A semantics approach for KQML – a general purpose communication language for software agents *

Yannis Labrou
Computer Science Department
University of Maryland, Baltimore County
Baltimore MD 21228
email: jklabrou@cs.umbc.edu
voice: (410) 455-2667
fax: (410) 455-3969

Tim Finin
Computer Science Department
University of Maryland, Baltimore County
Baltimore MD 21228
email: finin@cs.umbc.edu
voice: (410) 455-3522
fax: (410) 455-3969

Abstract

We investigate the semantics for Knowledge Query Manipulation Language (KQML) and we propose a semantic framework for the language. KQML is a language and a protocol to support communication between software agents. Based on ideas from speech act theory, we propose a semantic description for KQML that associates descriptions of the cognitive states of agents with the use of the language's primitives (performatives). We use this approach to describe the semantics for the basic set of KQML performatives. We also investigate implementation issues related to our semantic approach. We suggest that KQML can offer an all purpose communication language for software agents that requires no limiting pre-commitments on the agents' structure and implementation. KQML can provide the Distributed AI, Cooperative Distributed Problem Solving and Software Agents communities with an all purpose language and environment for intelligent inter-agent communication.

1 Introduction

Let us picture a company where employees keep calendars in their personal computers. A database keeps information on the employees, such as names, offices, phone numbers. Another database may register conference rooms, with additional information regarding capacity, availability, scheduled activities and so on. One may want to build a system that can schedule group meetings in the company, according to the availability of employees and locations. The well-known approach is to built an application from scratch, so that *one* application holds all necessary information and knowledge. The alternative would be to use the existing applications. Doing that, would require: 1) the applications to be able to comprehend each other's knowledge stores, despite differences in implementation languages and knowledge representation schemes, and 2) the applications to communicate with each other and dynamically make queries, answer them, assert or remove facts from their knowledge stores, in short, to interact intelligently.

This example is an instance of the larger problem of providing for an environment where software agents may effectively communicate and exchange knowledge and information. Addressing this problem is the primary goal of the *ARPA Knowledge Sharing Effort* (KSE) [23]. KSE is an initiative to develop the technical infrastructure to support the sharing of knowledge among systems [22]. Its goal is to develop new systems by selecting components from libraries of reusable modules and assembling them together. One of the key areas identified by KSE was that of protocols for communication between separate knowledge-based modules, as well as between knowledge-based systems and databases. The result was *Knowledge Query Manipulation Language (KQML)* (see [1, 2, 14] for documentation on KQML) a message format and a message-handling protocol to support run-time knowledge sharing and interaction among agents.

Interaction is more than an exchange of messages. Issues associated with it, are: *models of agents* (beliefs, goals, representation and reasoning), *interaction protocols* (an interaction regime that guides the agents) and *interaction languages* (languages that introduce standard message types that all agents interpret identically). KQML is intended to be a *universal* interaction language, that supports communication through explicit linguistic actions. Our focus in this paper is the formal description of the semantics of the language. Although the language is partly designed and in use, it lacks a formal semantics, and its current description [2] is based on natural language descriptions of its primitives called *performatives*. We believe that a formal semantics is necessary for the unambiguous definition of the language, and its appropriate use. Furthermore, the semantic description is related to implementation issues.

Research communities with a potential interest in such a language are those of *Distributed Artificial Intelligence* (DAI[1]) ,the subfield of AI concerned with concurrency in AI computations, *(Cooperative) Distributed Problem Solving*, that studies how a loosely coupled network of problem solvers can work together to solve problems that are beyond their individual capabilities [12], and *Multi Agent Systems*, concerned with coordinating behavior among a collection of (possibly pre-existing) autonomous intelligent agents. The rising demand for *software agents* that can interoperate [16], and for *intelligent agents* that can take advantage of the

*This work was supported in part by the Air Force Office of Scientific Research under contract F49620-92-J-0174 , and the Advanced Research Projects Agency.

[1] For an introduction to the issues that DAI is concerned with, see [4] and [15].

enormous resources of today's Internet (like Etzioni's *Internet softbots* [13]) provide a proving ground for a communication language. KQML can be used in any environment where software agents need to communicate something more than pre-defined and fixed statements of facts and provides for dynamic run-time interaction, so that intelligent agents can combine their efforts, or make use of other agents' abilities, in order to achieve their goals.

In the remainder of this paper we will begin by providing a brief introduction to speech act theory which underlies our approach to defining the semantics of KQML. We will then associate KQML messages with speech acts and present a general semantic framework for KQML. Following this framework, we will give the semantics for a small set of KQML performatives. In the final two sections of the paper we discuss the impact of our analysis on some software implementation issues and discuss the kinds of applications which are appropriate for KQML.

