An Overview of KQML: A Knowledge Query and Manipulation Language

KQML Advisory Group

with major contributions from

Hans Chalupsky
Tim Finin
Rich Fritzson
Don McKay
Stu Shapiro
Gio Wiederhold †

April 1992

Abstract

We describe a language and protocol intended to support interoperability among intelligent agents in a distributed application. Examples of applications envisioned include intelligent multi-agent design systems as well as intelligent planning, scheduling and replanning agents supporting distributed transportation planning and scheduling applications. The language, KQML for Knowledge Query and Manipulation Language, is part of a larger DARPA-sponsored Knowledge Sharing effort focused on developing techniques and tools to promote the sharing on knowledge in intelligent systems. We will define the concepts which underly KQML and attempt to specify its scope and provide a model for how it will be used.

Notice of DRAFT Status. This document presents the current draft of a specification under consideration by the DARPA Knowledge Sharing Effort. It is provided for information purposes, and should be treated as representing only the current status of discussions. It should not be interpreted as a finished product. This document should not be quoted or cited as representing the official position of DARPA, the Knowledge Sharing Effort, or any other organization.

The specifications herein are subject to change. To be placed on the distribution list for future releases of these documents, contact the authors or send electronic mail to Neches@ISI.edu.

Please send comments to Tim Finin, Computer Science, University of Maryland, Baltimore MD 21228; finin@cs.umbc.edu; 410-455-3522 or to Don McKay, Paramax Systems Corporation, PO Box 517, Paoli PA 19301; mckay@prc.unisys.com; 215-648-2256.

†This document replaces an earlier draft which was distributed without the complete list of contibutors due to an administrative oversight.

Contents

1 Introduction

This paper is an overview of the Knowledge Query and Manipulation Language (KQML) ongoing effort under the auspicies of External Interfaces Working Group of the DARPA Knowledge Sharing Effort [8]. The report is intended to motivate the need for a standard language for information exchange among a collection of interacting knowledge-based agents and give an overview for KQML.

1.1 Background and Motivation

We envision future large knowledge-based systems systems consisting of many dynamically interacting intelligent agents and intelligent components. Organizationally, they will differ depending on their layer, i.e., are they close to the user, are they deeply embedded, or do they interface with the real-world? Figure 1 shows these interfaces, and indicates the external interfaces needed.

While there are many ad hoc techniques for accomplishing the integration of intelligent agents and components, it is important that a common set of methods are adopted as early as is reasonable in order to facilitate the use of these new architectures. Among the issues which need to be dealt with are:

- What language does an agent use to formulate a query? What language is the reply made in?
- What protocol does an agent use to send a query and produce an answer?
- What is an effective way to present a communication protocol to users of knowledge-based programming disciplines such as procedural languages, backward chaining engines, forward chaining engines, classifiers, etc?
- How does an agent know which process to send queries to? How can it find out? What protocol does it use in finding out?

The integration technology we will develop will be based on KQML, Knowledge Query and Manipulation Language, a language and protocol intended to support interoperability among intelligent agents in a distributed application. KQML is part of a larger DARPA-sponsored Knowledge Sharing effort focused on developing techniques and tools to promote the sharing on knowledge in intelligent systems (Neches 1991).

The technical problem of coordinating many agents who must communicate with one another is a difficult one. In general, the agents must know:

- Which other agents to communicate with.
- How to establish a reliable communication channel with them.
- What protocol to use in the ensuing dialogue.
- What language to use to exchange information knowledge.
- What terms within the language to use to guarantee that the other agent will interpret the expressions in the same way.
- How to handle inconsistent information and the eventual mis-matches that arise from different views of the world.

These problems are compounded when we are trying to coordinate a multitude of "intelligent agents". The key aspects of the problem specifically addressed in this paper are

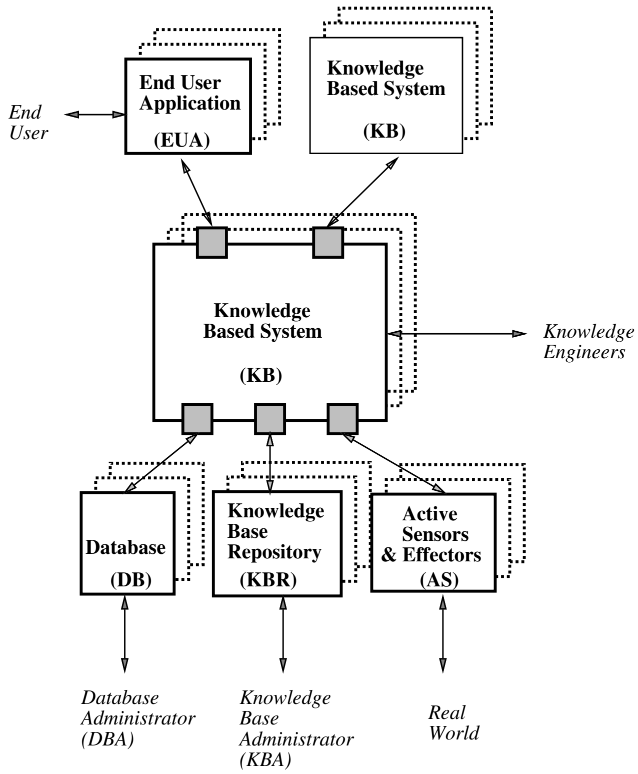

Figure 1: Major modules of a knowledge-based system. Future knowledge-based systems will be integrated in a distributed architecture in which knowledge and data will be shared among applications, knowledge-based systems, databases, knowledge repositories and sensor/effector systems. KQML will primarily support interaction with knowledge-based systems, either from applications or other knowledge-based systems.

- the description of the technical problem,
- the definition of a standard KQML for use in current and future intelligent information integration projects, and
- the design of an efficient, robust and scalable KQML implementation.

1.2 Modules in a Knowledge-Based System

To define the interfaces we first describe the different classes of intelligent components introduced in Figure 1.

End User Applications (EUAs). Here knowledge is acquired, possibly from multiple sources, fused, analyzed, and presented to an end user. Enough meta-knowledge must be made available to recognize sources, cost, expected volumes, and presentation devices. Knowledge of the source may lead to assessment of quality and completeness.

Knowledge Based Systems (KB). Here knowledge and meta-knowledge is made available to the EUAs, in response to their requests. Knowledge-based systems may monitor data resources such as databases (DBs) and active sensors (ASs) in order to acquire and monitor knowledge. Responses to EUA's may be deferred until certain conditions in source databases or active sensors are met. Knowledge engineers construct knowledge-based systems using a development interface or shell.

Knowledge Base Repositories (KBR). Associated with a knowledge-based system will be a repository for the specific knowledge base content which we call a *Knowledge Base Repository*. The knowledge base repository will assure persistency of the knowledge. Often a full history will be kept, so that older versions of a knowledge base will be accessible. Akin to the coordination and responsibility role of a database administrator, a *Knowledge Base Administrator* is charged with overseeing the design, integrity and content of a knowledge-base for use in knowledge-based system applications.

Database Systems (DBs). Here we group conventional databases and other data files. The focus of the database system is collecting and storing information persistently. Such database systems are typically maintained and updated autonomously to reflect the accurate operational status of an organization or other entity. A database administrator is in charge of the database design, integrity and content for use by other applications.

Active Sensors (AS). We must also consider direct input from real-world phenomena. Under active sensors we group all data sources that are not persistent. This means that subsequent requests are likely to provide new values, and older values will not be retrievable. An extension, not specifically considered here, are effectors that directly effect change on the real world.

1.3 The Interfaces

We can now discuss systematically the possible interfaces for these five system nodes, and indicate our focus. Since we do not distinguish direction we have 1/2(n*(n+1)= 15 potential interface types for 5 module types. In Figure 2 only one instance of each interface is shown.

This enumeration of interfaces is intended to define precisely what interfaces KQML is addressing now. The primary interest is in interfacing end user applications to knowledge-based systems and knowledge-based systems to each other. A secondary interest is the integration of knowledge and data from other sources such as databases and sensors/effectors. For the latter, KQML could serve as a wrapper supporting a set of common knowledge-based system behaviors for other components such as databases or sensors/effectors.

