SEMANTICS FOR AN AGENT COMMUNICATION LANGUAGE

Yannis Labrou

A Doctoral Dissertation for the PhD Defense Examination
Submitted to the Defense Committee
at the Computer Science and Electrical Engineering Department (CSEE)
University of Maryland Graduate School
Baltimore, Maryland 21228-5398

August 16, 1996

Program of Study Committee:

Chairman: Dr. Timothy Finin, Professor, Computer Science Electrical Engineering Dept., UMBC
Member: Dr. James Mayfield, Associate Professor, CSEE Dept., UMBC
Member: Dr. Yun Peng, Assistant Professor, CSEE Dept., UMBC
Member: Dr. Charles Nicholas, Associate Professor, CSEE Dept., UMBC
Member: Dr. Yelena Yesha, Professor, CSEE Dept., UMBC
Member: Dr. V.S. Subrahmanian, Associate Professor, Department of Computer Science, UMCP

Abstract

We address the issue of semantics for an *agent communication language*. In particular, the specification and semantics of Knowledge Query Manipulation Language (KQML), and the logical architecture of KQML–speaking agents are investigated. KQML is a language and protocol to support communication between (intelligent) software agents. First, we present a new specification for the KQML language that corrects a number of outstanding problems. Then, based on ideas from speech act theory, we propose a semantic description for KQML that associates "cognitive" states of the agent with the use of the language's primitives (performatives). We use this approach to describe the semantics for the whole set of *reserved* KQML performatives. Building on the semantics, we devise the conversation policies, *i.e.*, a formal description of how KQML performatives may be combined into KQML exchanges (conversations), using a *Definite Clause Grammar*. Finally, we present the code for a Prolog program that shows how the conversation policies can be incorporated in a software agent that speaks KQML.

The research presented here finalizes the specification of KQML and provides semantics for the language. The accompanying code shows how this research can be used to build KQML agents that make use of all of the powerful features of KQML. Thus, we offer the research community a completely defined communication language that can be used for knowledge exchange between software agents. Also, our research offers methods for: (1) a speech act theory–based semantic description of a language of communication acts, and (2) the specification of the protocols associated with these acts. Languages of communication acts address the issue of communication between software applications at a level of abstraction that could prove particularly useful to the emerging *software agents* paradigm of software design and developement.

This is a version of the Ph.D. dissertation of Yannis Labrou, formatted for smaller size and intented for private use. The content of this version is the same as that of the "official" version delivered to the Graduate School at UMBC, minus the *front matter*. The appearance of the "official" version is tailored to comply with Graduate School's guidelines, so this version might have some LaTeX glitches.

Please do not circulate

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This research is concerned with communication between software agents. In this first chapter, we intend to guide the reader through the maze of agent[1]–related buzzwords and technologies, define our notion of agent communication, and gradually introduce and position the agent communication language KQML (Knowledge Query and Manipulation Language), whose specification and semantics constitute the subject matter of this dissertation.

1.1 On software agent technologies

We view *software agents* as a new paradigm of software design. Software agents do not define a new discipline but rather a new way to view existing technologies as tools to build software applications that dynamically interact and communicate with their immediate environment (user, local resources and computer system) and/or the world, in an autonomous (or semi–autonomous), task–oriented fashion.

A wide variety of systems, languages, frameworks and standards efforts are associated with software agents; this is due in part to the vagueness of the term "software agent." Let us attempt to tease apart these approaches, and show how an *agent* communication language relates to each of them. Figure 1.1 shows a taxonomy of technologies important to software agents. We divide agent technologies into two broad categories: *agent languages* and *coordination protocols*.

The agent languages category included all languages that can be used to implement software agents. Virtually any programming language can be used for software agent development. One class of languages that has gained much attention lately is the so–called *scripting languages*, especially those designed for *mobile programs*. Languages like Tcl/Tk, Java, Telescript, *etc.*, offer the advantage of a level of abstraction that seems particularly attractive for the development of software agents. We place them under the agent languages rubric, because they can be used to program software agents. We distinguish such languages from agent communication languages, though, because they are designed primarily to control processes on a single platform. To the extent that these languages contain communication primitives tailored to agent development, they are largely concerned with the

[1]We will use the terms *agent* and *software agent* interchangeably. We will use "it" to refer to agents or to talk about them, instead of using "he" or "she."

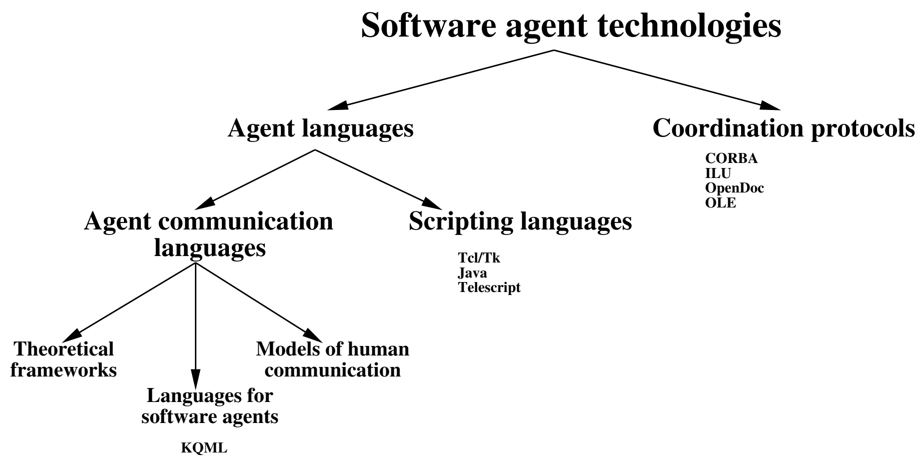

Figure 1.1: A taxonomy of agent–related technologies.

transportation of a single agent from one machine to another.

In contrast, agent communication languages are designed specifically to describe and facilitate communication among two or more agents. Three broad sub–categories may be identified under the label of agent communication languages: *models of human communication*, *theoretical frameworks*, and *communication languages for software agents*. Human communication is traditionally modeled in terms of speech act theory ([4, 62]). Considerable work has been done (*e.g.*, by Cohen and Lesveque ([12]), and Singh ([65])) to capture the assumptions and conventions of interaction between human agents and subsequently translate them into workable paradigms for the development of their artificial counterparts. Often, such work leads to theoretical frameworks for artificial agents with human–like capabilities ([13, 64]). Such frameworks attempt to account for all aspects of the internal state of an artificial autonomous agent, with particular attention to how this state changes as the agent interacts (and/or communicates) with the world or with other agents. Sometimes, as in the case of Agent Oriented Programming ([63]), those frameworks may evolve into implemented software systems. In contrast, *agent communication languages* (ACLs) are concerned strictly with the communication between such computational entities. An ACL (the sub–category that includes KQML) is more than a protocol for the exchange of data, because an *attitude* about what is exchanged by the agents is also communicated, rather than just content or data. An ACL may be thought of as a communication protocol (or a collection of protocols) that supports many *message types*.

The other main class of software agent technologies is that of standards and coordination protocols. CORBA, ILU, OpenDoc, OLE, *etc.*, are efforts that are often promulgated as solutions to the agent communication problem. Driving such work is the difficulty of running applications in dynamic, (and most of the time, object–oriented) distributed environments. The primary concern of these technologies is to ensure that applications can exchange data structures (usually objects) and methods across disparate platforms. Although the results of such standards efforts will be useful in the development of software agents, they do not provide complete answers to the problems of agent communication. Thinking of software agents as collections of data structures and methods on them introduces an inadequate level of abstraction, because such applications will be built on widely disparate platforms (not just object–oriented) and with varying degrees of intelligent behavior. Thus, these standards and protocols are best viewed as a substrate on which agent languages might be