2 Speech act theory and speech act semantics

Speech act theory is a high-level theoretical framework developed by philosophers and linguists to account for human communication. It has been extensively used, formalized and extended within the fields of Computational Linguistics and AI as a general model of communication between arbitrary agents. As such, we believe that speech act theory can provide us with a framework for the semantics of KQML, a language focused on the communication between software agents. Speech act theory is primarily concerned with the role of language as action. The following three distinct actions can be identified in a speech act: (1) a *locution*, i.e., the actual physical utterance (with a certain context and reference), (2) an *illocution*, i.e, the conveying of the speakers intentions to the hearer, and (3) the *perlocutions*, i.e., actions that occur as a result of the illocution. For example, "I order you to shut the door" is a locution, its utterance is the illocution of a command to shut the door and the perlocution may be (if all goes well) that the hearer shuts the door. An illocution is usually considered to have two parts: an *illocutionary force* and a *proposition*. The illocutionary force classifies speech acts into the following classes[2]: 1) *assertives*, that are statements of facts, 2) *directives*, that are commands, requests or suggestions, 3) *commisives*, e.g., promises, that commit the speaker to a course of action, 4) *declaratives*, that entail the occurrence of an action in themselves[3], and 5) *expressives*, that express feelings and attitudes.

There is no consensus in the literature regarding the semantic approaches for speech acts but no matter what one may consider as speech act semantics, it is necessary to make reference to the cognitive states of the agents that use them. After all, speech acts are supposed to be the result of agents' efforts to act upon the world and/or other agents. The representation of and reasoning about the states of agents and the world and how agents' actions affect them is a prerequisite for any semantic approach. There is a plethora of approaches regarding the abstractions (models) used for capturing and describing such states, depending on one's motivations. They range from informal references to propositional attitudes, like "believe" or "want", as in Searle's early work [25] where speech acts are used in the context of the investigation of *reference* and other *Philosophy of Language* issues, to strict formalisms, as in the work of Cohen and Levesque [6, 8, 7] that define a formal model of the cognitive state of an agent and then use it to interpret speech acts as actions that are derived, guided and controlled in the context of the cognitive states of the related agents. Campbell [5] uses predicates (that stand for epistemic operators), and propositions to describe mental states associated with specific speech acts (like warning or bargaining). Cohen and Perrault in their *plan-based theory of speech acts* [9] use a *believe* modal operator based on Hintikka's ideas about propositional attitudes, knowledge and belief [20]. Singh is interested in modelling agents in terms of beliefs and intentions [26] and uses this description to provide a semantic approach for speech acts [27], enhancing the usual model-theoretic framework with modal operators for the primitive concepts of *intention* and *know-how*. The common denominator of most of the formal semantic approaches is the *possible-world model* that has an axiomatization in terms of modal logic (for an introduction to the possible worlds model and the issues related to it see [21]).

We adopt Searle's description (approach) for speech acts [25, 24]. A speech act may be described as $F(P)$ where F is the illocutionary force indicator and P is the propositional content of the illocutionary act[4]. Searle suggests the following seven components of the illocutionary force:

1. The *illocutionary point* is a fundamental primitive notion. The illocutionary points are: *assertive*, *directive*, *commisive*, *declarative*, and *expressive*. The illocutionary point of a type of illocutionary act is achieved if the act is successful. The illocutionary point of a promise to do act P (commisive), is for the speaker to commit himself to doing P and the illocutionary act will be successful if the promise is to be kept in the future.

2. The *degree of strength* of the illocutionary point can distinguish between "shut the door!" and "could you please close the door?" that are both directives, but the first is a command and the second is a plea.

3. The *mode of achievement* suggests the special ways or set of conditions under which the illocutionary point has to be achieved in the performance of the speech act. A command may require a position of authority on behalf of the speaker; use of this authority may be necessary in issuing the utterance and eventually achieving the illocutionary point.

4. The *propositional content conditions* impose what can be in the propositional content P for a specific force F. For example, a speaker can not promise that a third agent will do something.

5. The *preparatory conditions* are conditions that should hold for the successful performance of an illocutionary act. In the case of a promise, such conditions might be that whatever was promised is in the hearer interest and the hearer in fact wanted him to issue the promise.

6. The *sincerity conditions* relate to the psychological (or cognitive) state of the agent. Agents have beliefs, intentions and desires. The propositional content of the illocutionary act should be identical to the propositional content of their psychological state.

[2] variations of this classification appear also in the literature

[3] as in "I name this ship the *Titanic*"

[4] The truth might be a little more complicated because P can by a proposition plus syntactic features and a context for the utterance.

7. Finally, the *degree of strength of the sincerity conditions* suggests the existence of a degree of strength in the expression of the psychological state of the speaker. "Requesting" and "begging", do not suggest the same level of desire for something to occur.

3 KQML and speech act theory, as a context for its semantics

KQML is intended as a general purpose communication language for the exchange of information and knowledge between software agents. Here is an example of a KQML message:

```
(tell       :language      prolog
            :ontology      Genealogy
            :in-reply-to   q1
            :sender        Gen-1
            :receiver      Gen-DB
            :content       ``father(John,Alice)'')
```

In KQML terminology, "tell" is a *performative*[5] (see Table 1 for more KQML performatives). Performatives explicitly suggest the illocutionary force. The value of the `:content` slot is an expression in some "computer interpreted" language[6], in other words it is the propositional content of the illocutionary act (technically, the illocutionary act is the "delivery" of a KQML message). The other parameters (*keywords*), introduce values that provide a context for the interpretation of the propositional content and at the same time hold information to facilitate the processing of the message. In this example, "Gen-1" is stating to "Gen-DB" (these are symbolic names for applications), in *Prolog*, that "father(John,Alice)". This is a response to the KQML message (illocutionary act) identified by "q1". The ontology[7] named "Genealogy" may provide additional information regarding the interpretation of the content.