EUA to EUA. If an application (EUA) accepts a server role for another EUA, it must satisfy the requirements we place on a general knowledge-based system (KB) acting as a server, as defined below.

EUA to KB. The transmission of knowledge or knowledge-based derivations from knowledge-based systems acting as servers to applications is a major path and is a focus of this document.

EUA to KBR. Direct access to the representation of knowledge (KBR), without the mediation of a (KB) can provide history, structure, and other information about the corresponding KB. However, uninterpreted knowledge may be risky to use; the expected user, the knowledge-based administrator, of the interface is expected to be experienced in the domain and the interface.

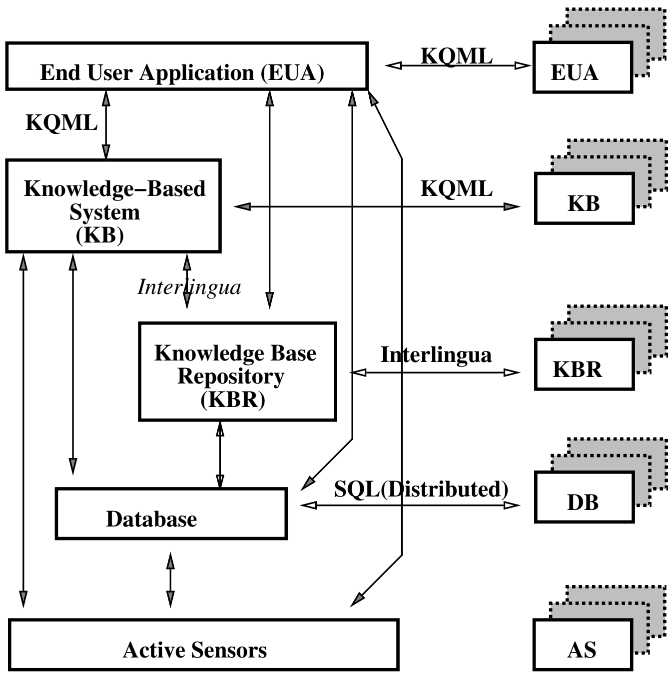

Figure 2: Internal interfaces among major modules: Interfaces which are labeled with KQML are the primary focus of this effort. KQML supports the communication interaction among components independent of the content languae used to exchange information. Those labeled Interlingua designate those interfaces which depend primarily upon the content of knowledge to be exchanged.

EUA to DB. Standards for DB access are well established, as SQL and RDA. They are however fairly awkward and not Turing-complete. We will not discuss those here. A number of commercial products and semi-products exist here. For instance, the Apple CL initiative generalizes such access for multiple SQL-based DBs. The common current approach is to interface via a standard language such as SQL. Knowledge-based access to databases has the potential to add knowledge supporting intelligent access to data and information stored in conventional databases.

EUA to AS. Differences in sensor technology have inhibited high-level standards here, although a variety of low-level standards exist. We will not discuss these here. As sensor-based systems start using more standard types of computing equipment we can expect to achieve more commonality.

KB to KB. The language in which to effect the interchange of knowledge among knowledge-based systems has two parts. One is the language in which to express the content of the information being exchanged. This is the focus of the Interlingua group under the Knowledge Sharing Effort (KSE). This kind of communication may only be effective where the ontologies match, are shared, can be circumscribed and can be translated. The other part of the interchange language is the specific communication language used to exchange the content expressions. The language in which to encapsulate, route and match senders with receivers of knowledge is the primary focus of the Knowledge Query and Manipulation Language (KQML) group under the KSE. KQML is of primary concern when implementing knowledge-based systems using knowledge repositories; the specific content language plays an important role, but secondary to KQML here.

KB to KBR. Questions on how to store knowledge representations in an effective and persistent way is part of the charter of the KQML working group. While this topic is not a focus of the current document, it is an important issue to address and to understand.

KB to DB. To make knowledge-based systems effective it is desirable that all voluminous factual information be maintained outside of the KB, although this may engender high costs in terms of performance. Research into suitable techniques, as caching, close coupling, etc., is in progress in many research sites. It appears that existing DB standards are insufficient to serve knowledge-based systems well. Object-oriented approaches are of interest. We are not now focusing on this issue but the topic is within the scope of the KQML group.

KB to AS. A knowledge-based system can have an important role in abstracting and monitoring sensor (AS) information. Certain information, when acquired, will force changes in knowledge, so that some learning mechanism much be invoked. A special case here is an active database system, which will also signal changes that may require knowledge update. This field is not well understood now, and probably not ready for standardization.

KBR to KBR. Interchange of knowledge content is hard to distinguish from a knowledge-based system to knowledge-based system interface, although it seems feasible if the internal representations match. The Interlingua is of primary importance here and KQML takes on a secondary role. The use of an Interlingua may motivate the internal consistency required for shared ontologies. The development of experimental shared ontologies is another part of the KSE.

KBR to DB. Active research and development is ongoing to use DB facilities for persistent storage of knowledge. Although current DBs are too inflexible, there seem to be no principles that would inhibit use of DBs. The focus will be on efficiency. Again — object-oriented structures hold out some promise in this arena.

KBR to AS. This interface does not make sense.

DB to DB. Distributed databases use replicated and fragmented data representations to enhance performance and availability. Consistency requires use of concurrency protocols. This is a relatively mature area, and outside of the purview of the KQML work.

DB to AS. Databases can be used to make sensor data persistent, but the update facilities provided by SQL are implementation and performance bottlenecks here. The DADAISM database design is configured to accommodate sensor data.

AS to AS. This is not a meaningful interface. Sensors are only senders of information, not receivers. An extension of the AS module to encompass effectors could enable this interface, but no intelligent processing would occur.

1.4 A Framework for Knowledge Interchange

We identify three major dimensions which can be distinguished for communicating intelligent agents: *Connectivity*, *Architecture* and *Communication*. We do not claim this is an exhaustive list, but rather serves to distinguish a number of current and planned systems.

Connectivity. The connectivity dimension focuses on the connectedness among communicating components distinguishing the specific interconnections and dependencies among components. For example, in many systems, the output of one component is the input to the next component; a full system is supported via a series of point-to-point links. In some systems, the output of one component is the input to several other components; a full system is made up of a collection of multicast links. In some other systems, e.g., blackboard systems, the output of one component is the input to an unknown number of other components; a full system is supported by broadcasting.

Architecture. The architecture dimension focuses on the degree to which agents may be added or removed from a system. In a static architecture, all system components are known at design time, and, the specific inputs as well as outputs are defined. In a dynamic architecture, not all system components are known at design time, and, the specific source of inputs as well as the destination of outputs are not fixed.

The architecture dimension captures the distinction between a system design in which all components must be present for a system to operate versus a system design in which multiple agents may be participating over time as well as entering and leaving active participation. In the former case, a system is a completely designed and assembled from known and well-understood interacting components. In the latter case, a dynamic architecture supports a highly flexible system in which intelligent agents may be added or removed at any time, and, there is sufficient overlap to allow the system to function with fewer or more of the components present.

Communication. The communication dimension focuses on the synchronicity of the communication between intelligent agents. Communication can be synchronous in which case a complete output is made available to another agent before the other agent can fully process it as a valid or complete input. Asynchronous communication allows for incremental processing via partial results and inputs.