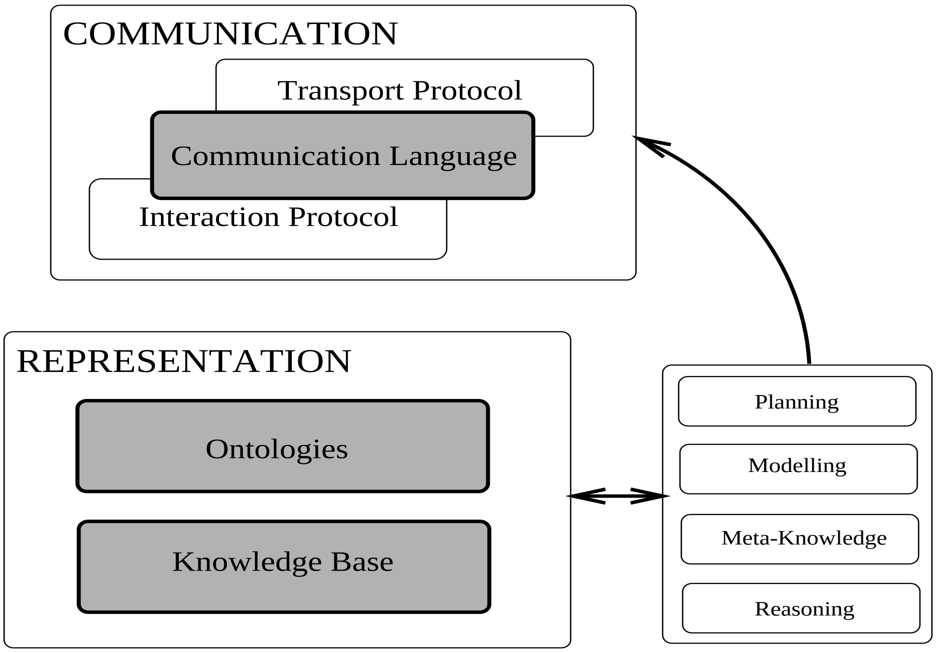

Figure 1.2: An abstract model for interoperating software agents

built.

1.2 A layered approach towards interoperability

Agent technologies, *e.g.*, programming languages, communication languages, protocols and agent models, may allow us to design and build software agents, but the essential property for the success of such applications in today's paradigm of networked computing is their ability to effectively communicate and exchange *knowledge* with one another. Such knowledge will be exchanged (and subsequently "understood") across different platforms, systems, architectures and languages, so achieving *interoperability* emerges as the next milestone in this new paradigm. Let us address the issue of software agents and the *desired interoperability* in more detail. Figure 1.2 summarizes the possible components of an agent capable of such amazing feats; they are grouped into representation components, communication components, and components that are not directly related to shared understanding.

Mutual understanding of what is represented may be divided into two subproblems: 1) translation from one representation language to another (or from one family of representation languages to another); and 2) sharing the semantic content (and often the pragmatic content) of the represented knowledge among different applications. Translation alone is not sufficient because each knowledge base[2] holds implicit assumptions about the meaning of what is represented. If two applications are to understand each other's knowledge, such assumptions must also be shared. That is, the semantic content of the various tokens must be preserved.

Communication, on the other hand, is a threefold problem: 1) interaction protocol; 2)

[2]We use the term *knowledge base* to refer to the knowledge content that a software agent incorporates.

communication language; and 3) transport protocol. The interaction protocol refers to the high level strategy, pursued by the software agent, that governs its interaction with other agents. Such a protocol can range from negotiation schemes and game theory protocols to protocols as simple as "every time you do not know something, find someone who might know and ask." The communication language is the medium through which the attitudes regarding the content of the exchange are communicated. It is the communication language that suggests whether the content of the communication is an assertion, a request or some form of query. The transport protocol is the actual transport mechanism used for the communication, such as TCP, SMTP, http, *etc.* Practical considerations may favor the use of one of those over others.[3]

Software agents may (or may not) have other components to help the agent carry out its business. The ability to reason about their own actions, to represent metaknowledge, to plan activities or to model other agents can enhance the capabilities of a software agent. Such components are peripheral to the knowledge base, although they are usually built on top of it. They may use the knowledge base or the communication language to carry through their agendas, but the issues associated with them should be viewed as orthogonal to the issues of mutual understanding and communication.

The Knowledge–Sharing Effort (KSE), sponsored by the Advanced Research Projects Agency (ARPA), the Air Force Office of Scientific Research (AFOSR), the Corporation for National Research Initiative (NRI) and the National Science Foundation (NSF), is an initiative to develop technical infrastructure to support knowledge sharing among systems ([55]). Although originally not intended as an initiative to address the issues of knowledge exchange between software agents, the KSE offers a layered approach towards interoperability, along the lines of Figure 1.2. The KSE is organized around the following three working groups, each of which addresses a complementary problem identified in current knowledge representation technology: *Interlingua*, *Shared Reusable Knowledge Bases*, and *External Interfaces*.

- The *Interlingua Group* is developing a common language for expressing the content of a knowledge base. This group has published a specification document describing the *Knowledge Interchange Format*, or KIF ([40]). KIF can be used to support translation from one content language to another, or as a common content language between two agents that use different native representation languages. Information about KIF and associated tools is available over the Internet. [4] A group within the KSE with a similar scope is the *KRSS Group* (Knowledge Representation System Specification). This group focuses on the definition of common constructs within families of representation languages. The group has produced a common specification for terminological representations in the KL–ONE family.[5]

- The *SRKB Group* (Shared, Reusable Knowledge Bases) is concerned with facilitating consensus on the content of sharable knowledge bases, with sub–interests in shared knowledge for particular topic areas and in topic–independent development tools and methodologies. It has established a repository for sharable ontologies and tools, which

[3]For systems that use *firewalls*, SMTP is a more likely choice than TCP.

[4]The URL is `http://www.cs.umbc.edu/kse/kif/`.

[5]This document and other information on the KRSS group is available as `http://www.cs.umbc.edu /kse/krss/`.

is available over the Internet.[6]

- The *External Interfaces Group* is interested in run–time interaction between knowledge–based systems and other modules in a run–time environment. Special attention has been given to two important cases: communication between two knowledge–based systems and communication between a knowledge–based system and a conventional database management system ([59]). The KQML language is one of the main results to come out of the External Interfaces Group of the KSE.[7]

These three groups of the KSE roughly address the interoperability issues at the levels identified by the three shaded boxes of Figure 1.2. The results of the research efforts, namely KIF, Ontolingua and KQML, are the specific solutions suggested for these interoperability issues and we briefly introduce them next.

1.2.1 Knowledge Interchange Format (KIF)

KIF is the solution suggested by the KSE for the syntactic aspects of representations for knowledge sharing. The language is intended as a powerful vehicle to express knowledge and meta–knowledge. There were two different intentions behind the development of a language like KIF: 1) creation of a *lingua franca* for the development of intelligent applications, with an emphasis on interoperation (in cooperation with the other components of the "package solution" of the KSE); and 2) creation of a common interchange format so that with the use of "translators" one could translate from language A to KIF and from KIF to language B instead of translating from A to B.[8] KIF has found its way into applications[9] ([41]), but it remains to be proven whether it will fulfill any of its intended roles ([43]).

KIF is a prefix version of first order predicate calculus with extensions to support non-monotonic reasoning and definitions. The semantics of the KIF core (KIF without rules and definitions) is similar to that of first order logic. Despite the extensions and restrictions, the core language retains the fundamental characteristics of first order logic, including compactness and the semi–decidability of logical entailment.

1.2.2 Ontologies and Ontolingua

The SRKB Working Group is also working on the problem of sharing the content of formally represented knowledge. Sharing content requires more than a formalism (such as KIF) and a communication language (KQML). Although the problem of understanding what must be held in common among communicating agents is a fundamental question of philosophy and science, the SRKB is focusing on the practical problem of building knowledge–based software that can be reused as off–the–shelf technology. The approach is to focus on common ontologies ([55]). Every knowledge–based system relies on some conceptualization of the

[6]The URL is `http://www.cs.umbc.edu/kse/srkb/`.

[7]General information about KQML is available from `http://www.cs.umbc.edu/kqml/`.

[8]So for n languages the number of translators needed would be $2 \times n$ instead of n^2. Of course the advantages of having a common interchange format go beyond such a reduction; but defining the advantages (and disadvantages) of such an approach has been an issue of debate.

[9]Primarily in conjunction with ACL, an implementation of KQML, which is standard KQML with a commitment to KIF as the content language.

world (objects, qualities, distinctions and relationships that matter for performing some task) that is embodied in concepts, distinctions, *etc.*, in a formal representation scheme. A *common ontology* refers to an explicit specification of the ontological commitments of a set of programs. Such a specification should be an objective (*i.e.*, interpretable outside of the program) description of the concepts and relationships that the programs use to interact with other programs, knowledge bases, and human users. An agent commits to an ontology if its observable actions are consistent with the definitions in the ontology.