We will use the term semantics to refer to: 1) everything that provides for an unambiguous interpretation of the performative, viewed as an illocutionary force indicator, 2) the perlocutionary effects, i.e, how agents' states change after sending or receiving a KQML message, and 3) criteria that suggest when the illocutionary point of the performative is satisfied.

Searle broke down the illocutionary force into seven components (presented in Section 2). Next, we examine those components that are of interest to us, and how they relate to our effort to provide meaning to performatives. The performative's *illocutionary point* and *degree of strength* are axiomatically defined by the designers (in our current analysis we ignore the degree of strength). Table 1 shows the illocutionary points for the performatives of this presentation. The *sincerity conditions* and their *degree of strength* are of no immediate interest, because we assume that all agents are sincere to the best of their ability. The *propositional content conditions* assure that agents do not make promises about other agents, they do not respond to queries not directed to them, etc. They are enforced by the *conversation policies* (more about them in the Section 6.1) and the application programmer[8]. The *mode of achievement* refers to establishing certain relationships between speakers and hearers that make certain illocutionary acts, meaningful. The mode of achievement is set by the "organizational" hierarchy or interaction protocol that the agents may use in their interaction. In Contract Net [28] , the fact that some agents act as managers and others as potential contractors, creates a context for the negotiation [11] , through bidding, that characterizes the protocol. The *preparatory conditions* are viewed as preconditions on the cognitive state, for an agent to use a performative.

For the *perlocutionary effects* we provide suggestions for the states of the sender, after sending a message and for the receiver, after processing it (presented as postconditions). The objective is to help with the interpretation of the performative, by suggesting the desired effects of its use, and to link (and restrict) the possible responses that will be acceptable follow-ups to the sender's action, by establishing preconditions for the possible response.

Finally, we need to know when the illocutionary point of a performative is eventually satisfied, e.g., a query is satisfied when it is answered appropriately. Other illocutionary acts are satisfied just by being uttered, such as telling (*tell*), and others, like asking (*ask-if* and other query performatives), require a further exchange of messages, i.e., a "conversation". Thus, we provide satisfiability (completion) condition, that indicate the state of affairs *after* the completion of the speech act (performative).

4 A framework for the semantics of KQML

The central idea is to formally define cognitive states for agents, use them to describe the performative, the preconditions, postconditions and satisfiability conditions, mentioned before, and associate those states with the use of the performatives. We use expressions in *First Order Predicate Calculus* (FOPC), to do that. In these expressions we use operators that have a reserved meaning (the operators will be identified by predicates). The use of such operators, to describe mental states of agents that use speech acts, can be found in approaches as diverse as Campbell's [5] and Singh's [27]. The operators used in this presentation are:

1. **Bel**, as in bel(A,P) which has the meaning that P is true for A. P is an expression in the native language of A's application. We will further refer to this operator in Section 7. For now, it suffices to say that P "exists" in the agent's *knowledge base* (or *virtual knowledge base*).

2. **Know**, like the following two operators, refers to the cognitive state of the agents[9]. Know(A,P) expresses a state of knowledge awareness on behalf of A, about P.

3. **Want**, as in want(A,P), to mean that agent A desires the event (or state) described by P, to occur.

4. **Intend**, as in intend(A,P), to mean that A has every intention of doing P.

---

[5] term first coined by Austin [3], to suggest that some verbs can be uttered so that they perform some action (later, it was decided that *all* verbs may be considered as performatives)

[6] In the full version of KQML (not presented here), the content may also be a KQML message itself.

[7] An ontology is a repository of semantic and primarily pragmatic knowledge over a certain domain. Ontologies are part of the Shared and Reusable Knowledge Bases Group of the KSE.

[8] It is necessary for the programmer to guarantee that an application does not use bizarre propositional contents for a certain performative, due to their pragmatic nature. Since KQML is opaque to the content of the message, there is no way to guarantee that, for instance, an agent does not promise that "the time is 12:30PM". However, the conversation policies will ensure that if agent A poses a query to agent B, B will respond only to A and A will receive responses to this query, only from B.

[9] As such, all three could be termed as *epistemic operators*.

Name	*Illocutionary point*	*Meaning*
tell	assertive	A states to B that A believes the content to be true
deny	assertive	A states to B that A does not believe the content true
ask-if	directive	A wants to know what B believes regarding the truth status of the content
ask-all	directive	A wants to know all B's responses that would make the content true of B (the response will be a collection of expressions)
stream-all	directive	like *ask-all*, but the responses are to be delivered one by one
eos	declarative	end of a stream of responses to an earlier query
error	assertive	A states to B that B's message was not processed by A
sorry	assertive	A states to B that B's message was processed by A, but no reply can be provided

Table 1: Performatives mentioned in this presentation, for sender A and recipient B.

Roughly, *know*, *want* and *intend* stand for the psychological states of knowledge, desire and intention, respectively. Only for the *bel* predicate, it is the case that P is an expression in the agent's implementation language. For all other three operators, P is an expression that combines other operators, and stands for an event or a state of affairs. For example, it is correct to say "know(A,bel(B,foo(a,b)))" (if B "speaks" Prolog) but not "know(A,foo(a,b))". One can ask if (and how) those operators are implemented in an application. The short answer is that only the *bel* operator has to have a concrete meaning (that depends on the application language or knowledge representation language and scheme), and the others prescribe a state of affairs for the agent that is associated with the use of the language. The use of a specific performative suggests an associated state for the speaker, as in assuming when one asks X, that he wants to know X.