Examples. We consider how three standard system architectures vary along these dimensions. First, consider a pipeline implementation where the complete output of one knowledge-based system is the entire input to the next. Further, there are a number of systems in a pipe to affect some system implementation. The final output may result either in a new input for the first element of the pipe, or, more likely, the final output will be analyzed and changes made to the input of (or selections made by) the first element of the pipe. Examples of this style of system include the PACT-0 demonstration of the collaborative multi-system design of an electromechanical device, and the 1992 Integrated Feasibility Demonstration of the DRPI where a planning system developed a detailed military forces employment and deployment plan and a simulator analyzed the plan with respect to transportation feasibility. The pipeline is characterized by

- **Connectivity:** a simple point-to-point connectivity, one knowledge-based system to the next

- **Architecture:** a static architecure known at system design time
- **Communication:** synchronous

The second system architecture to consider is a loosely coupled system in which a small number of components, approximately 10 or under, are to cooperate loosely in solving an overall system problem. An example here from the DRPI military transportation application domain is a planned architecture for a demonstration system to be put together over the next 12 to 18 months. The *"Intgrated Feasibility Demonstration"* system involves components in which the output of one component is input to multiple other components. We call this a loose coupling because while the interaction is more collaborative than the pipeline, each knowledge-based system component is still acting independently. That is, the level of integration consists of nearly complete output as available from a planning system, say, which is a complete input to a force module generator. Thus, very little of the distributed environment in which the knowledge-based systems participate is taken into account in the design or implementation of the knowledge-based system. The loosely coupled system is characterized by

- **Connectivity:** point-to-point but with some use of multicasting
- **Architecture:** a static architecure known at system design time
- **Communication:** synchronous and asynchronus

The final architecture we will consider is a tightly coupled system in which a larger number of components, approximately 100 or so, are to cooperate in a highly integrated system. Again, we take the application example from the DRPI because we are familiar with it. The ultimate military transportation system would use more components than in the current *Integrated Feasibility Demonstration System*, but primarily differs in how the systems interact. A planning system can make use of some of the incremental results from a scheduler but there may be several intervening knowledge-based systems in either the pipeline or loosely coupled systems. If the output of the scheduler's reasoning system which detected a scheduling conflict were immediately available to a planning system, the planning system may be able to respond with a different plan alternative immediately. The effect is to save potentially a considerable amount of work wasted on a plan alternative which is causing identifiable problems in other parts of the overall system. In order to accomplish this, each knowledge-based system component's reasoning system must be able to incrementally update other knowledge-based systems and be updatable itself in a dynamic manner. Further, for each knowledge-based system to be developed by application programmers in a reasonable amount of time, the reasoning systems should be augmented in as transparent a manner as possible. The tightly coupled system is characterized by

- **Connectivity:** point-to-point, multicast and broadcast communication
- **Architecture:** a dynamic architecure not fully known at system design time
- **Communication:** synchronous and asynchronus, but mostly asychronous

2 KQML

The Knowledge Query and Manipulation Language (KQML) is a language and an associated protocol to support the high level communication among intelligent agents. It can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to interact cooperatively in problem solving. We argue that KQML should be defined as more than a language with a syntax and semantics, but must also include a protocol which governs the use of the language (e.g., a pragmatic component).

2.1 Design Issues and Assumption

Architectural assumptions. Agents will typically be separate processes which may be running in the same address space or on separate machines. The processes will have a reliable information transport mechanism (e.g. TCP/IP streams) connecting them. We need a protocol that is simple and efficient to use to connect a few pre-defined agents on a single machine or on several machines on the same local area network. We also need the protocol to be an appropriate one to scale up to a scenario in which we have a large number (i.e. hundreds or even thousands) of communicating agents scattered across the global Internet and who are dynamically coming on and off line.

Communication Modes. KQML will support several modes of communication among agents along several independent dimensions. Along one dimension, it supports interactions which differ in the number of agents involved – from a single agent to a single agent (i.e., point-to-point), as well as messages from one agent to a set of agents (i.e., multicasting). Along another dimension, it permits one to specify the recipient agents either explicitly (e.g., by Internet address and port number), by symbolic address (e.g., to "to the *theTRANSCOMMapServer"* or even by a declarative description form of broadcast (e.g., "to any KIF-speaking agents interested in airport locations"). A final dimension involves synchronicity – the protocol must support *blocking* as well as *non-blocking* communication.

Syntactic assumptions. The actual representation of information at the most underlying level, is not of great importance; current implementations utilize Lisp s-expressions transmitted between processes in the form of ascii streams. The forms at the content layer depend on the content-language being used and may be represented as strings, if necessary. The forms at the message and communication layer will be ascii representations of lists with symbolp as the first element and whose remaining elements use the Common Lisp keyword argument convention.

However, newly emerging standards, such as those from the Object Management Group, may provide an entirely new basis for representing the content of the messages.

Security. Security is an issue in any distributed environment. We will need to develop conventions and procedures for authentication which will allow an agent to verify that another agent is who it purports to be. *Comment: We should take advantage of the Kerberos system for secure authentication being developed as a part of Project Athena. It appears to have all the right hooks and will be widely available on IBM, DEC, SUN and other Unix platforms.*

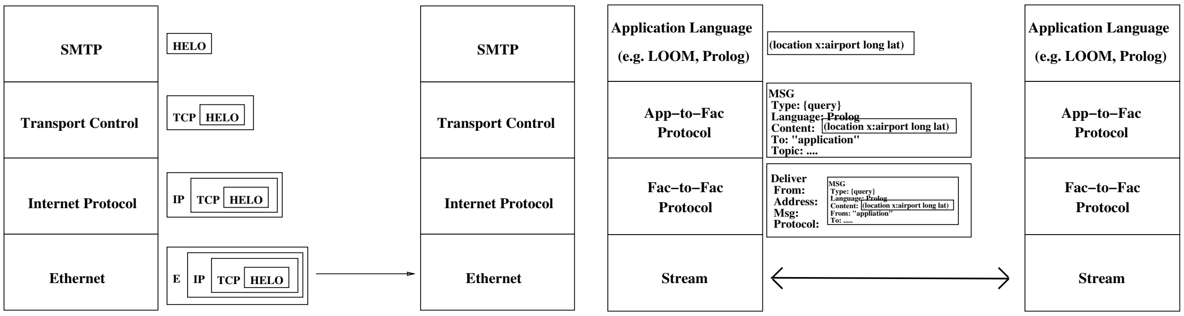

Figure 3: Modern Internet communication is governed by a "protocol stack" with distinct, well-defined layers. Communication between intelligent agents should also be governed by a protocol stack with distinct, well-defined layers.

Transaction. Interactions among knowledge-based systems have a different kind of transaction processing which will require something other than the now standard two-phase commit. That is because interacting agents may use information and knowledge gained from one information source for longer periods of time than read/write locks support. In one way, knowledge-based systems are similar to other advanced systems such as software engineering or CAD/CAM design environments (see Computing Surveys, 1991). Further, interactions among knowledge-based systems may better be cast in terms of belief spaces and/or logics of belief than in terms of low level transactions. The development of a good model to support transactions among intelligent agents is a research topic for the KQML group to consider sometime in the future. Developing a workable solution which is incrementally implementable may prove key to the ultimate success of the KQML effort.

Protocol Approach. The Knowledge Query and Manipulation Language (KQML) is a language and a protocol to support the high level communication among intelligent agents. It can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to interact cooperatively in problem solving. We argue that KQML should be defined as more than a language with a syntax and semantics, but must also include a protocol which governs the use of the language (i.e., a pragmatic component).

Using a protocol approach is common in modern communication and distributed processing. The first diagram in Figure 3 shows a simplified version of the standard protocol stack for network communication over an Internet. At the top of the stack is the application-level protocol, in this case SMTP (*Simple Mail Transfer Protocol*) and at the bottom is the low level protocol in which data is actually exchanged. From a mailer's point of view, it is communicating with another mailer using the SMTP protocol. It need not know any of the details of the protocols which support its communication.

We are developing a similar approach to support communication among intelligent agents – defining a protocol stack for transferring knowledge across the Internet. The second diagram in Figure 3 shows a simple protocol stack we are using for the model of KQML. We assume that the KQML protocol stack is an application protocol layer of the standard OSI model and thus assume reliable communication.

