe, recommend-all.
4.3 Networking and Facilitation performatives

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

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

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

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

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

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

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

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

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

- from_forward = sender_{performative_name}
- to_forward = receiver_{performative_name}

See Figure 12 and Figure 13 for an example of an exchange that involves the forward performative.
Note: The :in-reply-to parameter for forward is optional and as far as we know only makes sense in the context of responding to recommend-one, recommend-all, broker-one and broker-all in which case the forward is a direct response to the <performative>. In the case of forward being used to respond to broker-one and broker-all, the :sender value of the embedded performative is omitted.

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

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

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

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

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

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

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

See Figure 13 for the continuation of this exchange.

Figure 12: A conversation involving the forward performative. See Figure 13, also.
Continuing the exchange that is shown in Figure 12, agent B sends to agent D the following forward message.

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

There are two possible scenarios for D upon receiving this last message.

**Scenario 1:** D can deliver directly to C, i.e., D knows of C even though C does not know of D. In this case C sends the following message

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

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

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

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

Figure 13: The rest of the exchange of Figure 12.
The constraint is that **performative name** can be one of the performatives that can be used with *advertise* (see page 19). This is a request to find an agent that *can* and *will* process the **content**, *(i.e., an agent that has sent an *advertise* with such a **content**) in the name of the receiver of the *broker-one* *(so all responses from the third party will be directed to the broker, i.e., the **receiver**)*. After receiving the response, the broker will send it to the **sender** of the *broker-one*, wrapped in a *forward* originating from the broker-ed agent. See Figure 14 for an example of an exchange that involves the *broker-one* performative.

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

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

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

```
(recommend-one
  :sender   <word>
  :receiver <word>
*  :in-reply-to <word>
  :reply-with <word>
  :language <word>
  :ontology <word>
  :content  (performative_name
              :sender   <word>
              :language <word>
              :ontology <word>
              :content  <expression> ))
```
Agent facilitator has received an advertise message from agent A, identical to the first message in Figure 6, except for receiver advertise = facilitator and sender ask-if = facilitator). Later, agent C sends the following message to the facilitator

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

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

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

Agent A responds with the following message

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

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

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

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

Figure 14: An example of a broker-one performative and the follow-up
The constraint is that `performative_name` be one of the performatives that can be used in `advertise` (see page 19). This is a request to suggest an agent that can process the `:content` (again, as is the case with `broker-one`, use is made of the `advertise` messages that the `receiver_{recommend-one}` has received). Since more than just an agent name is needed in order for `sender_{recommend-one}` to be able to contact this agent, the appropriate response of `receiver_{recommend-one}` will be to forward the `advertise` message that satisfies the request. See Figure 15 for an example of an exchange that involves the `recommend-one` performative.

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

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

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

The constraint is that `performative_name` can be one of the performatives that can be used in `advertise` (see page 19). This performative is like a `broker-one` but responses will be directed back to the issuer of the `recruit-one`. In effect, `recruit-one` is equivalent to
Agent *facilitator* has received an *advertise* message from agent A, identical to the first message in Figure 6 (except $receiver_{advertise} = facilitator$ and $sender_{ask-if} = facilitator$). Later, agent C sends the following message to the *facilitator*

\[
\text{(recommend-one}
\begin{array}{ll}
\text{:sender} & C \\
\text{:receiver} & facilitator \\
\text{:reply-with} & id3 \\
\text{:language} & \text{KQML} \\
\text{:ontology} & \text{kqml-ontology} \\
\text{:content} & (\text{ask-if}
\end{array}
\begin{array}{ll}
\text{:sender} & C \\
\text{:language} & \text{Prolog} \\
\text{:ontology} & \text{foo} \\
\text{:content} & "\text{bar(X,Y)}" )
\end{array})
\]

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

\[
\text{(forward}
\begin{array}{ll}
\text{:from} & A \\
\text{:to} & C \\
\text{:sender} & facilitator \\
\text{:receiver} & C \\
\text{:in-reply-to} & id3 \\
\text{:reply-with} & id5 \\
\text{:language} & \text{KQML} \\
\text{:ontology} & \text{kqml-ontology} \\
\text{:content} & (advertise
\end{array}
\begin{array}{ll}
\text{:sender} & A \\
\text{:receiver} & C \\
\text{:reply-with} & id1 \\
\text{:language} & \text{KQML} \\
\text{:ontology} & \text{kqml-ontology} \\
\text{:content} & (\text{ask-if}
\end{array}
\begin{array}{ll}
\text{:sender} & C \\
\text{:receiver} & A \\
\text{:in-reply-to} & id1 \\
\text{:language} & \text{Prolog} \\
\text{:ontology} & \text{foo} \\
\text{:content} & "\text{bar(X,Y)}" ))
\end{array})
\]

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

Figure 15: An example of a *recommend-one* and a response to it.
(forward
:from <word>
:to <word>
:sender <word>
:receiver <word>
*:in-reply-to <word>
:reply-with <word>
:language <word>
:ontology <word>
:content (performative_name
  :sender <word>
  :receiver <word>
  :in-reply-to <word>
  :reply-with <word>
  :language <word>
  :ontology <word>
  :content <expression>))

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

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

The constraint is that performative_name can be one of the performatives that can be used in advertise (see page 19). This performative is like a broker-all but responses will be directed to the issuer of the recruit-all. In effect broker-all is equivalent to a series of forward messages, like those mentioned in the description of recruit-one.
Agent facilitator has received an advertise message from agent A, identical to the first message in Figure 6 (except for receiver<sub>advertise</sub> = facilitator and sender<sub>ask-if</sub> = facilitator). Later, agent C sends the following message to the facilitator

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

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

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

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

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

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

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

- In each domain of KQML-speaking agents there is at least one agent with a special status called *facilitator* that can always handle the networking and facilitation performatives. Agents advertise to their facilitator, *i.e.*, they send *advertise* messages to their facilitators, thus announcing the messages that they are committed to accepting and properly processing. Advertising to a facilitator is like advertising to the community (either of their own domain or of some other domain). Agents can still advertise on a one-to-one basis, if they so wish, and such advertisements do not commit them to processing messages from agents other than the :**receiver** of the *advertise*. Actually, such advertisements will never be shared with other agents, because of the “personal” nature of the *advertisements*, *i.e.*, they are addressed to particular agents and only *facilitators* can supersede that; see Table 5, also. Agents can use their facilitator either

  - to have their queries properly dispatched to other agents, using *recruit-one*, *recruit-all*, *broker-one* or *broker-all*, or
  - to send a *recommend-one* or a *recommend-all* to get the relevant *advertise* messages and directly contact agent(s) that may process their queries.

- Agents can access agents in other domains either through their facilitator, or directly. This implies that a smart facilitator may be built in such a way that whenever it cannot find a useful, relevant *advertise* from an agent in its domain, it may query another facilitator, in some other domain. Such an action initiates a sub-dialogue with another facilitator in order to serve the original query. Elaborate protocols of this kind are examples of conversations (interactions) that be built on top of the conversation policies presented in [3]

- Facilitators may request the services of other facilitators in the same way that regular agents may request the services of their facilitator. Facilitators do not *advertise*, not even to other facilitators. The model we imply is one where regular agents *advertise* their services to their facilitators and thus facilitators become providers of *query-processing* information about the agents in their domain; such information can then be accessed by any agent (regular or facilitator), using the *facilitation performatives*.

- We use the term *facilitator* to refer to all kinds of special services that may be provided by *specialized* agents, such as Agent Name Servers (ANS), proxy agents, or brokers ([2]).

References