The semantics are implemented through the *conversation policies* to be provided by the KQML developers, and the *handler functions*, to be provided by the application programmer [10]. The conversation policies indicate what performatives can follow the utterance of a certain performative, so that agents can have meaningful conversations. The conversation policies are an integral part of the semantics and are consistent with the preconditions, postconditions and completion conditions, to be introduced for the performatives. For example, when an *ask-if* is uttered, it can only be followed (see Figure 1) by a *tell* or *deny*[11] which, in return, can only be uttered as a response to an "asking". Figure 1 gives an example of the conversation policy for the small subset of KQML performatives introduced here. It as part of an Augmented Transition Network specification, with the constraints and relating actions missing. Details about the implementation and functionality of conversation policies (along with details for the structure and construction of KQML speaking agent) can be found in Section 6. The *handler functions* are defined in order to process messages *received* by an application and should be consistent with the semantics described here. Handler functions are not application dependent, but rather language dependent[12], in the sense that all applications using the same language share the same handler functions.

[10] The software architecture of a KQML speaking agent is shown in Figure 2 and more details about it are given in Section 6.

[11] A *sorry* or an *error* may also occur.

5 Semantics for KQML performatives

The general semantic description of a KQML performative has the following six constituents:

1. A natural language description of the performative's intuitive meaning.
2. An expression in our logic that describes the illocutionary act. For all practical purposes, this is a formal representation of the natural language description.
3. Preconditions that indicate the necessary state for an agent in order to send a performative and for the receiver to accept it and process it.
4. Postconditions that describe the states of agents after the utterance of a performative (for the sender) and after the receipt (but before a counter utterance) of a message (by the receiver)[13].
5. Completion conditions for the sender that indicate the final state of the sender, after possibly a conversation has taken place and the intention suggested by the performative that started the conversation, has been fulfilled.
6. Any natural language comments that we might find suitable to enhance the understanding of the performative.

If there are non-null preconditions for the receiver, this will mean that the performative can only be some-kind of response to the use of another performative that established

[12] For an application written in Prolog, a handler function to handle *ask-if* messages, looks like this:

```
handle(ask-if,Content):-
(call(Content) ->
(reply_to_message_with(tell,Content));
(reply_to_message_with(deny,Content))).
```

where reply_to_message_with interacts with the conversation module, that implements the conversation policies, to provide the appropriate values for the other message parameters and finally deliver the response.

[13] After the receiver replies, a new *cycle* of preconditions and postconditions gets started.

those preconditions[14]. No preconditions are necessary for the receiver of a performative that starts a conversation (see Pre(B) for the query performatives, such as *ask-if, ask-all, stream-all*).

In a conversation, the postconditions for the sender of a message should be a subset of the preconditions for the receiver of the message that may follow (compare Post(A) for *ask-if* and Pre(A) for *tell*).

When no conversation is necessary after the utterance of a performative, completion (satisfiability) conditions are a subset of the postconditions. Such performatives are satisfied just by being successfully uttered and processed by the intended recipients.

In the rest of this section we give the semantic descriptions for the eight performatives in Table 1. In these descriptions A is the sender, B is the receiver and X is the propositional content. All expressions mentioned as preconditions, postconditions and completion conditions, are the minimum necessary for our specification of KQML.

- **ask-if(A,B,X)**
  1. A wants to know what B believes regarding the truth status of the content.
  2. want(A,know(A,Y)),
     where Y may be one of the following:
     bel(B,X), bel(B,NOT(X)), NOT(bel(B,X))
     (this means that Pre(A) could also be stated as:
     want(A,know(A,bel(B,X))) OR
     want(A,know(A,bel(B,NOT(X)))) OR
     want(A,know(A,NOT(bel(B,X)))))
  3. Pre(A): want(A,know(A,Y))
     (optionally, NOT(know(A,Y)) should also hold)
     Pre(B): NONE[15]
  4. Post(A): intend(A,know(A,Y))
     Post(B): know(B,want(A,know(A,Y)))
  5. Completion(A): know(A,Y)
  6. Not believing something is not necessarily the same as believing its negation, although this may be the case for certain systems.

- **ask-all(A,B,X)**
  1. A wants to know all of B's responses that make X true of B. X is an expression with variables and A wants *all* the expressions that are true for B and have values for these variables[16].
  2. want(A,know(A,Y)),
     where Y is bel(B,Y') and Y' is a finite collection of Y_1, Y_2, ... Each Y_i is an instance of X with values for the variables in X, identified by the `:aspect` parameter and each Y_i appears once in this collection (the collection might be empty).
  3. Pre(A): want(A,know(A,Y))
     (optionally, NOT(know(A,Y)) should also hold)
     Pre(B): NONE
  4. Post(A): intend(A,know(A,Y))
     Post(B): know(B,want(A,know(A,Y)))
  5. Completion(A): know(A,Y)
  6. An *ask-if* would be appropriate to ask "is it past 5 o'clock?" and an *ask-all* would be more suitable to ask "what time it is?". It is not necessary that when X has free variables, an *ask-all* should be used. An *ask-if* with content *foo(X,Y)* makes perfect sense (for PROLOG "speaking" agents), if one wants to know if there exist X such that foo(X,Y) is true. But if the same expression is used with an *ask-all*, one expects something like [foo(a,b),foo(a,c)]. The use of *ask-all* assumes that the application's language provides built-in features for collections (such as a list in our PROLOG example).