SKTP, a Simple Knowledge Transfer Protocol, supports KQML interactions and is de-

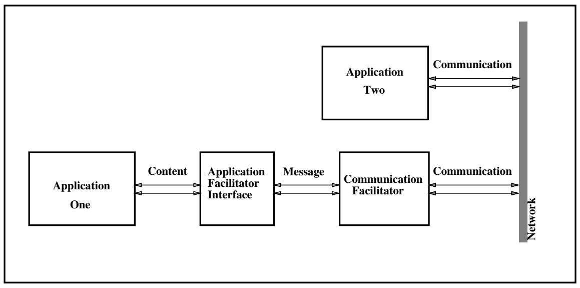

Figure 4: KQML expressions can be thought of as consisting of a content expression encapsulated in a message wrapper which is in turn encapsulated in a communication wrapper.

fined as a protocol stack with at least three layers: content, message and communication. These layers are built upon some reliable network transport mechanism.

2.2 KQML Layers

KQML expressions can be thought of as consisting of a content expression encapsulated in a message wrapper which is in turn encapsulated in a communication wrapper, as shown in Figure 4. Thus the language is thought of as being divided into three layers: content, message and communication. The content layer contains an expression in some language which encodes the knowledge to be conveyed. Because the content is opaque to KQML, the message layer adds a set of features which describe the content, i.e. the language it is expressed in, the ontology it assumes and the kind of speech act it represents (e.g., an assertion or a query). The final communication level adds a second layer of features to the message which describe the lower level communication parameters, such as the identity of the sender and recipient and whether the communication is meant to by synchronous or asynchronous.

Content Layer. KQML makes no commitments about the content layer. One can use it with any number of content languages, such as KRSL [3], KIF [8] or LOOM [4]. All that two intelligent agents need to do is agree on a language to use for communication. This does not preclude the use of or diminish the need for an *interlingua* such as KIF to support knowledge sharing, but it does permit two agents who are using the same internal language to use it as the communication language in the protocol.

Message Layer. The message layer is used to encode a message that one application would like to have transmitted to another application. These messages are of two general types — content messages and declaration messages. A "content" message contains a description of a piece of knowledge being offered or sought. "Declaration" messages are used to announce the presence of an agent, register its name, provide descriptions of the general types of information that the agent will send/receive, and the actual content messages sent between agents. The message layer can also be thought of as a "speech act layer". One of the most important attributes to specify about the content is what kind of "speech act" it represents – an assertion, a query, a response, an error message, etc.

Content Messages. A content message is used to describe a query, assertion or other speech act involving some sentence in the content language. It is represented as a list whose first element is the atom MSG and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. The following example message is a query expressed in KIF for which exactly one answer is sought.

```
(MSG
  :TYPE query
  :QUALIFIERS (:number-answers 1)
  :CONTENT-LANGUAGE KIF
  :CONTENT-ONTOLOGY (blocksWorld)
  :CONTENT-TOPIC (physical-properties)
  :CONTENT (color snow ?C))
```

Declaration Messages. A declaration message is used to provide metainformation about the content messages that the agent will generate and would like to receive. These declarations can be used to register a service (e.g., "I'll answer questions about the physical properties of blocks") and to register a need for a service (e.g., "I want to be keep current on the location of every block"). Syntactically, a declaration is a list whose first element is the atom DCL and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. The following example announces that agent *ap001* is willing to export assertions expressed in KIF about the color properties of things in a blocks world ontology.

```
(DCL
   :TYPE assert
   :DIRECTION export
   :MSG
    (MSG
      :TYPE assert
      :CONTENT-LANGUAGE KIF
      :CONTENT-ONTOLOGY (blocksWorld)
      :CONTENT-TOPIC (physical-properties)
      :CONTENT (color ?X ?Y)))
```

Communication Layer. At the communication layer, agents exchange *packages*. A package is a wrapper around a message which specifies some communication attributes, such as a specification of the sender and recipients. A package is represented as a list whose first element is the atom PACKAGE and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. In the following example, application *ap001* is sending a synchronous query to application *ap002*:

```
(PACKAGE :FROM ap001
         :TO ap002
         :ID DVL-f001-111791.10122291
         :COMM block
         :CONTENT
          (MSG
           :TYPE query
           :QUALIFIERS (:NUMBER-ANSWERS 1)
           :CONTENT-LANGUAGE KIF
           :CONTENT (color snow _C)))
```

2.3 KQML Content Layer

KQML makes no commitments about the content layer. One should be able to use it with any number of content languages, such as KIF [?] or LOOM [4]. All the two intelligent agents need to do is to agree on a language to use for communication. This does not preclude the use of or diminish the need for an *interlingua* such as KIF to support knowledge sharing. It does allow two agents who are using the same internal language to use it as the communication language in the protocol.

2.4 KQML Message Layer

The message layer is used to encode a message that one application would like to have transmitted to another application. These messages are of two general types — content messages and declaration messages. A “content” message contains a description of a piece of knowledge being offered or sought. “Declaration” messages are used to announce the presence of an agent, register its name and provide descriptions of the general types of information that the agent will send and would like to receive and the actual content baring messages sent between agents.

The message layer can also be thought of as a “speech act layer”. One of the most important attributes to specify about the content is what kind of ”speech act” it represents – an assertion, a query, a response, an error message, etc.

2.4.1 Content Messages

A content message is used to describe a query, assertion or other speech act involving some sentence in the content language. It is represented as a list whose first element is the atom MSG and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. Possible keyword arguments are:

:TYPE - <*Speech Act*>
: The speech act type of the message (e.g., query, assert, retract, etc.).

:QUALIFIERS - <*keyword list*> A keyword tagged list of qualifiers appropriate to the message type.

:CONTENT-LANGUAGE -<*A language name*>
: A term naming the language in which the CONTENT field is expressed.

:CONTENT-ONTOLOGY - <*An ontology name*>
: A term or list of terms chosen from a standard list naming the ontologies assumed.

:CONTENT-TOPIC - <*topic name*>
: A term or list of terms describing the topic of the knowledge within the given ontology.

:CONTENT - <*A sentence in the content language*>
: The actual knowledge to by conveyed expressed as a sentence in the content-language.

The following example message is a query expressed in KIF for which exactly one answer is sought.

```
(MSG
  :TYPE query
  :QUALIFIERS (:number-answers 1)
  :CONTENT-LANGUAGE kif
  :CONTENT-ONTOLOGY (blocksWorld)
  :CONTENT-TOPIC (physical-properties)
  :CONTENT (color snow ?C))
```

2.4.2 Declaration Messages

A declaration message is used to provide metainformation about the content messages that the agent will generate and would like to receive. These declarations can be used to register a service (e.g., "I'll answer questions about the physical properties of blocks") and to register a need for a service (e.g., "I want to be keep current on the location of every block").

Syntactically, a declaration is a list whose first element is the atom DCL and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. Possible keyword arguments are:

:TYPE - <*Speech act*>
The speech act type of the embedded (MSG ...) expression (e.g., assert, query).

:DIRECTION - <*OneOf(IMPORT, EXPORT)*>
Specifies whether the information is to be imported or exported.

:COMM - Specifies whether the service is being offered or sought in a blocking or nonblocking communication mode.

:MSG - a (MSG ...) expression which specifies the content level information that is to be imported or exported.

The following example announces that agent *a001* is willing to export assertions expressed in KIF about the color properties of things in a blocks world ontology.

```
(DCL
  :TYPE assert
  :DIRECTION export
  :MSG (MSG
         :TYPE assert
         :CONTENT-LANGUAGE KIF
         :CONTENT-ONTOLOGY (blocksWorld)
         :CONTENT-TOPIC (physical-properties)
         :CONTENT (color ?X ?Y)))
```

2.5 KQML Communication Layer

At the communication layer, agents exchange *packages*. A package is a wrapper around a message which specifies some communication attributes, such as a specification of the sender and recipients. A package is represented as a list whose fist element is the atom PACKAGE and whose remaining elements are alternating attribute-value pairs using the Common Lisp keyword argument format. Possible keyword arguments are:

:TYPE - <*Message type*>.
This is the type of the embedded message, i.e. either a *content* message or a *declaration* message. *Comment: If we have separate facilitator agents, then we know that packages of type* declaration DCL or MSG.

:FROM - <*Agent ID*>
The unique identifier of the sending agent.

:TO - <*Agent ID*>
The unique identifier or identifiers of the recipient agent(s).

