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

List of Tables

List of Figures

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>   ::= '<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>
```

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

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

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 the 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 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 $\texttt{parameter}_{performative}$ to refer to the value of a particular performative parameter, *i.e.*, $\texttt{sender}_{advertise}$ 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 `:in-reply-to` suggests that the *ask-all* follows a relevant *advertise* message.

```
(ask-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),bar(c,d)]")
```

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).[1]

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

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*.

```
(ask-one
```

[1]From now on we will use "VKB" to refer to either "exists in the KB" or "provable."

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

Table 2: Summary of reserved performatives for `:sender` S and `:receiver` R.

Category	*Name*	*Response Required*	*Response Only*	*No Response*	`:content`
Discourse	ask-if	X			`<expression>`
	ask-all	X			`<expression>`
	ask-one	X			`<expression>`
	stream-all	X			`<expression>`
	eos		X		empty
	tell		X		`<expression>`
	untell		X		`<expression>`
	deny		X		`<expression>`
	insert			X	`<expression>`
	uninsert			X	`<expression>`
	delete-one			X	`<expression>`
	delete-all			X	`<expression>`
	undelete			X	`<expression>`
	achieve			X	`<expression>`
	unachieve			X	`<expression>`
	advertise			X	`<performative>`
	unadvertise			X	`<performative>`
	subscribe	X			`<performative>`
Intervention and Mechanics	error		X		empty
	sorry		X		empty
	standby	n/a	n/a	n/a	`<performative>`
	ready	n/a	n/a	n/a	empty
	next	n/a	n/a	n/a	empty
	rest	n/a	n/a	n/a	empty
	discard	n/a	n/a	n/a	empty
Facilitation and Networking	register			X	`<expression>`
	unregister			X	empty
	forward			`:content`	`<performative>`
	broadcast			`:content`	`<performative>`
	transport-address			X	`<expression>`
	broker-one			`:content`	`<performative>`
	broker-all			`:content`	`<performative>`
	recommend-one	X			`<performative>`
	recommend-all	X			`<performative>`
	recruit-one			`:content`	`<performative>`
	recruit-all			`:content`	`<performative>`

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.

Category	*Name*	*advertise*	*subscribe*	*standby*	*forward broadcast*	Facilitation *performatives*
Discourse	ask-if	X	X		X	X
	ask-all	X	X		X	X
	ask-one	X	X	X	X	X
	stream-all	X	X	X	X	X
	eos			X	X	
	tell			X	X	
	untell				X	
	deny				X	
	insert	X			X	X
	uninsert				X	
	delete-one	X			X	X
	delete-all	X			X	X
	undelete				X	
	achieve	X			X	X
	unachieve				X	
	advertise				X	
	unadvertise				X	
	subscribe	X		X	X	X
Intervention and Mechanics	error				X	
	sorry				X	
	standby				X	
	ready				X	
	next				X	
	rest				X	
	discard				X	
Facilitation and Networking	register					
	unregister					
	forward					
	broadcast					
	transport-address					
	broker-one				X	
	broker-all				X	
	recommend-one		X	X	X	
	recommend-all		X	X	X	
	recruit-one				X	
	recruit-all				X	

Table 4: *Advertise, subscribe, standby, forward, broadcast* and the *facilitation performatives* are the only performatives that may have a `<performative>`, *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*.

Category	*Name*	*All agents*	*Facilitators only*	*Only if advertised*
Discourse	ask-if	X		
	ask-all	X		
	ask-one	X		
	stream-all	X		
	eos	X		
	tell	X		
	untell	X		
	deny	X		
	insert	X		
	uninsert	X		
	delete-one	X		
	delete-all	X		
	undelete	X		
	achieve	X		
	unachieve	X		
	advertise	X		
	unadvertise	X		
	subscribe	X		
Intervention and Mechanics	error	X		
	sorry	X		
	standby	X		
	ready	X		
	next	X		
	rest	X		
	discard	X		
Facilitation and Networking	register		X	
	unregister		X	
	forward	X		
	broadcast	X		
	transport-address		X	
	broker-one		X	X
	broker-all		X	X
	recommend-one		X	X
	recommend-all		X	X
	recruit-one		X	X
	recruit-all		X	X

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 `<performative>`. A subtle distinction has to be drawn between an agent's ability to process a *performative* in principle and to process a `<performative>`, *i.e.*, a KQML message of that *performative* 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`.

```
        :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 in 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, $untell_{\texttt{<expression>}}$ is *not* the same as $tell_{\neg\texttt{<expression>}}$.

```
(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, $deny_{\texttt{<expression>}}$ is the same as $tell_{\neg\texttt{<expression>}}$.

> 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 $\texttt{reply}-\texttt{with}_{insert}$ 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.