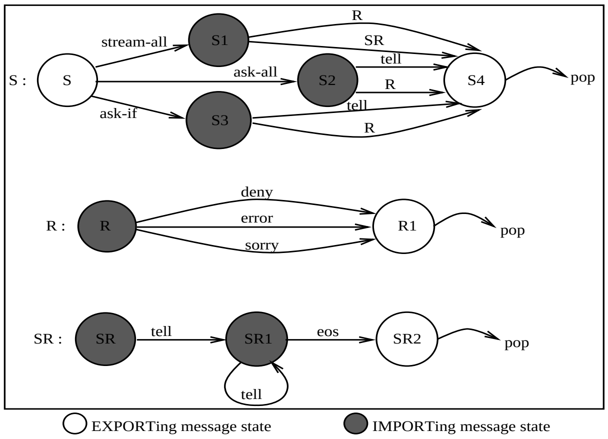

Figure 1: A simple example of an ATN to parse sequences of KQML messages.

- **stream-all(A,B,X)** Everything mentioned for *ask-all* holds for *stream-all*, too. A is interested in a series (possibly infinite) of statements of facts, as a response. The only difference is in the expected delivery format of the response. Either because the sender can not (or does not want to) process collections or due to receiver's inability to provide collections, the elements of the would be collection are to be delivered one by one (using *tell* since they are statements of facts for B). This performative also allows for responses to be delivered one at a time, as they are computed, thus permitting "pipelining" and efficient handling of very large, or even infinite, collections. The *eos* performative is to be used to mark the end of this multi–response (this is for A's benefit).

- **tell(A,B,X)**
  1. A states to be that A believes the content to be true.
  2. bel(A,X)[17]

[14] To provide an example, consider the situation that A asks B the time and B responds *12:00PM*. From our point of view, two speech acts take place (so two messages with the appropriate performatives have to be exchanged), the asking and the response to the asking. A precondition for B to respond would be that A asked him and for B that he still wants to know the time. For A to pose the question, there is a precondition that A wants to know the time (and possibly that A does not know the time already).

[15] For expository purposes we have made the simplifying assumption that agents know what other agents know, so they only ask them questions that they can answer. We have to do that for the sake of completeness of the subset we present here. In the full KQML version, there are ways for agents to learn what other agents can answer.

[16] the variables for which A wants values, are specified by the :aspect parameter in the KQML message

[17] This interprets *tell* as an assertive. If interpreted as a directive, it should be want(A,know(B,bel(A,X))).

3. Pre(A): bel(A,X) , know(A,want(B,know(B,Y))) A does not lie and B is interested in knowing. Y is any of the Y's mentioned in *ask-if, ask-all, stream-all.*
   Pre(B): intend(B,know(B,Y))
4. Post(A): know(A,know(B,bel(A,X))) (optional)
   Post(B): know(B,bel(A,X))
5. Completion(A): know(B,bel(A,X))
6. The completion condition holds, unless a *sorry* or *error* suggests B's inability to acknowledge properly the *tell.*

- **deny(A,B,X)** Everything mentioned about *tell* holds for *deny,* if `bel(A,X)` is replaced with `NOT(bel(A,X))`.

For the next two performatives, we will need three extra predicates. We consider three stages in the handling of a *received* message. First, it is physically *received* (something we implicitly assume throughout the analysis[18]), second, *processed,* in the sense that it is a valid KQML message and will be delivered to the application for processing, and third, *delivered* to the application (technically, a handler function takes over) and the application will reply to that accordingly. We will use the predicates **receive**, **process** and **respond**, for those 3 stages, respectively. The predicates refer to the stages when completed and reference of each of one of those, assumes that the prior stages have occurred. Reference to the message being handled is made through *Id* (specified in the `:reply-with` parameter), and *Id* refers to the message as a whole.

- **error(A,B,Id)**
  1. A states to B that is not going to process the KQML message identified by Id.
  2. NOT(process(A,Id))
  3. Pre(A): receive(A,Id)
     Pre(B): NONE
  4. Post(A): know(A,know(B,NOT(process(A,Id))))
     Post(B): know(B,NOT(process(A,Id)))
  5. Completion(A): know(B,NOT(process(A,Id)))
  6. An agent might respond with an *error* if either he cannot successfully parse it as a KQML message, or the message is not an acceptable one, in the context of a "conversation" between the two agents.

- **sorry(A,B,Id)**
  1. A states to B that although he processed the message, he has no response to provide.
  2. NOT(respond(A,Id))
  3. Pre(A): process(A,Id)
     Pre(B): NONE
  4. Post(A): know(A,know(B,NOT(respond(A,Id))))
     Post(B): know(A,NOT(respond(A,Id)))
  5. Completion(A): know(B,NOT(respond(A,Id)))
  6. The best analogy for understanding the performative, is what happens when you are asked the time and you do not know what time it is.

[18] Addressing the issue of agent notification for messages delivered and received, is among those considered in KQML's implementation.