:ID - <*Package ID*>
A unique identifier for this message. This should be generated at this layer (e.g. by the facilitator agent if one is being used) and is used to refer to the message later.

:COMM - <*Oneof(block, nonblock)*>
Specifies whether or not the communication is to be carried out in a *blocking* or *nonblocking* mode.

:IN-RESPONSE-TO - <*Package ID*>
A list of one or more package IDs which refer to earlier messages that this package is in response to.

:CONTENT - an (DCL ...) or (MSG ...) expression.

In the following example, application *ap001* is sending a synchronous query to application *ap002*:

```
(PACKAGE
  :FROM ap001
  :TO ap002
  :ID DVL-f001-111791.10122291
  :COMM block
  :CONTENT
    (MSG
      :TYPE query
      :QUALIFIERS (:NUMBER-ANSWERS 1)
      :CONTENT-LANGUAGE KIF
      :CONTENT (color snow _C)))
```

2.6 KQML Performatives

Message types play an important part in this protocol. They appear at the message level in both content and declaration messages and are akin to a "speech act" type in the theory of natural language communication. Message types determine what one can "do" or "perform" with the sentences in the content language.

2.6.1 Content Language

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

Since KQML is not assumed to be a superset of the Interlingua, it has to identify sentences of the KB by way of quoted sentences of a language that can be translated into the object language of the KB. This language is called the *content language* (CL).
The languages for the contents of requests and replies can be declared with `declare-content-languages`:

(MSG

`:TYPE` – `declare-content-languages`

`:REQUEST-CONTENT-LANGUAGE` – <*content-language*>
Declares what the content language for requests will be. From then on all content language sentences of requests received by the provider should be assumed to be in that language. The language for requests can be `interlingua` (the default), `local` to use the object language of the local KB of the provider, or a string that specifies some other content language

known by the provider. This request can only be handled successfully if the provider knows how to translate sentences of the request CL into sentences of the object language of its local KB.

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

)

2.6.2 Discourse Contexts

Requests and replies should be relativized to a current discourse context. A discourse context is a subset of the sentences that define the local KB of the provider.

The following messages allow to establish a discourse context that contains a subset of the sentences of the local KB of the provider. With `set-discourse-context` we can explicitly set the current discourse context to a subset of the sentences that define the local KB:

(MSG

`:TYPE` – `set-discourse-context`

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

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

)

`add-to-discourse-context` allows to add additional sentences to the current discourse context:

(MSG

`:TYPE` – `add-to-discourse-context`

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

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

)

`assert` allows to add a sentence which is not necessarily a member of the local KB to the current discourse context:

(`MSG`

`:TYPE` – `assert`

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

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

)

With `remove-from-discourse-context` we can remove a set of sentences from the current discourse context:

(`MSG`

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

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

`:CONTENT` – <*sentence-pattern*>
Request that all sentences in the current discourse context which match the supplied sentence pattern are removed from the current discourse context.

)

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

(`MSG`

`:TYPE` – `assign-truth-value`

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

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

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

)

2.6.3 Definitions

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

2.6.4 Question Answering

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

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

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

`(MSG`

- `:TYPE` – `query-sentence-status`
- `:REQUEST-CONTENT-LANGUAGE` – <*content-language*> (see above)
- `:REPLY-CONTENT-LANGUAGE` – <*content-language*> (see above)
- `:WORKLEVEL` – <*worklevel type*>
  Answers to the query in the `:CONTENT` slot will be found by performing some kind of inference. The total amount of inference (or work) to be invested to find all requested answers is controlled by the value of `:WORKLEVEL`. Its value can be `minimal` to request quick but probably incomplete answers, `maximal` to request the provider to derive answers without any (or maximal) resource bounds, or a number that specifies a maximal number of work units to be invested by the provider. What a work unit is is defined by the local KB of the provider.
- `:HOW-MANY` – <*natural number*>
  The provider should report new answers at the earliest possible point after it has derived at least `:HOW-MANY` new answers since the last report (default is 1). If the number of allotted resources got exhausted all answers derived so far will be reported. This kind of control is important if there is more than one answer, e.g., if the query is an open sentence or a pattern.
- `:REPORT-MODE` – <*Oneof(*`suspend,continuous`*)*>
  Controls what the provider should do after it has reported a number of answers as specified by `:HOW-MANY`. If the value is `suspend` the provider will suspend its answering activity until it receives a continuation message. If the value is `continuous` the provider will continuously try to find new answers until either no more answers can be found, the allotted resources are exhausted, or it receives a control message that tells it to stop. New answers will be reported whenever at least `:HOW-MANY` new answers are available.
- `:TRUTH-VALUES` – <*(phrase, phrase)*>
  A pair of CL phrases that describe valid truth values. All derived answers are required to have a truth value that is within the range of truth values

defined by the two supplied values. If the local KB of the provider does not have a notion of an ordering of truth values then the range is just the set of the two values. If the two values are identical this set will be a singleton set. The default is a special value `any` which indicates that answers of any truth value are acceptable.

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

)

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

`(MSG`

`:TYPE` – `query-about-topic`

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

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

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

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

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

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

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

)

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

`(MSG`

`:TYPE` – `assert-and-infer`

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

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

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

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

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

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

`:ASSERTION-MODE` – <*Oneof(*`actual,hypothetical`*)*>
If the value of this slot is `actual` then the sentences will be added as normal assertions to the current discourse context of the provider. If the value is `hypothetical` the sentences will be hypothetically assumed in the current discourse context (added to it) until all the answers are reported. Then all these assumptions and the answers depending on them will be removed again.

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

)

2.6.5 Control Messages

`control` messages allow some control of ongoing work performed by the provider.

(MSG

`:TYPE` – `control`

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

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

)

2.6.6 Replies

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

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

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

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

`(MSG`

- `:TYPE` – `success-reply`
- `:VALUE` – <*Oneof(*`success,failure`*)*>
- `:REQUEST-ID` – <*Package ID*>
  Holds the ID of the package that contained the request to which this message refers.
- `:EXPLANATION` – <*string*>
  If the value of `:VALUE` was `failure` this slot can be used to hold an english sentence that explains to any humans involved why a certain request could not be handled.

`)`

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

`(MSG`

- `:TYPE` – `content-reply`
- `:REQUEST-ID` – <*Package ID*> (see above)
- `:REPLY-NUMBER` – <*natural number*>
  The different report modes for queries allow for multiple replies to a particular query. The value of this slot indicates the number of this particular reply. Default is 1.
- `:CONTENT` – <*sentence set*>
  Contains a set of sentences that constitute replies to the query identified by `:REQUEST-ID`.

`)`

2.6.7 To Do

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

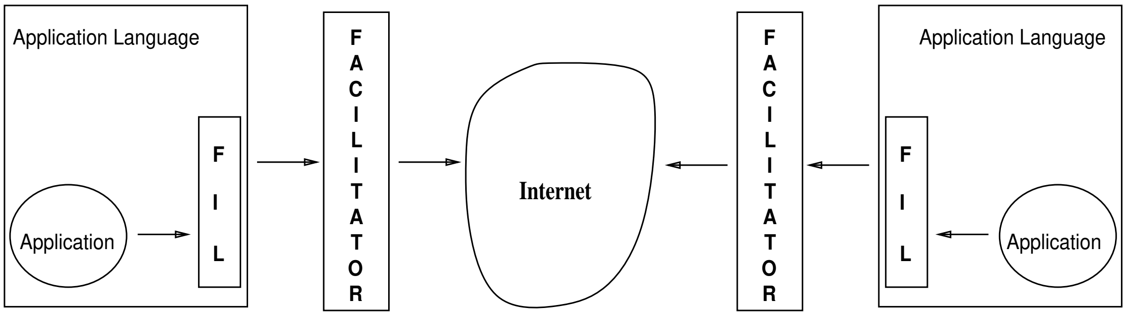

Figure 5: SKTP architecture for implementing KQML.