```
(undelete
        :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 *undelete* can only be used after a corresponding *delete-one* or *delete-all*. In either case, it *undeletes* 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).

```
(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 sent 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.

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

This performative indicates that the `:sender` commits to process the whole embedded message if the $\texttt{sender}_{advertise}$ receives it (presumably from $\texttt{receiver}_{advertise}$ in the future). The subsequent KQML message ought to be identical to whatever the $\texttt{content}_{advertise}$ 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_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).
- $\texttt{sender}_{advertise} = \texttt{receiver}_{performative\_name}$
- $\texttt{receiver}_{advertise} = \texttt{sender}_{performative\_name}$
- $\texttt{reply} - \texttt{with}_{advertise} = \texttt{in} - \texttt{reply} - \texttt{to}_{performative\_name}$

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 $\texttt{content}_{advertise}$ from *any* agent and not only from $\texttt{receiver}_{advertise}$. For all practical purposes, an *advertise* to a *facilitator* is an *advertise* to the community. Since in order for the $\texttt{sender}_{advertise}$ to process such a message, the proper value for the $\texttt{in} - \texttt{reply} - \texttt{to}_{performative\_name}$ is needed, the $\texttt{sender}_{advertise}$ can rest assured that such knowledge was acquired only through the facilitator that was the $\texttt{receiver}_{advertise}$.

```
(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 $\texttt{reply}-\texttt{with}_{ask-if}$ 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 $\texttt{sender}_{subscribe}$. 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 $\texttt{content}_{subscribe}$ 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 $\mathtt{in-reply-to}_{advertise}$ because *advertise* messages are starting points for conversations, and there is no $\mathtt{reply-with}_{subscribe}$ 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)

```
(subscribe
              :sender       A
              :receiver     B
              :in-reply-to  id0
              :reply-with   id1
              :language     KQML
              :ontology     kqml-ontology
              :content      (ask-all
                                         :sender       A
                                         :receiver     B
                                         :in-reply-to  id0
                                         :reply-with   id2
                                         :language     Prolog
                                         :ontology     foo
                                         :content      "bar(X,Y)" ))
```

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

```
(tell
              :sender       B
              :receiver     A
              :in-reply-to  id2
              :reply-with   id3
              :language     Prolog
              :ontology     foo
              :content      "[bar(a,b),bar(a,c)]" )
```

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

```
(tell
              :sender       B
              :receiver     A
              :in-reply-to  id2
              :reply-with   id4
              :language     Prolog
              :ontology     foo
              :content      "[bar(a,b)]" )
```

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 sent 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 $\texttt{receiver}_{standby}$ to handle the embedded performative as it would normally do, *but* in addition, the `:receiver` should inform the $\texttt{sender}_{standby}$ 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 $\texttt{receiver}_{standby}$ (which is also the $\texttt{sender}_{ready}$) 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

```
(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

```
(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

```
(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 $\texttt{content}_{ask-all}$ 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

```
(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

```
(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

```
(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

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

in which case the exchange ends with B delivering

```
(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

```
(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).

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

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 $\texttt{sender}_{unregister}$). 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).

```
(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:

- $\texttt{from}_{forward} = \texttt{sender}_{performative\_name}$
- $\texttt{to}_{forward} = \texttt{receiver}_{performative\_name}$

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 $\texttt{content}_{broadcast}$ 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 $\texttt{to}_{forward}$, 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 $\rightarrow$ B $\rightarrow$ A $\rightarrow$ 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 $\texttt{receiver}_{broker-one}$). After receiving the response, the broker will sent 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 $\texttt{receiver}_{advertise} = \mathit{facilitator}$ and $\texttt{sender}_{ask-if} = \mathit{facilitator}$). 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 $\texttt{receiver}_{advertise} = facilitator$ and $\texttt{sender}_{ask-if} = facilitator$). Later, agent C sends the following message to the *facilitator*

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

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

```
(forward
     :from         A
     :to           C
     :sender       facilitator
     :receiver     C
     :in-reply-to  id3
     :reply-with   id5
     :language     KQML
     :ontology     kqml-ontology
     :content      (advertise
                       :sender       A
                       :receiver     C
                       :reply-with   id1
                       :language     KQML
                       :ontology     kqml-ontology
                       :content      (ask-if
                                          :sender       C
                                          :receiver     A
                                          :in-reply-to  id1
                                          :language     Prolog
                                          :ontology     foo
                                          :content      "bar(X,Y)" )))
```

Note that $\texttt{receiver}_{advertise} = C$ instead of *facilitator* which was the value of $\texttt{receiver}_{advertise}$ in A's *advertise*. Since A's *advertise* was made to the *facilitator*, the value of the $\texttt{receiver}_{advertise}$ 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 $\texttt{to}_{forward} = \texttt{receiver}_{performative\_name} = X$, where X is to be provided by the $\texttt{receiver}_{forward}$, *i.e.*, the $\texttt{receiver}_{recruit-one}$, making use of the *advertise* performatives known to it (likewise for the $\texttt{in}-\texttt{reply}-\texttt{to}_{performative\_name}$) 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 $\texttt{receiver}_{advertise} = facilitator$ and $\texttt{sender}_{ask-if} = facilitator$). Later, agent C sends the following message to the *facilitator*

```
(recruit-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

[1] ARPA Knowledge Sharing Initiative. Specification of the KQML agent-communication language. ARPA Knowledge Sharing Initiative, External Interfaces Working Group working paper., July 1993.

[2] Tim Finin, Anupama Potluri, Chelliah Thirunavukkarasu, Don McKay, and Robin McEntire. On agent domains, agent names and proxy agents. In *CIKM Intelligent Information Agents Workshop*, Baltimore, MD, December 1995.

[3] Yannis Labrou. *Semantics for an Agent Communication Language.* PhD thesis, University of Maryland, Baltimore County, August 1996.