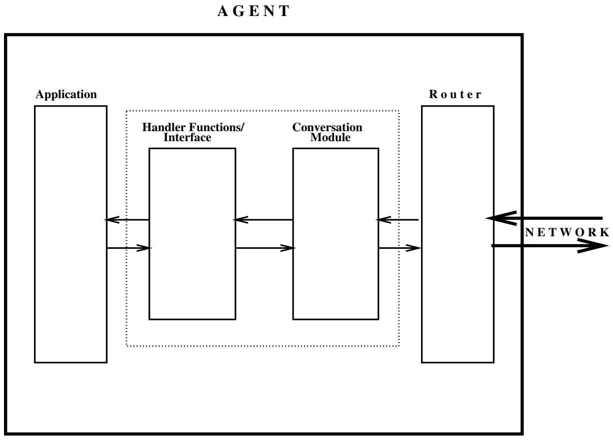

Figure 2: Logical architecture of a KQML speaking agent.

- **eos(A,B,Id)** This performative is somewhat unusual with respect to the other performatives mentioned because it is only purpose is to notify B that there are no more responses to a request for a multi-response query.

An example

Here, is an example of a conversation between agents with symbolic names Gen-DB and Gen1. Gen1 wants to know who are John's parents, and sends a *stream-all* to Gen-DB,

```
(stream-all :sender        Gen1
            :receiver      Gen-DB
            :language      Prolog
            :ontology      Genealogy
            :aspect        ``X''
            :reply-with    q1
            :content       ``parent(John,X)'')
```

and, in time, Gen-DB responds accordingly:

```
(tell       :sender        Gen-DB
            :receiver      Gen-1
            :language      Prolog
            :ontology      Genealogy
            :in-reply-to   q1
            :content       ``parent(John,Alice)'')

(tell       :sender        Gen-DB
            :receiver      Gen-1
            :language      Prolog
            :ontology      Genealogy
            :in-reply-to   q1
            :content       ``parent(John,Bob)'')

(eos        :sender        Gen-DB
            :receiver      Gen-1
            :in-reply-to   q1)
```

6 KQML semantics and architecture of KQML speaking agents

The logical architecture of a KQML speaking agent is shown in Figure 2. It is based in the KQML implementation developed at *UNISYS* [14] . We identify the following four parts:

Application. In the case that this is a non–distributed application, the application programmer has to identify the points in the program where external information is needed. At those points, queries (in the general sense) have to be delivered to other applications (*agents*) that can answer them. The problem of what to send to whom can be attacked in several ways: 1) if the query-answering capabilities of each agent are well known in advance (like in [17] and in [10], where early versions of KQML were used for inter–agent communication) the application programer encodes the information in the distributed application so that when a query has to be answered by an agent in the outside world, the application knows in advance whom to query, 2) if the application operates in an environment mostly consisting of *open systems* [19, 18] the application can ask a *facilitator*[19] to appropriately deliver its query, or, 3) the application can ask the facilitator (or other agents) to take care of appropriately delivering the query or "discuss" the matter with the facilitator or other agents, in order to deliver the query on its own, or collect information from agents and facilitators, so that it can make its own decisions regarding the delivery of its queries (such an approach is also best suited for an open systems' community). KQML provides performatives to support the implementation of all the above mentioned approaches. Only in this last case, has the application programmer to provide code in order to use the extra information regarding other agents' capabilities.

Handler functions and Interface Module. The application programmer has to provide functions (called *handler functions*) that will process the various *performatives*. For example, for the *ask-if* performative the handler function (written in the application's native language) should access the application, check the truth status of the expression for the application and accordingly convey this information to the agent that made the query. Normally either the *tell* or the *deny* performative should be used in such a case. Through them, the application can state either that the expression is true, or that it is not known to be true or that the negation of the expression is true. In order for the application programmer to provide the handler functions he has to know the exact meaning of the various KQML performatives (here on called *semantics of the performatives*) and the policies that govern their use (*conversation policies*). We further refer to the conversation policies in Section 6.1.

Conversation Module. The conversation module lies between the router and the handler functions and interface module. Every message, either received by the agent or sent to some other agent, has to go through the conversation module. This module implements the conversation policies and checks all messages in order to decide if they are allowable continuations of the agent's current conversations with other agents. Our approach regarding the implementation of this module and its role and functionality in the overall architecture of an agent, is the subject of Section 6.1. We consider this module to be a partial implementation of the semantics.

Router. The router handles all KQML messages going to and from its associated application. Each KQML speaking software agent has its own router process but all routers are identical. Routers are content independent message routers that provide the agent with a single point of contact for the rest of the network. It provides both client and server functions for the application and manages multiple simultaneous connection with other agents.