The SRKB Group has worked on the construction of ontologies for various domains. Ontologies are written in KIF, using the definitional vocabulary of Ontolingua.[10] Each ontology defines a set of classes, functions, and object constants for some domain of discourse, and includes an axiomatization to constrain their interpretation. The resulting language (the basic logic of KIF plus the vocabulary and theory from the ontologies) allows for the sentences to be interpreted unambiguously and independent of context, making the relevant detail explicit. These ontologies can then be used by communicating applications in order to share their ontological commitments. The group has developed a considerable number of ontologies, in a variety of domains that might be of interest to software applications.

1.2.3 Knowledge Query Manipulation Language (KQML)

KQML was conceived as both a message format and a message–handling protocol to support run–time knowledge sharing among agents. The key features of KQML may be summarized as follows:

- KQML messages are opaque to the content they carry. KQML messages do not merely communicate sentences in some language, but rather communicate an attitude about the content (assertion, request, query, *etc.*).
- The language's primitives are called *performatives.* As the term suggests, the concept is related to *speech act theory* (more on *speech act theory* in Section 2.1). Performatives define the permissible actions (operations) that agents may attempt in communicating with one another.
- An environment of KQML speaking agents may be enriched with special agents, called *facilitators*, that provide such functions as: association of physical addresses with symbolic names; registration of agents and/or services offered and sought by agents; and communication services (forwarding, brokering *etc.*). To use a metaphor, facilitators act as efficient secretaries for the agents in their network domain.

Intelligent interaction amongst agents involves more than an exchange of messages. KQML is an attempt to dissociate the many issues of intelligent interaction from the communication language, with the understanding than an *agent communication language* should define a set of standard message types that are to be interpreted identically by all interacting parties when needing to communicate something more than pre–defined or fixed statements of facts. Chapter 3 introduces in more detail the basic ideas behind KQML with an emphasis on the *speech act theory* origins of the language; speech act theory is fundamental for the approach we decided to follow in this research, towards the semantic description of the language.

[10] See `http://www-ksl.stanford.edu/knowledge-sharing/ontologies/README.html` for more information on ontologies and the various projects with which the SRKB Group has been involved.

1.3 The scope of this research

Although KIF, Ontolingua and KQML were conceived as a packaged solution to the interoperability question, we study KQML in isolation. This dissertation is about revising the old and outdated KQML specification and providing semantics in an approach that emphasizes the commitment to the linguistic focus of the language and the non-commitment to agent models, belief models and inter–agent interaction models. This is the result of our observation that almost all communication languages that have appeared to date are tied to specific applications or interaction models and thus cannot be used by a wide range of software systems. We do *not* intend to provide a paradigm for the linguistic communication between software agents, a theory of *agency*, or a scheme for the semantic description of speech acts in general.

Chapter 2 is a survey of the literature that is relevant to this research. In Chapter 3 we introduce the necessary material for understanding and justifying our approach and we briefly lay down the course we are taking. The following three chapters constitute the core of this research: Chapter 4 is a detailed specification of the language with various examples of the proper use of its primitives; in Chapter 5 we provide a semantic description of the language's primitives, based on the linguistic origins of KQML; and in Chapter 6 we present the *conversation policies* for the language's primitives, *i.e.*, a formal specification of the possible placement of the language's *communication acts* in exchanges between agents. Chapter 7 discusses the code that implements the *conversation policies*, in Prolog, along with the *handler functions* (a listing of the code for the *conversation policies* is included in Appendix A). Finally, in Chapter 8 we summarize the contributions of this research.

Our research makes KQML a fully specified language, with semantics and conversation policies for its communication primitives. But, in the abstract, KQML also suggests a novel approach towards interaction and knowledge exchange between software applications, of the kind termed as *software agents*; agents interact through *communication acts* that communicate *attitudes* about information and these acts can be combined into complex interactions between agents. So, beyond KQML, our research offers a method to define the semantics of communication primitives that express *attitudes*, along with a method for the specification of the possible interactions when using such primitives.

Chapter 2

Literature Survey

In this chapter we will provide the background necessary for this research and we will review the literature and related work. We provide an introduction to speech act theory, we explore the issue of speech act semantics, as it relates to our work, and finally we present Searle's approach regarding speech acts, which has been a primary influence for this research. Then we provide a brief introduction to *Distributed Artificial Intelligence* (DAI) and explore the issue of (agent) communication languages (like KQML) in the context of DAI. Finally we offer a brief introduction to KQML, its origins, its objectives and its primary features.

2.1 Speech act theory

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.

Austin pointed out that the major role of language, communication, is a kind of action ([4]). He suggested that speakers do not simply utter sentences that are true or false, but rather perform speech actions such as requests, suggestions, promises, *etc.* As a result, speech act theory is concerned with language as action and speech acts are considered the minimal units of human communication. All utterances are speech acts, with the primary understanding that all utterances are *actions* of some sort.

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 listener, 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 listener shuts the door. Most elementary speech acts that are performed by successful utterances are of the form $F(P)$ where F is the illocutionary force and P is the propositional content of the illocutionary act.[1] The illocutionary force

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

classifies speech acts according to the following classes:[2] 1) *assertives*, which are statements of facts, 2) *directives*, which are commands, requests or suggestions, 3) *commisives*, *e.g.*, promises, which commit the speaker to a course of action, 4) *declaratives*, which entail the occurrence of an action in themselves,[3] and 5) *expressives*, which express feelings and attitudes. In the example given above, the illocutionary force would be that of an order (a *directive* in speech act terminology) and the propositional content would be "Shut the door." It is not necessarily the case that a verb is needed to express the illocutionary force. "I order you to shut the door" is equivalent to "Shut the door !", with the exclamation mark suggesting the order (or phonetic features if vocal communication takes place).

2.1.1 Speech act semantics

There is a variety of viewpoints in the literature regarding what should be considered as semantics for speech acts.[4] One approach is that of interpreting as semantics the conditions under which speech acts may be said to have occurred, thus identifying the sort of speech act that took place, *e.g.*, assertive *vs.* directive, and/or direct *vs.* indirect speech act. Another viewpoint is interested in identifying the effects that a speech act may or ought to have on the speaker's and listener's cognitive states, and how an agent ought to respond to a specific speech act. Since this approach relates to the perlocutionary aspects of speech acts, it may be classified as *pragmatics* rather than semantics. The third viewpoint is that semantics are the conditions under which one can affirm that a given speech act has been satisfied, *e.g.*, a command is satisfied if it is obeyed and a promise is satisfied if it is kept in the future.

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. So representation 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 ([62]) 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, 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 ([11], [15] and [12]). Campbell ([9]) proposes using 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* ([16]), use a *believe* modal operator based on Hintikka's ideas on the propositional attitudes of knowledge and belief ([47]). Singh is interested in modeling agents in terms of beliefs and intentions ([65]) and uses this description to provide a semantic approach for speech acts ([67]), 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

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

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

[4] A collection of recent work on speech acts, with an AI interest, is [14].

the *possible–world model* that has an axiomatization in terms of modal logic. See [48] for an introduction to the possible worlds model and the issues related to it.

The research approaches for the semantics for speech acts have been driven by the objectives that the semantic approach serves; the approach often defines what is considered as semantics. In our research, for reasons explained in Chapter 3, we consider Searle's linguistic description of speech acts (and illocutionary acts in particular) that is briefly described next.

2.1.2 Speech act theory *à la* Searle

Searle, in extending Austin's ideas, tried to specify necessary and sufficient conditions for the successful performance of speech acts, involving the intentions of the participants ([62]). In [61], he introduced a formal model for speech acts, and analyzed the illocutionary force into seven components, attempting to capture all aspects of the illocutionary act. In our approach towards defining KQML semantics in terms of speech act theory, we adopt Searle's descriptive framework regarding the illocutionary force. The seven components of illocutionary force that Searle suggests are the following:

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 further distinguishes illocutionary acts with the same illocutionary point. For example, both "Shut the door!" and "Would you please close the door?" are directives, but one is a command and the other 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* constrain what can be in the propositional content P for a specific force F. For example, a speaker cannot 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 listener's interest and the listener 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 an agent's psychological state.

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.

According to Searle, the illocutionary force should be regarded as the sum of these seven components.