3 SKTP

3.1 Introduction

SKTP is a design for an implementation of the KQML protocol stack. The design follows the layered organization of the protocol. One section of the code handles the encapsulation and labeling of content expressions (implementing the message layer). Another section determines the destination of the messages and arranges (via some standard transport mechanism) for their delivery and the return of any immediate responses.

An important feature of SKTP is its tight integration with the implementation language of an application. This provides a nearly seamless interface between the application and the communication protocols, significantly reducing the difficulty of programming communicating agents and allowing a much tighter collaboration between processes than has been easily achievable before.

A preliminary implementation of SKTP has been written in Common Lisp and currently links applications written in a dialect of Prolog. We are designing interfaces to additional languages and systems.

There are four protocol layers shown in Figure 3. Each has a matching component in the implementation design shown in Figure 5. The overall communication is between *applications* written in an *application language*. Applications exchange expressions which have some meaning. This is the *content layer*. Expressions are selected for transmission to remote sites and wrapped in *messages*. This is the *message layer* and is implemented in Figure 5 by the module labeled **Facilitator Interface Library (FIL)**.

Messages might not have unambiguously specified destinations; they may have multiple destinations; they may require special handling. The layer which handles this "routing" of messages is the *communication layer* and is implemented by a separate agent called a **facilitator**. The underlying stream which carries the structured data between facilitators is, of course, the TCP/IP protocols provided by the Internet.

Each application is associated with a facilitator. Figure 6 shows an imagined collection of application communicating via SKTP over an Internet. While the diagram looks complex, the important gain made by the communication layer (implemented by the *facilitators*) is that all network communication is made using the same protocol, instead of a different protocol for each pairing of systems.

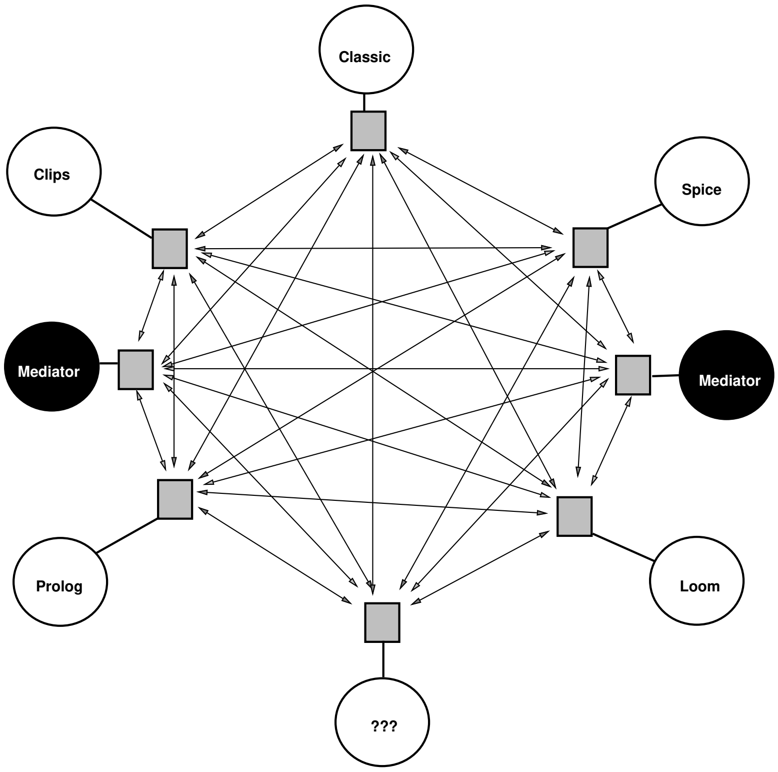

Figure 6: A network of processes communicating using the SKTP architecture.

3.2 Facilitator Interface Library

The *Facilitator Interface Library* (FIL) is the code which connects the varying worlds of different AI languages and systems to the communication world of KQML. The FIL performs three functions

- It interprets a set of declarations which describe the internal knowledge base transactions (e.g. definitions, queries, assertions) should be imported from or exported to remote systems.
- It contains code which monitors those internal transactions and arranges for the appropriate expressions to be transmitted as messages to a facilitator which will route them appropriately.
- It contains code which provides access points for a facilitator to deliver messages to the application (e.g. queries to be answered, assertions to be stored, etc.)

Because the FIL is tightly integrated with the application system, it is partly implemented in the underlying implementation language. For example, in our current prototype we have applications written in a dialect of Prolog which is implemented in Common Lisp. The FIL for these applications is written partly in Prolog and partly in Common Lisp.

3.2.1 Declarations

When using SKTP, an application program does not need to be modified to make "calls" on communication primitives. Instead, it is written as though the information that it needs was available locally (or, if it is primarily a supplier of information, as though there was no need to communicate). The program is augmented by a set of declarations which describe the internal transactions (assertions, queries, etc.) that are to be exported to remote sites and what types of transactions it is willing to process from remote sites. Declarations describe the following attributes of expression:

- Whether the expression is to be exported (sent to a remote site) or imported (accepted from a remote site)
- The type of the expression (e.g. assertion, query, definition)
- A characterization of the expressions of this type which are to be selected. E.g. in a relational system, the description might contain the name of the relation and the number of arguments, or in an object-oriented system it might be the class of the object.

Declarations which describe exports have to result in code which monitors the internal flow of expressions, selects appropriate ones for encapsulation, and is prepared to insert any replies into the the applications internal flow as though they originated locally.

Programming Models The basic idea of this approach is to completely hide the communication primitives from the application programmer. This is why the FIL will frequently need to be partly written in the underlying implementation language: it needs access to the internal routines of the language itself to help determine *when* expressions need to be transmitted and *which* expressions should be selected.

While this is a difficult job for the implementor of the FIL, it has a couple of significant advantages. The first is that it makes application programming **much easier**. The application programmer doesn't have to think about communication issues while writing the application, just prepare a set of declarations to go with it. The declarations themselves are written at a higher level of abstraction than communication code and so are easier to write. The second advantage is that it relieves the implementor of the FIL from having to design and implement a creative and clever way to integrate the communication primitives with the non procedural languages used in AI systems.

Tighter Collaboration This approach also makes it possible for applications to collaborate at a much tighter level of coupling than the simple "pipe" model of communication which is the only model currently used in DRPI or PACT. For example, in the current SKTP, if an application's internal processes require a particular goal to be satisfied remotely, the system will transmit that goal to a particular remote system and the answers will be seamlessly integrated into the local system's inferencing cycle. The system answering the remote query may also generate additional remote queries (possibly back to the originating system). All of this is transparent to the originating application which operates as though all the necessary information was being provided locally. Because the library intercepts *internal* transactions, two processes can actively collaborate, in parallel, on a single goal, without explicitly programming that collaboration.

This greatly elevates the state-of-the-practice for collaboration among separately written processes.

3.2.2 Exporting Messages

Declarations which state the the application is going to be exporting expressions require that the FIL contain code which will monitor the generation of these expressions, and act on appropriate ones.

When an application declares that it needs to export some of its queries to remote agents, the FIL creates code which monitors the internal generation of queries, queries which are normally generated for use by an internal inference engine, looking for ones which match the declared description. Queries which match the declarations are encapsulated as *messages* and passed to a *facilitator*. Depending on the application, the language, and the designer of the FIL, the FIL might wait for answers, or it might not; it might merge the answers from remote sites with local answers, or replace the local answers outright. These and other design decisions are made by the designer of the FIL and may be passed on as declaration options to the application programmer.

Similar considerations apply for declarations which state that the application is going to be exporting assertions. The primary difference is where in the implementation the FIL has to look for the expressions and what to do with any replies that are received.

3.2.3 Importing Messages

If an application is willing to answer queries for remote agents, or it is interested in receiving assertions from remote agents, it declares this in the same way as it would declare a need to export expressions. However, in this case the FIL has to establish a set of properly advertised functions or entry points to which a facilitator can deliver the queries or assertions.

The actual implementation of this connection depends on the design of the facilitator and the type of connection it has with its FILs. In various implementations the facilitator might be part of the same lisp image, or it might be a separate process connected by shared memory or some type of interprocess communication channel. Naturally the kind of "advertisement" needed to let the facilitator know how to deliver messages would depend on the type of connection between the two modules.