[19] Facilitators are specialized agents that are designated with the task of facilitating the communication of agents by primarily holding information regarding the query answering capabilities of the agents in their network domain.

```
EXPORT - (stream-all :sender a  :receiver  c :reply-with  id-a-1 :content expr1)  @T1
IMPORT - (ask-if     :sender b  :receiver  a :reply-with  id-b-1 :content expr2)  @T2
IMPORT - (tell       :sender c  :receiver  a :in-reply-to id-a-1 :content expr3)  @T3
EXPORT - (ask-all    :sender a  :receiver  b :reply-with  id-a-2 :content expr4)  @T4
IMPORT - (tell       :sender c  :receiver  a :in-reply-to id-a-1 :content expr5)  @T5
EXPORT - (tell       :sender a  :receiver  b :in-reply-to id-b-1 :content expr6)  @T6
IMPORT - (tell       :sender c  :receiver  a :in-reply-to id-a-1 :content expr7)  @T7
IMPORT - (tell       :sender b  :receiver  a :in-reply-to id-a-2 :content expr8)  @T8
IMPORT - (eos        :sender a  :receiver  c :in-reply-to id-a-1)                 @T9
```

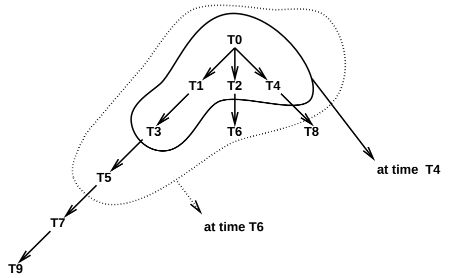

Figure 3: Sequence of imported and exported messages for agent "a".

6.1 Implementation of the conversation policies for KQML performatives

The purpose of the conversation module is to assure that the agent is involved in meaningful conversations with other agents and keep track of them, despite the possibly asynchronous behavior of the agent. The conversation module is an implementation of the conversation policies that suggest: 1) which performatives start a conversation, and 2) which performative is to be used at any given point of a conversation. Figure 3 gives an example of a series of messages sent and received by an agent name a during some time period. Between times T_1 and T_9 messages from three different conversations are handled. The conversation module should handle something like that appropriately, keeping track of all three ongoing threads. Here is the scheme we suggest for doing that:

1. When a message (either to be imported to the application or to be exported to some other agent) reaches the conversation module, the module attempts to match it against one of the ongoing conversations.

2. If the message is not an acceptable continuation of some current thread, an attempt is made to start a new thread with it[20].

3. If no new thread can start with the current message, a message with the *error* performative will be sent to the sender (if the message is to be imported) or a signal is delivered to the application (if the message is to be exported).

We obviously have to define the acceptable threads of message exchanges and provide the module with the means to test them. We view the problem as one of parsing where the

[20] Not all performatives can be starting points for new threads. In the example of Figure 3 we consider the performatives *ask-if*, *ask-all*, *stream-all* to be acceptable starting points. We believe that eventually only *advertise* performatives (that are used to make known to other agents the capabilities of an agent) should be starting points.

grammar defines the conversation policies and messages are the terminals (so any series of messages in the *same* thread, is a "sentence" to be parsed). It differs from the usual parsing paradigm, though, in that the "sentence" might well be unfinished, meaning that the thread might not be complete (see Figure 3 as of time T_4 or T_6). Figure 1 shows part of an *Augmented Transition Network (ATN)* specification that can be used to perform this task for the subset of KQML performatives of Table 1, The ATN defines the conversation policies for this subset. For illustrative reasons the states where a message is to be imported are shaded. Not presented here are the *tests* and *actions* of the ATN that handle the necessary constraints among the various fields of the messages in order to define a thread[21] (conversation). The terminals are not known in advance. As mentioned before, the terminals are KQML messages with values for all their fields. Every time that a new message is to be handled by the module, the message becomes a potential new terminal. Referring to the described, top-level procedure, this new terminal is appended to the first "sentence" (thread) and an attempt is made to successfully parse the new sentence. If this fails, the second "sentence" is tried and so one.

An implementation of the conversation policy for a considerably extended set of KQML performatives is in progress. We believe that by providing a conversation module that can cooperate with the router the agent will be able to better handle asynchronous behavior, help the agent keep track of its business and provide the means to the application programmer to build more complex schemes of inter-agent communication (protocols like the Contract Net, see [28, 11]).

7 Software agents and KQML

We argue that our semantic approach does not constrain the kinds of software agents that can use KQML. Although the propositional attitudes represented by the predicates **know**, **want**, **intend** make reference to cognitive states for the agents, the cognitive states are necessary for understanding the performatives but not for using them. If the application designer wants to build a belief model to implement those mental states on top of the application, so that the application can better support a problem solving strategy or protocol, so be it. KQML does not require the existence of such a protocol or a cognitive model. Operators like **want** and **know** are materialized by virtue of use of a performative and are implied by the use of the language, rather, than the other way round, i.e., cognitive states implying a certain use of the language.

The really interesting question is how to interpret the **bel** operator in a given computer program. It depends on what the programmer ascribes to the program. For a PROLOG application (or a logic based system in general), **bel** might stand for whatever can be proved true in the system. Similar arguments can be made for other applications that adhere to the physical symbol–system hypothesis (frames, scripts, rule–based systems, semantic nets). How about a neural net? One can still suggest an interpretation that associates input and output. The same argument can be made for devices (such as thermostats), or databases. A functional approach to provide materialization for **bel** and common sense about how it should be interpreted for a given system, will do. If not, the `:ontology` slot can solve the problem. By choosing an interpretation from a library of such, the application can make known to its conversational partner what **bel** means for it.