2.2 Distributed Artificial Intelligence

Distributed Artificial Intelligence [5] (DAI) is the subfield of Artificial Intelligence (AI) concerned with concurrency in AI computations, at many levels ([7]). Research in DAI concentrates on understanding the knowledge and reasoning techniques needed for intelligent coordination, and on embodying and evaluating this understanding in computer systems ([25]). Two major arenas may be identified in the field: the first, called *(Cooperative) Distributed Problem Solving* (CDPS), studies how a loosely coupled network of problem solvers can work together to solve problems that are beyond their individual capabilities ([26]), and the second, called *Multi Agent Systems*, is concerned with coordinating behavior among a collection of (possibly pre–existing) autonomous intelligent agents. It is the goal of the field, *i.e.*, realizing in computers many of the social capabilities that we take for granted in humans, that accounts for its interdisciplinary nature, that has attracted researchers coming from a variety of directions; from distributed computing systems to discourse analysis, from formalisms for representing nested beliefs in agents to cognitive studies of human performance in organizations, from solving inherently distributed problems in applications, such as communication network management, to analyzing the evolution of cooperation in populations of artificial systems ([25]).

DAI systems come in "all shapes and colors" but generally they fall into one of four categories: Object–Based Concurrent Programming (OBCP) systems, Blackboard Frameworks, Integrative Systems, and Experimental Test–beds. OBCP systems are the extension of object–oriented systems to DAI systems.[6] The Actor model of concurrent computation ([44], [1]) is the most notable of such approaches and has found its way into systems like MERING IV ([28]). Many DAI systems have been implemented under the shared–memory blackboard framework, in which numerous semantically disparate knowledge sources communicate and synchronize via a common shared data area called a "blackboard" ([58]). GBB ([18]) is an example of a generic shell for blackboard systems and MACE ([37]) is a DAI system built around a distributed blackboard system. Integrative systems are the results of an effort to provide for integration of multiple problem–solving approaches, thus moving away from the single architectural and problem–solving paradigms of the early DAI systems. Contract Net ([20]), ABE ([27]) and MICE ([24]) are attempts at such frameworks. Finally, experimental test–beds, such as DVMT ([51]) allow for controlled experimentation as a foundation for developing DAI systems.

DAI system designers and researchers grapple with seven basic problems: 1) description,

[5] The following are collections of the most influential work in the field: [8], [38], [39], [21], [22], [5] . For an introduction to the issues that DAI is concerned with, see [7] and [35].

[6] An analysis of the special problems raised is offered in [36].

decomposition, distribution and allocation of tasks; 2) communication and interaction languages and protocols; 3) achieving coherent collective behavior; 3) modeling other agents and organized activity; 5) recognizing and resolving inter–agent disparities; 6) implementation languages, frameworks and environments, and 7) methodologies to address practical engineering problems for DAI systems. These problems are inherent to the design and implementation of any DAI system ([7]).

The major problem of interest to our research is that of communication and interaction languages and protocols. One may identify three aspects in this problem: 1) *modes of interaction*, such as negotiation, forum–based discussion, scientific community theory building, marketplace transactions, and so on, that are metaphors for coordination frameworks; 2) *interaction protocols* that are implementations of some mode of interaction and guide the execution of communicative acts, and 3) *communication* or *interaction languages* which can have a highly structured syntax and a small vocabulary with a fixed number of message types or can be more flexible as to facilitate dialogue with rich semantics, that provide the means for the communication of applications. The Contract Net Protocol ([68]), an implementation of negotiation as a metaphor for distributed problem solving ([20]), provides a language and exchange sequences for adaptive task allocation ([20]) and is the best known demonstration of the three aspects of communication mentioned above. Except for Contract Net, there has been little other work on *explicit* communication protocols. *Partial Global Planning* ([23]), a protocol for plans and goals exchange, and *multistage negotiation* ([17]), a negotiation protocol that allows for incremental plan elaboration, are among the very few similar efforts.

2.2.1 DAI and interaction languages

Communication (Interaction) Languages have been a part of DAI since its very early days. They have been considered as part of a broader scheme, that includes the interaction protocol[7] (the bottom template of the communication suite in Figure 1.2) they served and the architecture of the agents that used them. Standard message types, based on the standard architecture of agents were the means of communication in PUP6 ([50]). Similarly, the Contract Net Protocol ([68]) introduced a common inter–node language, which consisted of structured message types, such as task announcements, bids and award message, to be used in the context of a negotiation ([20]) mode of interaction (interaction regime). In both cases, the purpose served by a communication language is to communicate messages, that represent the agents' knowledge and that are to be interpreted in a well–defined manner and cause certain actions on behalf of both sender and receiver.

Almost every DAI application uses such pre–defined message types to achieve communication of knowledge among agents, but very few attempts have been made to provide a communication language that is defined outside the context of a specific architecture and interaction protocol. The focus has been to provide efficient architectures for agents and problem solving strategies for a distributed environment, and thus the communication language is tailored to the application, supplied with only the primitives (message types, or *performatives* in KQML terminology) necessary for the application domain.

[7] An interaction protocol can be described as the policies the agents follow with regard to their interactions with other agents. These policies would, *e.g.*, determine the conditions under which a permission may be issued, a statement may be believed or a request might be acceded (if doing so does not lead to task overload).

The only other research effort with similar intentions to KQML, *i.e.*, to provide an interaction language that is not tailored to a single interaction protocol, or problem–solving strategy or agent architecture, yet is rich enough to accommodate a variety of such, is *Agent Oriented Programming* (AOP), proposed by Shoham ([63]). AOP is actually a comprehensive framework to support multi–agent interactions. In AOP agents are viewed as entities whose state consists of mental components such as beliefs, capabilities, choices and commitments. Although this so–called *intentional* description of a software system is nothing new, AOP introduces a formal language with syntax and semantics to describe the mental states and an interpreted programming language, called AGENT–0,[8] that has semantics consistent with those of the mental states, and in which a programmer can program an agent. A part of AGENT–0 is a communication language that introduces primitives for the interaction of agents. The primitives are speech acts, their semantics are provided in terms of their execution (the communication acts update the belief and the commitment space of an agent), and may be executed conditionally (according to whether certain mental states hold). AGENT–0 is limited in many important ways, for example facts have to be atomic sentences (no conjunction, disjunction, or modal operators allowed) and commitments can only be made for primitive actions (no planning), but AGENT–0 was primarily intended as a prototype, to illustrate the principles of AOP. An attempt to alleviate some of the deficiencies of AGENT–0 was made in PLACA, which includes operators for planning to do actions and achieve goals, but PLACA is also an experimental language and has not reached a production state. Of interest to the KQML research is the work of Davis that introduced AGENT–K,[9] an attempt to bring AOP and AGENT–0 to the KQML universe. AGENT–K, an extension to AGENT–0, is a language in the AOP paradigm that uses KQML for communication. Even with a language like AGENT–K there still remains the issue of cooperation of agents written in the AOP paradigm with agents that do not fall in the AOP paradigm. Finally it remains to be proven the extent to which the AOP paradigm provides a powerful enough framework for serious agent programming.

In conclusion

We navigated through *speech act theory* and *Distributed AI* looking for clues on how to treat communication languages and the semantic issues associated with them, only to find out that communication languages have not been studied in isolation. In speech act theory, the concern is understanding and modeling speech and language as a model for human interaction, and in DAI, the communication language is only an integral (often customized) component of the whole approach. The next chapter describes in more detail the problem we attacked and outlines the approach we decided to follow.

[8] AGENT–0 is only one of the possible programming languages in the AOP paradigm. In AOP there is no "proper" programming language. Extensions or replacements of AGENT–0 may be introduced but they will have to be consistent with the intentional description introduced by the paradigm.

[9] Look at URL http://www.csd.abdn.ac.uk/ pedwards/publs/agentk.html for more details.

Chapter 3

The Intentions and Approach of this Research

Having introduced the background material, we can now lay down the basic concepts behind KQML and outline the problems we are addressing and the approach we follow.

3.1 The basic ideas behind KQML

Next, we introduce KQML and the basic ideas behind it, we address the issue of KQML implementations, and discuss the particular KQML implementation we make reference to in this work.