3.3 Facilitators

Facilitators bridge the gap between KQML messages and the Internet world of host names and TCP/IP streams. Using the metaphor of the Internet protocol stack, they are the KQML equivalent of Internet routers.

Facilitator accept messages from FILs and rely on the information in the message's fields to determine the appropriate destinations for the message. In some cases an application may identify a particular site as being the target of a message, either by host name (e.g. To: louise.vfl.paramax.com) or more symbolically (e.g. to: whichever machine is currently advertising itself as "geosys server A"). In other cases, the application may not know what an appropriate site is; the facilitator must rely on values of other message fields and a knowledge of what other sites are available in order to decide where to route the message.

3.3.1 Routing

Among the fields of the a KQML messages are:

- language - The language in which the encapsulated expression is written.

- type - "query", "assertion", "definition", etc.
- ontology - The general "framework" or "context" which the sender of the messages assumes and which the receiver must share.
- topic - The specific subject matter of the message. This field can only be interpreted in the context of a given *ontology*.

The declarations written for a program must provide sufficient information to allow the FIL to provide values for these fields. The *facilitator* uses them to search a database of remote agents who have declared that they are suitable targets for these messages. For example, if a facilitator receives (from a FIL) a message which is described as being:

- **Type:** *query*
- **Language:** *relational*
- **Ontology:** *DRPI-93*
- **Topic:** *Airports:Location*

it must look for one or more systems, somewhere on its connected network, which have advertised that they are willing to *import* queries of this type and answer them (By having matching "import" declarations.). It does this by searching a database of declarations looking for entries which match those of the question. When it finds them, it delivers the message to facilitators which are "representing" them and, waits for either an acknowledgment of receipt or actual replies.

While this process does not seem difficult on the surface, there are several problems which will require extensive work, especially as the number of agents available on a network increases and as the complexity of the information being exchanged increases.

3.3.2 Ontology and Topic Matching

The task of matching the declared *ontology* and *topic* of a message against a database of similar declarations is not well defined. While it is not difficult to develop simple examples and simple implementations to handle them it is also not difficult to create complex examples with no obvious implementation strategy.

Consider the case of a small and simply structure *ontology* which is divided into a small and shallow class hierarchy, such as **travel**, divided into fewer than ten possible subclasses such as *air*, *rail*, *car*, etc. Queries may be tagged as having one of these classes as their *topic*; knowledge bases can choose to advertise that they are willing to answer queries about one or more of these classes. As long as all of the participants understand which queries are about which topic and abide by the rules implicit in the simple ontology, the problem of matching messages with remote systems is reduced to simple string matching.

However, if the ontology is not quite as trivial, for example if it is described by a class hierarchy of moderate depth, such as the animal kingdom, then the problem is not so trivial. For example, if a knowledge base advertises that it is willing to IMPORT QUERIES about the class of *mammals* and a facilitator has a client trying to EXPORT a QUERY about *cows*, making the match is more difficult.

The routing task must be relatively simple in order to keep the facilitators relatively small and fast. The design of *ontologies* to be used for this purpose must be made with these problems and constraints in mind.

3.3.3 Database of Knowledge Based Services.

The second problem to be overcome is how a database of currently available applications is to be maintained. Actually gathering the data is not difficult. An assumption of this design is that all applications provide their FILs with declarations of the queries they can import and the assertions they can export, and that their associated *facilitators* will transmit these announcements over the network. The question is where and how should the database be implemented; there are several alternatives.

The database can take a variety of forms. It may be replicated in every facilitator, it may be centralized on an advertised machine, it may be stored in a distributed form across the network. Implementation strategies are based on the requirements of a particular environment.

For small sets of machines, a replicated implementation may be easiest. That is, each machine maintains its own complete copy of the list of network services. Maintenance of this list has to be performed in realtime; whenever a service begins or ends operation it has to be added to or removed from the list. With a small number of machines the overhead for each machine is not too great.

However, for even modest collections of machines (e.g., more than ten or twenty) the burden of broadcasting service announcements to every known machine, and the burden of processing such announcements from every known machine becomes noticeable, making a centralized approach is more suitable. One machine could serve as a repository for a single database. All processes would send both assertions of services they are making available and queries for needed services to this single machine.

In a very large network, e.g. a large campus network or the Internet itself, any central server will be both a bottleneck and a single point of failure. On this scale, a distributed approach is needed. A good example of this is the Internet distributed name service.

3.4 Implementation

Prototype versions of the components described above have been implemented. We have implemented

- A *facilitator interface library* for an implementation of the language Prolog.
- A *facilitator* which runs as a separate process within the same Common Lisp image as the Prolog language.
- A TCP/IP based communication package which links multiple Common Lisp images on different machines.

3.4.1 Prolog Facilitator Interface Library

An implementation of a FIL for Prolog has to handle the following events:

- A declaration by the local application of its communication status (what it needs and what it can provide)
- An assertion by the local application which needs to be transmitted to a remote application.
- A query by the local application which needs to be transmitted to a remote application.

- An assertion by a remote application which has been received by the local facilitator
- A query by a remote application which has been received by the local facilitator and needs to be answered

Declarations by the Local Application This facilitator provides routing for four types of application declarations:

- *Export Queries* Applications which want to send queries to remote sites.
- *Export Assertions* Applications willing to transmit new assertions to remote sites.
- *Import Queries* Applications willing to receive (and answer) queries from remote sites.
- *Import Assertions* Applications which want to receive assertions from remote sites.

Each declaration is accompanied by a description of the type of assertion or query to be exported or imported.

Declaration: Exporting Queries When a Prolog application declares that it needs to export some of its queries to remote sites, the facilitator interface creates code which will automatically transmit queries of the appropriate type to the facilitator.

The current implementation handles this by generating a Prolog rule of the form:

```
(query) :-
        =(L, (call-lisp (remote-solve (query))))
        member( (args), L)
```

For example, if an application declares that it wishes to export queries of the form:

```
(color X Y)
```

The facilitator interface will assert the following rule:

```
(color X Y) :-
        = (L, (call-lisp (remote-solve (color X Y))))
        member ( (X Y), L)
```

The function *remote-solve* transmits the query (with substitution performed on bound variables) to the facilitator which arranges for it to be answered by a remote site. The result is expected to be a list of variable bindings, e.g.

```
((sky blue) (emerald green))
```

This method of dealing with locally generated queries is simple and provides an effective way of dealing with the fact that the remote site returns a list of all solutions while the local site only expects one at a time while it backtracks through them and solves the problem of merging local answers with remote ones in a simple way. It is also very easy to implement.

Declaration: Exporting Assertions *Exporting assertions* is a declaration primarily used by forward chaining applications and not those implemented in Prolog, but we have included it here for the sake of completeness.

When a Prolog application declares that it is willing and able to export assertions of a particular type, it needs to create code to arrange that assertions which match those described by the declaration are forwarded to the facilitator. The current implementation has modified the low level "assert" and "retract" functions in the Prolog implementation to intercept and transmit matching assertions (and retractions) to the facilitator.

3.4.2 Declaration: Import Queries and Import Assertions

When a Prolog application declares that it is willing to accept queries of a particular type or that it is interested in receiving assertions of a particular type, all it needs to do is transmit that declaration to its local facilitator. The facilitator is responsible for insuring that other applications are aware of this service. The transmission is performed by a simple function call provided by the facilitator package.

When a Prolog system is willing to support this type of activity it it needs to provide the facilitator with functions to call whenever remote queries or assertions arrive from a remote site. This *registration* is made by a call to a function provided by the facilitator package.

Handling Locally Generated Queries and Assertions When the local Prolog generates a query or assertion which needs to be transmitted to a remote site, the preliminary work of the declaration handling (see above) has already arranged for the expression (the query or assertion) to reach the facilitator interface code. The next step is to package the expression into a *message*.