It is our view that a belief model or a cognitive model is not necessary for a software agent to talk KQML. It can be useful to have one, either elaborate or primitive, but nothing more that a functional interpretation of the **bel** operator is necessary, for the semantics to make sense. All that is necessary is a program and handler functions. In between these two, many things can be included. A belief space, a cognitive model, a goal space, a problem solving strategy, or various combinations of the above. But none of that is mandatory for KQML to be used. In KQML, like in human communication, the personal agendas and beliefs of the agents suggest the choice of words, but the words themselves have an accepted meaning.

8 Conclusion

We have presented an approach for the definition of the semantics of KQML. Although it is eventually the programmer that materializes the semantics through the handler functions that he writes, we have provided a framework that the programmer has to comply with. This framework is more detailed and formal than the existent so far ([2]), and will be supported by a software module (the *conversation module*) that will guide and restrict the possible uses of the languages primitives (performatives). The framework is based on *speech act theory* and primarily Searle's ideas.

We envision KQML as a general purpose communication language for software agents of all kinds. We believe that we offer an approach towards the semantics of the language that makes no commitments to application languages, agent models, programming paradigms, problem solving strategies and protocols. This approach stems from our belief that all those issues are peripheral to the communication language itself, which should be rich enough to accommodate a variety of propositional attitudes and offer enough leeway to implement all kinds of models, strategies and protocols, beneath the language. Ideally, KQML will rise to its full potential with the use of the results of the other research efforts of the KSE, because those efforts will provide the means for interagent understanding of the propositional context itself.

In the future, we intend to further apply our semantic approach to the full set of the up to date KQML performatives and refine the structure of a KQML speaking agent. All material related to KQML and the KSE can be accessed through the World Wide Web[22].

References

[1] External Interfaces Working Group ARPA Knowledge Sharing Initiative. KQML Overview. Working paper, 1992.

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

[3] J.L. Austin. *How to do things with words*. Harvard University Press, Cambridge, MA, 1962.

[4] Alan H. Bond and Les Gasser. An analysis of problems and research in DAI. In *Readings in Distributed Artificial Intelligence*, pages 3–35. Morgan Kaufman Publishers, San Mateo, California, 1988.

[21] The fields for the KQML subset presented here, are: :sender, :receiver , :reply-with and :in-reply-to.

[22] URL is http://www.cs.umbc.edu/kqml/

[5] John A. Campbell and Mark P. D'Inverno. Knowledge interchange protocols. In Y. Demazeau and J.-P. Muller, editors, *Decentralized A.I.: Proc. of the First European Workshop on Modelling*, pages 63–80. Elsevier Science Publishers B.V. /North Holland, Amsterdam, 1990.

[6] Philip R. Cohen and Hector J. Levesque. Intention = Choice + Commitment. In *Proceedings of the National Conference on Artificial Intelligence*, pages 410–415, July 1987.

[7] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[8] P.R. Cohen and H.J. Levesque. Persistence, intention, and commitment. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 33–69. MIT Press, Cambridge, MA, 1990.

[9] P.R. Cohen and C.R. Perrault. Elements of a plan–based theory of speech acts (1979). In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 169–186. Morgan Kaufman Publishers, San Mateo, CA, 1988.

[10] M. Cutkosky, E. Engelmore, R. Fikes, T. Gruber, M. Genesereth, and W. Mark. PACT: An experiment in integrating concurrent engineering systems. 1992.

[11] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 333–356, Morgan Kaufmann, 1988).

[12] E.H. Durfee, V.R. Lesser, and D.D. Corkill. Cooperative distributed problem solving. In A. Barr, P.R. Cohen, and E.A. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Vol. IV*, pages 83–147. Addison–Wesley Pub. Co., Reading, MA, 1989.

[13] Oren Etzioni and Daniel Weld. A softbot-based interface to the internet. *CACM*, 37(7):72–76, 1994.

[14] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing.* Ohmsha and IOS Press, 1994.

[15] L. Gasser. An overview of DAI. In Nicholas M. Avouris and Les Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 9–30. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[16] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *CACM*, 37(7):48–53, 1994.

[17] Mike Genesereth. Designworld. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 2,785–2,788. IEEE CS Press.

[18] Carl Hewitt. Offices are open systems. *Communications of the ACM*, 4(3):271–287, July 1986. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 321–330, Morgan Kaufmann, 1988).

[19] Carl Hewitt and Jeff Inman. DAI betwixt and between: From "intelligent agents" to open systems science. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6), December 1991. (Special Issue on Distributed AI).

[20] J. Hintikka. *Knowledge and Belief.* Cornell University Press, Ithaca, New York, 1962.

[21] Kurt Konolige. *A Deduction Model of Belief.* Pitman, London, 1986.

[22] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36 – 56, Fall 1991.

[23] R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA Knowledge Sharing Effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, San Mateo, CA, November 1992. Morgan Kaufmann.

[24] J. Searle and D. Vanderveken. *Foundations of illocutionary logic.* Cambridge University Press, Cambridge, UK, 1985.

[25] John R. Searle. *Speech Acts.* Cambridge University Press, Cambridge, UK, 1969.

[26] M.P. Singh. Towards a formal theory of communication for multiagent systems. In *Proceedings of the IJCAI'91*, 1991.

[27] M.P. Singh. A semantics for speech acts. (to appear in Annals of Mathematics and Artificial Intelligence), 1992.

[28] Reid G. Smith. The contract net protocol: High level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 357–366, Morgan Kaufmann, 1988).