3.1.1 KQML as a library of communication acts

We view KQML as a communication language for the exchange of information and knowledge between software agents, through the use of a set of message types. Here is an example of a KQML message:

```
(ask-if
          :sender       A
          :receiver     B
          :language     prolog
          :ontology     foo
          :reply-with   id1
          :content      ``bar(a,b)'' )
```

In KQML terminology, *ask-if* is a *performative*[1] (see Table 4.2 for a summary of all KQML performatives); performatives explicitly suggest the illocutionary force. The value of the `:content` slot is an expression in some "computer interpreted" language or another KQML message and represents the content of the communication (illocutionary) act. The other parameters (*keywords*) introduce values that provide a context for the interpretation of the `:content` and at the same time hold information to facilitate the processing of the message. In this example, A is querying B (these are symbolic names for agents[2]), in *Prolog* (the `:language` value), about the truth status of *bar(a,b)*.[3] Any response to this KQML message (illocutionary act) will be identified by *id1* (the `:reply-with` value). The ontology named *foo* may provide additional information regarding the interpretation of the `:content`. Cohen *et al.* ([13]) have objected to the use of the term *performative* as being incorrect from a linguistic point of view and they have suggested using the term *communicative actions* to refer to KQML's primitives, arguing that KQML messages only communicate a desire for some action and in that sense they are *attempts* and not *illocutionary acts*. Although their argument is valid, we will keep using the term because it is still part of the KQML vocabulary, and is a reminder of the linguistic origin of the primitives of KQML.

So, KQML is an abstraction, a collection of *communication* primitives plus the assumptions of a simple model for inter–agent communication and an abstract design for KQML–speaking agents. There is no such thing as an *implementation of KQML*, *per se*, in the sense that KQML is not an *interpreted* or *compiled* language that is offered in some hardware platform or an abstract machine. Agents speak *KQML* in the sense that they use those primitives, this *library of communication acts*,[4] with their intended *meaning*. Despite the above statement regarding KQML implementations, we are going to use the term *KQML implementation* to refer to the pieces of code that facilitate the building of KQML–speaking agents on particular platforms. As a final note, since KQML is only an abstraction, it becomes obvious that a semantic description is of paramount importance; the semantic description, the focus of this research, *is* KQML.

---

[1]The term was first coined by Austin ([4]), to suggest that some verbs can be uttered so that they perform some action. The view that *all* verbs may be considered as performatives eventually dominated the scholarship on the issue.

[2]We will use the term *agents* to indiscriminately refer to all kinds of KQML–speaking programs and applications, throughout this document.

[3]We will consistently use nonsensical expressions for agent names, ontologies and contents, in this document.

[4]I want to credit Benjamin Grosof of T.J. Watson Research Center for suggesting this term to describe KQML.

3.1.2 KQML implementations as frameworks

We explained that KQML is not implemented; when we talk about KQML implementations, we refer to the apparatus necessary for a software application to achieve asynchronous sending and receiving of KQML messages over the network. Although such an implementation can be done in a number of ways there are some abstract requirements that are part of the KQML *model* but not of the implementation itself:

- An agent sends and receives KQML messages, called performatives,[5] that access the knowledge (knowledge base, knowledge store or knowledge content) of the agent.
- KQML–speaking agents may interact *asynchronously* with more than one other agent at the same time.
- The particular network protocol used (SMTP, TCP/IP, HTTP *etc.*), *i.e.*, the transport mechanism for the performatives, is not part of the requirements and we particularly encourage implementations that allow for multiple transport mechanisms.
- Agents are known to one another by their symbolic names, rather than their IP addresses.
- For agents that reside in the same network domain there exists at least one agent with a special status, a *facilitator*, that is particularly suited to process the so–called *networking* and *facilitation* performatives.

Any *KQML implementation* should conform with the above abstract requirements.

3.1.3 An example of a KQML implementation

We subscribe to the logical architecture for a KQML speaking agent shown in Figure 3.1 and based on the KQML implementation developed by *UMBC* and *Lockheed–Martin* [32] . We identify the following 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 process 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 known in advance (as in [42], [19], and [54] where early versions of KQML were used for inter–agent communication) the application programmer 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 agent operates in an environment mostly consisting of *open systems* ([46, 45]) the agent can ask a *facilitator* to appropriately deliver its query; or, 3) the agent can "discuss" the matter with the facilitator or other agents, in order to collect information, 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

[5] We will use `<performative>` to refer to a particular instance of a KQML message that uses a *performative_name*, as in the example in the beginning of Section 3.1, and *performative* to refer to a *performative_name* in the abstract, as in *tell* or *ask-if*, when we want to address its meaning or semantics.

provides *networking* and *facilitation* performatives to support the implementation of all the above mentioned approaches.

Handler Functions and Interface Module. The application programmer has to provide functions (called *handler functions*) that will process the various *performatives*. Every time a KQML message is received, the respective handler function is called, the call takes whatever action necessary (this may include sending a message as a response) and finally the call exits. 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*, the *untell*, or the *deny* performative should be used in such a case. Using those performatives, the application can state either that the expression is true, 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 or she has to know the exact meaning of the various KQML performatives (the *semantics* of the performatives) and how they might be combined with other performatives (the *conversation policies*).

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. A router provides both client and server functions for the application and manages multiple simultaneous connections with other agents.

The *router* and the *handler functions* make an agent KQML–ready; they are an implementation of the shaded template of Figure 1.2. What we labeled as *application* in Figure 3.1 are the rest of the components shown in Figure 1.2.

3.2 The problem and our approach

3.2.1 The problem

The goals of this research can be summarized as follows:

1. To provide a complete specification for KQML that revises, corrects, extends and finalizes the currently available specification ([3]) and specifically to give
   - the syntactic definition and specification for the set of reserved performatives, and
   - the logical architecture of a KQML–speaking agent and of an environment of KQML–speaking agents.
2. To provide a semantic description for the language, in a way that
   - captures all the intuitions about the language, expressed in previous documentation of the language ([29], [2]) and is faithful to the linguistic origins of the language, and
   - does not make commitments to specific system designs and architectures

The need for such work is enforced by the fact that

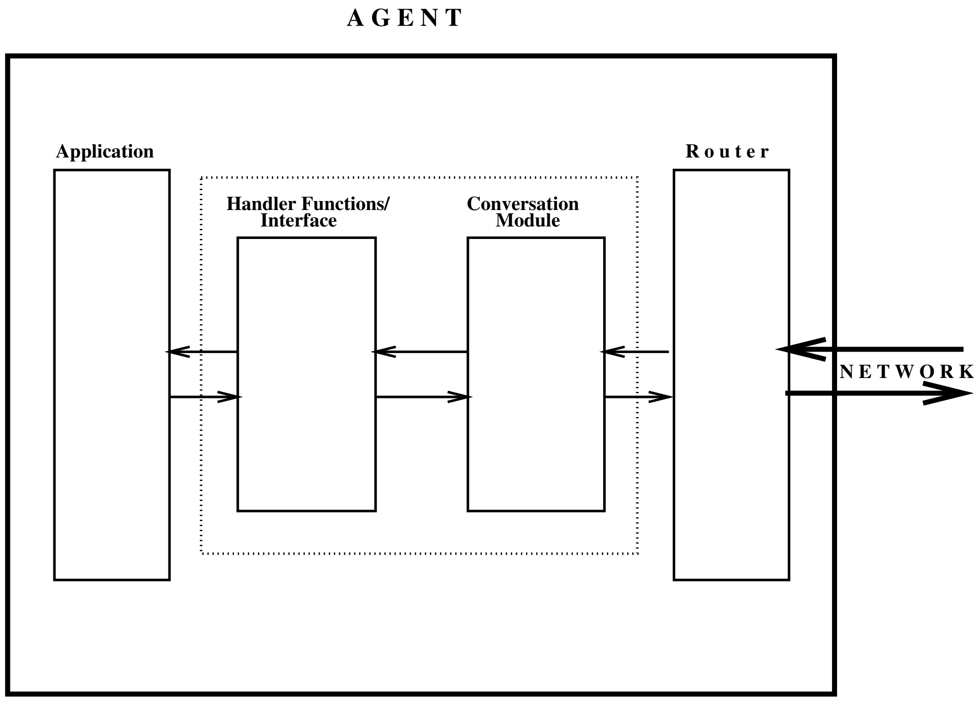

Figure 3.1: Logical architecture of a KQML speaking agent. The *conversation module* is not part of the UMBC/Lockheed–Martin architecture, discussed in Section 3.1.3; it is an addition we propose and discuss in Chapter 7.

- the old KQML specification is not up–to–date with the evolving views of its developers, inaccurate, and with many aspects unspecified, and
- the lack of semantics, for which KQML has been criticized repeatedly, is an impediment to the success and adoption of the language; since there is no implementation (in the pure sense) of KQML, the semantic definition *is* the language.