The facilitator provides a function for making *messages*. The interface package simply provides values for the following *message* fields:

- **content** The expression itself.
- **language** In this case, the name of the particular Prolog dialect, *Frolic*.
- **type** *query*, *assertion*, *retraction*, ...
- **ontology** This is a name which signifies the shared assumptions that the programs have about the knowledge they are using. It is a keyword shared among programs to keep other programs with the same **topic** from answering.
- **topic** For one simple ontology, this is simply the particular predicate used in the expression, e.g. COLOR/2. For another we used a list which represented the predicate and its arguments which were represented by either constants or untyped variables, e.g. (available JohnSmith ?Time ?Date ?Duration).

Prolog is not a good language in which to experiment with *ontology* definitions because most "real" ontologies tend to be object oriented while Prolog is relation oriented. For example, a service which can answer queries of the form:

```
(location ?x:airport ?y:coordinates)
```

which can be translated as "What is the location of a particular airport?" is not likely to advertise itself as a "location" server, providing information about the location of various objects. It is more likely to be an "airport" server, able to answer questions about various characteristics about airports, including their location. That is, it is more likely to be able to answer

```
(number-of-runways ?x:airport ?n:number)
```

than

```
(location ?x:museum ?y:coordinates)
```

A given application is likely to be able to provide some information about a set of objects in some "knowledge space". What kind of knowledge space is described by the *ontology* field. But the task of describing which objects in that space, and what relations about those objects, falls to the *topic* field.

A second function, *send-msg-to-facilitator* passes it on to the facilitator for routing.

Handling Remotely Generated Queries and Assertions To handle remotely generated queries and assertions all the interface package has to do is provide a function for the facilitator to call when it needs to pass a query to be processed or an assertion to be added/retracted from the database.

3.4.3 Common Lisp Facilitator

The facilitator's role is to route messages to appropriate recipients. Messages are not usually addressed to a specific individual site but to either a symbolically named service (e.g. Shipping-database or Planning-System-7) or a service which has advertised that it is willing to accept messages of this type. The facilitator is responsible for tracking which remote applications are interested in receiving assertions or are willing to answer queries on various topics.

To accomplish this, each facilitator maintains its own database of remote applications. Each entry in the database provides the Internet address of the host that the application is running on and a TCP/IP port address for the facilitator on that host. The entries are indexed by the types of messages the applications are willing to accept. Messages are characterized, as described earlier, by the same fields used to construct them: *type*, *language*, *ontology*, *topic* and also *communication style*.

The database is maintained jointly by all active facilitators using the following rules:

- When a local application declares that it is willing to import queries or assertions, the facilitator broadcasts that to all sites which may be running a facilitator.
- When a facilitator receives a declaration from another facilitator it acknowledges it by sending a list of imports that its applications are willing to accept.

The first rule lets everyone know about any new services. The current implementation is awkward in that it requires a list of machines where facilitators might be running. We will be replacing this with a separate service which tracks running facilitators and distributes new messages to them.

The second rule insures that new facilitators which announce their services are immediately apprised of other facilitators on the net and can build their own database.

3.4.4 Common Lisp TCP/IP

The facilitator is implemented using a locally written TCP/IP interface which allows Common Lisp applications to act as TCP/IP stream clients or servers. It provides client functions to open streams to remote TCP/IP ports using hostnames (or Internet addresses) and service names (or numbers). It also creates a separate process (within a Lucid Common Lisp image) which monitors a specified port and will spin off additional subprocesses when remote system communicate with that port. (That is, it implements a standard UNIX server program.)

4 Conclusions

KQML is a language which supports the high level communication among intelligent agents. It can be used as a language for an application program to interact with an intelligent system or for two or more intelligent systems to interact cooperatively in problem solving. SKTP, a Simple Knowledge Transfer Protocol, supports KQML interactions and is defined as a protocol stack with at least three layers: content at the application level, message at the application to facilitator level, and communication at the facilitator to facilitator level. Additional layers appear below these three to supply reliable communication streams between the processes. The content layer contains an expression in some language which encodes the knowledge to be conveyed. The message layer adds additional attributes which describe attributes of the content layer such as the language it is expressed in, the ontology it assumes and the kind of speech act it represents (e.g. an assertion or a query). The final communication layer adds still more attributes which describe the lower level communication parameters, such as the identity of the sender and recipient and whether or not the communication is meant to by synchronous or asynchronous.

We have implemented an experimental prototype of SKTP which uses *communication facilitators* as intelligent "routers" to simplify the application interface and realize the protocol. Facilitators provide a declarative framework in which applications specify their knowledge needs and the knowledge services they offer, establish communication channels between appropriate agents, and mediate the resulting dialogue.

KQML is part of a larger DARPA-sponsored Knowledge Sharing effort focused on developing techniques and tools to promote the sharing on knowledge in intelligent systems. The next steps in this research will be apply this integration approach in several distributed testbeds. Examples of applications envisioned include intelligent multi-agent design systems supporting collaborative designs of complex circuits and devices by multiple design teams as well as intelligent planning, scheduling and replanning agents supporting distributed transportation planning and scheduling applications.

5 Acknowledgements

The concepts and ideas in this paper are the result of contributions from a great many people. We list here some of their names.

Jose-Luis Ambite
Hans Chalupsky
Surajit Chaudhari
Steve Cross
Jim Davis
Tim Finin
Rich Fritzson
Mike Genesereth
Bruce Hitson
Michael Huhns
Eric Mays
Don McKay
Bob Neches
Clifford Neuman
Ramesh Patil
Peter Rathmann
Stu Shapiro
Marty Tenenbaum
Craig Thompson
Jay Weber
Gio Wiederhold
Mike Williams

References

[1] H. Chalupsky. Belief ascription by way of simulative reasoning. Unpublished dissertation proposal, 1991.

[2] Tim Finin, Rich Fritzson, Don McKay, Robin McIntire, and Tony Ohare. The intelligent system server delivering AI to complex systems. In *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence – Architectures, languages and Algorithms*, October 1989.

[3] N. Lehrer. KRSL Version 2.0. Language specification and manual, 1992.

[4] Robert MacGregor and Robert Bates. The LOOM knowledge representation language. In *Proceedings of the Knowledge-Based Systems Workshop*, April 1987.

[5] A. S. Maida and S. C. Shapiro. Intensional concepts in propositional semantic networks. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 291–330. Morgan Kaufmann, Los Altos, CA, 1985.

[6] Don McKay, Tim Finin, and Anthony O'Hare. The intelligent database interface. In *Proceedings of the 7^{th} National Conference on Artificial Intelligence*, 1990.

[7] J. G. Neal and S. C. Shapiro. Knowledge representation for reasoning about language. In J. C. Boudreaux, B. W. Hamill, and R. Jernigan, editors, *The Role of Language in Problem Solving 2*, pages 27–46. Elsevier Science Publishers, 1987.

[8] 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.

[9] Anthony B. O'Hare. The intelligent data interface language. Technical Report PRC-LBS-8908, Unisys Paoli Research Center, June 1989.

[10] Anthony B. O'Hare and Amit Sheth. The architecture of BrAID: A system for efficient AI/DB integration. Technical Report PRC-LBS-8907, Unisys Paoli Research Center, June 1989.

[11] N. Roussopoulos. An incremental access method for ViewCache: Concept, algorithms, and cost analysis. *ACM TODS*, 16(3):535–563, 1991.

[12] N. Roussopoulos and A. Delis. Modern client-server DBMS architectures. Technical report, Computer Science, 1991.

[13] S. C. Shapiro. The SNePS semantic network processing system. In N. V. Findler, editor, *Associative Networks: The Representation and Use of Knowledge by Computers*, pages 179–203. Academic Press, New York, 1979.

[14] S. C. Shapiro and W. J. Rapaport. SNePS considered as a fully intensional propositional semantic network. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier: Essays in the Representation of Knowledge*, pages 262–315. Springer-Verlag, New York, 1987.

[15] S. C. Shapiro and W. J. Rapaport. Models and minds: knowledge representation for natural-language competence. In R. Cummins and J. Pollock, editors, *Philosophical AI: Computational Approaches to Reasoning*, pages 215–259. MIT Press, Cambridge, MA, 1992.