The benefits (contributions) of such work will be:

- We make KQML a fully specified (syntactically and semantically) language that can be of great use to the research community for communication between (intelligent) software agents, especially since no other such language currently exists, and
- We standardize the language and defining the framework for future extensions, thus solving the problem of the various *ad hoc* interpretations that have appeared in existing applications that use KQML.

Our approach to the problem has three components:

1. A revised specification, and definition of the logical architecture (both for stand–alone agents and groups of agents that "speak" KQML).
2. A semantic description of the languages primitives, based on speech act theory.
3. A formalism for how the communication acts (*performatives*) are combined as building blocks for interactions among KQML–speaking agents (*conversations*).

Since KQML's primitives are viewed as speech acts, speech act theory and semantic approaches for speech acts is one guideline of our research. Distributed Artificial Intelligence, on the other hand, provides us with the most well–defined context for languages that allow communication at the knowledge level ([56],[57]), Next, we describe the approach we follow towards attacking the problem, taking our cues from speech act theory and DAI.

3.2.2 Our approach

We want the semantic approach to provide insights for the use and meaning of the language's primitives, to preserve their speech act origin, to capture all the intuitions expressed about them and the language, and to avoid commitments to particular models and architectures. The approach we propose is based on the following two fundamental assumptions:

- The communication language can be dissociated from the agent model and the interaction regime. Apart from limiting the applicability of the language, such commitments hinder attempts for a formal description of the language (see [66]).
- The primitives of a communication language can be modeled after speech acts. This has been the norm for almost all such languages that appear in the literature, but speech act theory has only been used as an informal framework of the semantics of the primitives.

We treat performatives as speech acts. We adopt the descriptive framework for speech acts, suggested by Searle, as described in [62], [61] and [71], and briefly outlined in Section 2.1.2. We believe that Searle's ideas, despite their linguistic origin and the lack of consideration for the computational aspects, provide the means for distinguishing the various performatives and enable us to identify the attributes that are of interest in an implementationally efficient approach.

The semantic approach we propose is based on the use of epistemic operators to describe the cognitive states of agents. We use them to devise *First Order Predicate Calculus*–like expressions, that suggest the minimum set of preconditions and postconditions that govern the use of a performative, along with conditions that suggest the final state for the successful performance of the speech act (performative); these expressions describe the cognitive (mental) states of the agents involved in an exchange. An integral part of the semantic description is a specification, given in the form of a Definite Clause Grammar, that defines the appropriate sequences of KQML messages in a valid conversation among KQML–speaking agents. The use of epistemic operators to describe mental states of (usually human) agents, in between speech acts, was inspired by a similar use of them in [16] and [9], and is common practice in almost any speech act work. We refrain from referring to them as *modal operators* or *modalities*, even though intuitively they are, because those terms imply the existence of some modal logic framework.

3.2.3 Alternative semantic approaches considered

We decided on the approach outlined in the previous section after considering:

1. An *operational semantics* approach (rather a methodology that covers a variety of approaches than a single approach), as suggested in [10]. Such approaches are often

used for modeling and/or providing the semantics of communication protocols or concurrent processes. Roughly, in an operational semantics approach one would have to define system configurations for KQML agents and then define the performatives as transition relations on these configurations. Such an approach would be problematic because:

- It would be extremely complicated to define configurations for KQML agents since KQML performatives involve the *Virtual Knowledge Bases* of agents, *i.e.*, what can be inferred (rather than what is actually represented in them), especially without making any assumptions about the knowledge representation scheme they use. Also, the fact that an agent may be a collection of software components, instead of a single program or process, would add to the difficulty of the problem. In sort, employing an operational semantic approach would require a simplified and less powerful conceptualization of KQML agents than what the language is designed for.
- It would obscure the speech act flavor of the performatives which is a result of the linguistic orientation of the language. Knowing *which* performative to use, or *when* to use it is important to an application programmer and is more than what transition relations on system configurations may suggest.

2. The *possible worlds* model approach, followed in [67] and [12] (approaches to speech act semantics), that we believe to be inadequate because:

   - It would introduce an unacceptable complexity to the semantic description. The possible worlds model, albeit theoretically sound, is a mathematical construct whose understanding requires a considerable effort which would deter potential users of KQML from using the language.
   - It is doubtful whether it would allow us to capture the fine differences between performatives. Such approaches appear in semantic descriptions of speech acts, from a *Linguistics* and a *Philosophy of Language* perspective and the main effort is to deal with truth and satisfiability. The semantic description is the same for all illocutionary acts of the same illocutionary force, *i.e.*, all "questions" are treated the same way, and so on. In KQML, there are at least four performatives for "querying" and they are not all the same.
   - It would require a formal model of agency that the agents should subscribe to, which can be a very rigid constraint.

Cohen *et al.* have also addressed the issue of semantics for a language like KQML, *i.e.*, for a language with communication primitives modeled after speech acts (*communicative actions*) ([13]). Their approach, based on earlier work of theirs on *intention* and *commitment* ([12]), uses a formalization of those modalities (that define a theoretical model of agency) to provide semantics for the communication primitives. This work is still in progress and it is still in question whether their framework will suffice to provide semantics for all *reserved* KQML performatives. In any case, we view their work as complimentary to the work presented here, which relies on a less "heavy" formalism and includes in its intended audience programmers and developers who choose to use KQML.

Chapter 4

The KQML Specification

This chapter constitutes a revision of the current KQML specification document ([3]). Although the differences regarding the syntax of KQML messages and the reserved performative parameters are minimal, there are significant changes regarding the set of reserved performatives, their meaning and intended use. Parts of Sections 4.1 and 4.2 appear in the current KQML specification document ([3]) and are included here for reasons of completeness of this presentation.

4.1 KQML transport assumptions

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

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

  NOTE: The latter property is less useful than it may appear, unless there is a guarantee of *agent reliability* as well. Such a guarantee is a policy issue, and may vary among systems but it is important (as an assumption) for the semantic description given in Chapter 5.

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

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

4.2 KQML string syntax

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

Unlike Lisp function invocations, parameters in performatives are indexed by keywords and are therefore order independent. These keywords, called *parameter names*, must begin with a colon (:) and must precede the corresponding *parameter value*. Performative parameters are identified by keywords rather than by their position due to a large number of optional parameters to performatives. Several examples of the syntax appear throughout Chapter 4.

The KQML string syntax in BNF is shown in Figure 4.1. The BNF assumes definitions for <ascii>, <alphabetic>, <numeric>, <double-quote>, <backslash>, and <whitespace>. "*" means any number of occurrences, and "-" indicates set difference. Note that <performative> is a specialization of <expression>. In length–delimited strings, *e.g.*, "#3"abc", the whole number before the double–quote specifies the length of the string after the double–quote.

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

Figure 4.1: KQML string syntax in BNF

4.3 Reserved performative parameters

As described in Section 4.2, performatives take parameters identified by keywords. This section defines the meaning of some common performative parameters, by coining their keywords and describing the meaning of the accompanying values. This will facilitate brevity in the performative definitions of Section 4, since those parameters are used heavily.

The following parameters are *reserved* in the sense that any performative's use of parameters with those keywords must be consistent with the definitions below. These keywords and information parameter meanings are summarized in Table 4.1. The specification of reserved parameter keywords is useful in at least two ways: 1) to mandate some degree of uniformity on the semantics of common parameters, and thereby reduce programmer confusion, and 2) to support some level of understanding, by programs, of performatives with unknown names but with known parameter keywords.

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

These parameters convey the actual sender and receiver of a performative, as opposed to the virtual sender and receiver in the :from and :to parameters of a *forward* performative (see Section 4.4.3).

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

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

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

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

Table 4.1: Summary of reserved parameter keywords and their meanings.

The `:content` parameter indicates the "direct object" (in the linguistic sense) of the performative. For example, if the performative name is `tell` then the `:content` will be the sentence being "told". The `<expression>` in the `:content` parameter must be a valid expression in the representation language specified by the `:language` parameter (or KQML in some cases). Figure 4.1 suggests that expressions in the `:content`, that have parentheses (like the Prolog expressions that appear in the examples throughout this chapter) should be enclosed in `<double-quote>`s (" "). Furthermore, the constants used in the `<expression>` must be a subset of those defined by the ontology named by the `:ontology` parameter.

> NOTE: The BNF suggests that both `:language` and `:ontology` are restricted to only take `<word>`s as values, and therefore complex terms, *e.g.*, denoting unions of ontologies, are not allowed. The definitions for `<quotation>` and `<comma-expr>` in Figure 4.1, are intended to accommodate expressions in KIF that use special operators.

4.4 The reserved performatives

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

- When referred to in text, performative names are written in italics, *e.g.*, *ask-all*, *tell*, *etc.*
- In text, we use the names of reserved performative parameters to refer to their values, so `:sender` refers to the particular sender of a performative, `:content` refers to the content and so on.
- Occasionally, we use $\texttt{parameter}_{performative}$ to refer to the value of a particular performative parameter, *i.e.*, $\texttt{sender}_{advertise}$ to refer to the sender of an advertise in a particular case.
- We use `<performative>` to refer to a particular instance of a performative.

The performatives examined in this document are organized in three (3) categories and their meaning and some properties of interest can be found in Table 4.2 (page 28), Table 4.3 (page 29), Table 4.4 (page 30) and Table 4.5 (page 31). The parameters presented with the *performatives'* specifications are mandatory and define the minimum for proper use of the *performative*. Parameters preceded by an asterisk (*) are not always mandatory. For example, the `:in-reply-to` for *ask-if* is mandatory if the *ask-if* follows a relevant *advertise*, but not in other cases. The asterisk itself is not part of the KQML syntax; we only use it as a meta–syntactic marker. Finally, although often some of the values of the parameters can be inferred, we choose completeness over economy.

4.4.1 Discourse performatives

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

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

Agent A sends the following performative to agent B. The `:in-reply-to` suggests that the *ask-all* follows a relevant *advertise* message.

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

and agent B replies with the following KQML message

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

Figure 4.2: An *ask-all* performative and the appropriate response.

The `:sender` wishes to know if the `:content` is true of the receiver. *True* of the `:receiver` is taken to mean that either the `<expression>` matches a sentence in the receiver's Knowledge Base (KB) or is provable of the `:receiver`, *i.e.*, matches a sentence in the receiver's Virtual Knowledge Base (VKB).[1]

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

The `:sender` wishes to know all *instantiations* of the `:content` that are true of the `:receiver`; `<expression>` has free variables that are bound to values in the *instantiations* of the *response*. Those instantiations will be delivered in the form of a collection provided by `:language`. Of course, the notion of the collection is language dependent. In the example in Figure 4.2 (`:language` is *Prolog*) such a collection is just a *list*.

```
(ask-one
```

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

Name	*Page*	Meaning
ask-if	27	S wants to know if the :content is in R's VKB
ask-all	27	S wants all of R's instantiations of the :content that are true of R
ask-one	32	S wants one of R's instantiations of the :content that is true of R
stream-all	32	multiple-response version of ask-all
eos	32	the end-of-stream marker to a multiple-response (stream-all)
tell	34	the sentence is in S's VKB
untell	34	the sentence is **not** in S's VKB
deny	34	the **negation** of the sentence is in S's VKB
insert	35	S asks R to add the :content to its VKB
uninsert	35	S wants R to reverse the act of a previous insert
delete-one	37	S wants R to remove one matching sentence from its VKB
delete-all	37	S wants R to remove all matching sentences from its VKB
undelete	37	S wants R to reverse the act of a previous delete
achieve	38	S wants R to do make something true of its physical environment
unachieve	38	S wants R to reverse the act of a previous achieve
advertise	40	S wants R to know that S can and will process a message like the one in :content
subscribe	42	S wants updates to R's response to a performative
error	44	S considers R's earlier message to be mal-formed
sorry	44	S understands R's message but cannot provide a more informative response
standby	45	S wants R to announce its readiness to provide a response to the message in :content
ready	45	S is ready to respond to a message previously received from R
next	49	S wants R's next response to a message previously sent by S
rest	49	S wants R's remaining responses to a message previously sent by S
discard	49	S does not want R's remaining responses to a previous (multi-response) message
register	50	S announces to R its presence and symbolic name
unregister	50	S wants R to reverse the act of a previous register
forward	51	S wants R to forward the message to the :to agent (R might be that agent)
broadcast	52	S wants R to send a message to all agents that R knows of
transport-address	50	S associates its symbolic name with a new transport address
broker-one	55	S wants R to find one response to a <performative> (some agent other than R is going to provide that response)
broker-all	55	S wants R to find all responses to a <performative> (some agent other than R is going to provide that response)
recommend-one	57	S wants to learn of an agent who may respond to a <performative>
recommend-all	57	S wants to learn of all agents who may respond to a <performative>
recruit-one	57	S wants R to get one suitable agent to respond to a <performative>
recruit-all	59	S wants R to get all suitable agents to respond to a <performative>

Table 4.2: Summary of reserved performatives for :sender S and :receiver R.

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

Table 4.3: This is the set of performatives discussed in this document and their properties when used in conversations. The properties have the following meaning: "response required" means that the `:receiver` processes the performative and generates the response on its own; "response only" means that the performative can only be used in the context of responding to some other performative; "no response" means that those performatives **do not** require (but might allow) a response (there is also the possibility of a follow-up message); and `:content` refers to the type of the `:content` ("n/a" stands for not applicable; see Section 4.4.2 for an explanation). *Forward*, *broadcast*, *broker-one*, *broker-all*, *recruit-one* and *recruit-all*, do not require a response *by default*. Whether there is a response or a follow-up to them, depends solely on the `:content`, *i.e.*, on the `<performative>` that appears in the `:content` and its properties in conversations.

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

Table 4.4: *Advertise, subscribe, standby, forward, broadcast* and the *facilitation performatives* are the only performatives that may have a `<performative>`, *i.e.*, a KQML message, as `:content` ("facilitation *performatives*" refers to *broker-one*, *broker-all*, *recruit-one*, *recruit-all*, *recommend-one* and *recommend-all*). Note that the facilitation performatives allow exactly the same performatives as *advertise*, which makes sense since the processing of the facilitation performatives depends on advertisements. The facilitation performatives may appear in the `:content` of *advertise* messages if and only if a non-facilitator is the `:sender` of the *advertise*.

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

Table 4.5: This table lists the performatives that various kinds of agents may process. We distinguish between agents that are *facilitators* and agents that are not *facilitators*. The categories have the following meaning: "all agents" refers to all agents, whether they serve as facilitators on not; "facilitators only" only applies to agents that are facilitators; and "only if advertised" refers to non-facilitator agents that have to *advertise* for the specific `<performative>`. A subtle distinction has to be drawn between an agent's ability to process a *performative* in principle and to process a `<performative>`, *i.e.*, a KQML message of that *performative* with a particular `:content`. So, for example, although all agents can process *ask-if*, *i.e.*, they have *handler functions* for that performative, they still have to *advertise* their ability to process an *ask-if* with a particular `:content`.

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

This performative is the same as *ask-all* but only one expression is sought as a response. Any of the *tell* performatives of Figure 4.3 would constitute the appropriate response to an *ask-one* message similar to the *ask-all* message of Figure 4.2.

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

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

This performative's meaning is identical to that of *ask-all*, except for the format of the delivery of the response. Instead of delivering the collection of matches in a single performative, a series of performatives, one for each member of the collection, should be sent. This only holds of course, if the response to the corresponding *ask-all* would have been a *tell*. See Figure 4.3 for an example of an exchange that involves the *stream-all* performative and note that the collective response is equivalent to that of Figure 4.2.

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

This performative only serves the purpose of marking the end–of–stream of the multi–response to a *stream-all* (see Figure 4.3).

Agent A sends a message to agent B

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

and agent B replies with the following KQML message

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

and later agent B sends

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

and finally concludes the response with

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

Note that B's response is equivalent to B's single performative response to the similar *ask-all* of Figure 4.2.

Figure 4.3: A *stream-all* performative and the appropriate responses.

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

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

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

This performative indicates that the `:content` expression in not true of the sender, *i.e.*, it is not part of the sender's VKB. This does not necessarily mean that the expression's negation is true of the sender. In other words, $untell_{\texttt{<expression>}}$ is *not* the same as $tell_{\neg\texttt{<expression>}}$.

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

This performative indicates that the **negation** of the `:content` is true of the sender, *i.e.*, it is in the sender's VKB. In other words, $deny_{\texttt{<expression>}}$ is the same as $tell_{\neg\texttt{<expression>}}$.

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

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

The `:sender` requests the `:receiver` to add the `:content` to its KB (see Figure 4.4).

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

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

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

An example that involves *insert* and *uninsert* can be seen in Figure 4.4.

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

Agent A sends the following performative to agent B

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

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

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

and some time later B sends the following message to A

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

which is followed a bit later by

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

Figure 4.4: An *insert* performative following a related *advertise*, and an example of a proper *uninsert*. Note that $\mathtt{reply-with}_{insert}$ is not preset by the `:sender` of the *advertise*.

This performative is a request to delete one sentence from the receiver's KB. The sentence to be deleted is the one that would have been the `:content` of the response if an identical *ask-one* KQML message had been sent and a *tell* performative had been used in the response.

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

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

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

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

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

> NOTE: An *undelete* can only be used after a corresponding *delete-one* or *delete-all*. In either case, it *undeletes* whatever was *deleted* in the first place, assuming of course that the original *delete* action was executed successfully (no *error* or *sorry* was sent as a response).

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

The `:receiver` is asked to want to try to make the `:content` true of the system. Of course this can always be done by just *inserting* the `:content` in the KB, but this performative makes sense when the `:receiver` has a representation of the real world in its KB and the result of the attempt to "make the `:content` true" will be some action in the real world the effect of which will be to modify the respective part of the representation of the real world and thus make the `:content` true in the KB. In other words, the `:content` can be made true only as a result of some action outside of the system, in the physical world. See Figure 4.5 for an example of an exchange that involves the *achieve* performative.

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

This performative is a request to reverse an *achieve* that took place in the past. See Figure 4.5 for an example of an exchange that involves the *unachieve* performative.

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

```
(advertise
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        (performative_name
                                     :sender         <word>
                                     :receiver       <word>
                                     :in-reply-to    <word>
```

Agent A sends the following performative to agent B (the `:in-reply-to` value suggests that B has sent an *advertise* for such an *achieve* message), requesting that B set a new value for the motor torque of `motor1`

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

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

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

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

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

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

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

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

Figure 4.5: An *achieve* performative and the appropriate response, later followed by an *unachieve* request.

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

This performative indicates that the `:sender` commits to process the whole embedded message if the $\texttt{sender}_{advertise}$ receives it (presumably from $\texttt{receiver}_{advertise}$ in the future). The subsequent KQML message ought to be identical to whatever the $\texttt{content}_{advertise}$ is, except for the `:reply-with` value that is going to be set by the `:receiver` of the *advertise*. There are constraints that apply to such a message:

- `performative_name` can be one of `ask-if`, `ask-one`, `ask-all`, `stream-all`, `insert`, `delete-one`, `delete-all`, `achieve` and `subscribe` (or one of the *facilitation performatives* if the `:sender` is not a facilitator; see also Table 4.4).
- $\texttt{sender}_{advertise} = \texttt{receiver}_{performative\_name}$
- $\texttt{receiver}_{advertise} = \texttt{sender}_{performative\_name}$
- $\texttt{reply} - \texttt{with}_{advertise} = \texttt{in} - \texttt{reply} - \texttt{to}_{performative\_name}$

See Figure 4.6 for an example of an exchange that involves the *advertise* performative.

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

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

Agent A sends the following performative to agent B

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

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

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

and agent A responds accordingly, as committed to do

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

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

Figure 4.6: An example of an *advertise* and appropriate follow-ups to that.

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

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

Figure 4.7: An example of an *advertise* of a *subscribe* of a *ask-all*.

This performative is a request to be updated every time that the would–be response to the message in `:content` is different than the last response delivered to the $\texttt{sender}_{subscribe}$. Additionally, since a point of reference is needed for the `receiver` of a *subscribe*, it should issue the first response immediately after receiving the performative and then store the last response in order to compare. We do not need something like an *unsubscribe* performative because a *subscribe* does not affect the VKB, so there is nothing to be undone. If an agent has lost interest to the responses to a prior *subscribe*, a *discard* (see page 49) may be used to inform the other agent. See Figure 4.8 for an example of an exchange that involves the *subscribe* performative.

> NOTE: The `performative_name` in the $\texttt{content}_{subscribe}$ might be any of the performatives that require a response (see Table 4.3).

Agent A sends to agent B the following KQML message, whose `:in-reply-to` tag suggests that is a follow-up to an *advertise* (see Figure 4.7 for this advertise; it is an example of a really long KQML message)

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

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

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

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

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

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

Figure 4.8: A *subscribe* request and appropriate responses.

4.4.2 Intervention and mechanics of conversation performatives

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

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

This performative suggests that the `:sender` received a message, indicated by the value of `:in-reply-to`, that it does not comprehend. The cause for an *error* might be: 1) syntactically ill-formed message, 2) the message has wrong performative parameter values, or 3) it does not comply with the *conversation protocols* (see Chapter 6). This performative can appear as a response to *any* performative, if necessary. See Figure 4.9 for examples of cases that may lead to an *error* performative being sent.

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

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

```
(standby
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
```

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

Normally the `:receiver` of a performative will deliver its response as soon as a response is generated. The *standby* performative that takes a `<performative>` as its content, acts like a modifier on the usual order of affairs. It is a request to the $\texttt{receiver}_{standby}$ to handle the embedded performative as it would normally do, *but* in addition, the `:receiver` should inform the $\texttt{sender}_{standby}$ that it has generated the response and then withhold it until the `:sender` requests for it. In effect, *standby* warns the `:receiver` that the response to the `:content` should not be delivered until the `:sender` of the standby sends an appropriate notification. From the above it is obvious that `performative_name` may be any of the performatives of Table 4.3 that require a response.

> NOTE: In short, *standby* transfers control of the timing of the responses to the `:sender` of the original query, thus reversing the *default protocol*, according to which the `:receiver` delivers its responses at will.

See Figure 4.10 for an example of an exchange that involves the *standby* performative.

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

This performative is used by an agent to announce its readiness to deliver at least one of its responses to a KQML message that has been embedded in a *standby* performative. The use of *standby* does not put the additional constraint on the $\texttt{receiver}_{standby}$ (which is also the $\texttt{sender}_{ready}$) to generate all of its possible responses before announcing its readiness. See Figure 4.10 for an example of an exchange that involves the *ready* performative.

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

Agent B has received the *ask-all* message of Figure 4.2. If B sends either of the following 3 messages as a response to agent A, agent A will respond with an *error*.

Example 1

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

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

Example 2

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

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

Example 3

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

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

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

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

Figure 4.9: Examples of the three situations that may result in an *error*.

Agent A sends a message identical to the *stream-all* of Figure 4.3 but this time a *standby* is used.

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

and agent B this time responds with

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

Then, agent A requests the first response with

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

and finally A delivers

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

Note that the `:in-reply-to` value of the *tell* matches the `reply-with` value of the *stream-all* and not that of the *next*, since *tell* is the response to the *stream-all*. From that point on, a couple of different scenarios are possible (see Figure 4.11).

Figure 4.10: The exchange of Figure 4.3 when *standby* is used.

<u>**Scenario 1**</u>: Agent A requests the second response and B delivers it

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

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

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

<u>**Scenario 2**</u>: Agent A might request for the remaining responses all together

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

in which case the exchange ends with B delivering

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

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

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

Figure 4.11: The possible scenarios that the exchange of Figure 4.10 might continue with (Figure 4.10 shows the exchange of Figure 4.3 when *standby* is used).

This performative is used by an agent that has sent a *standby* in order to request a response from its interlocutor, after the interlocutor (the `receiver` of the *standby*) has announced that it has the response(s) (with the use of *ready*). See Figure 4.10 and Figure 4.11 for an example of an exchange that involves the *next* performative.

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

This performative is to be used by an agent to request for the remaining of the responses, in an exchange that started with a *standby*. In effect, *rest* results to an undoing of the *standby*, since it puts in effect the default protocol where the `:receiver` is in charge of the pace of the conversation and may deliver its responses at will. See Figure 4.10 and Figure 4.11 for an example of an exchange that involves the *rest* performative.

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

This performative indicates to the `:receiver` that the `:sender` is not interested in any more responses (presumably to a multi-response performative). See Figure 4.10 and Figure 4.11 for an example of an exchange that involves the *discard* performative.

> NOTE: Performatives that may result to a multi-response are: *stream-all*, *subscribe*, *recommend-all*.

4.4.3 Networking and Facilitation performatives

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

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

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

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

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

```
(transport-address
        :sender         <word>
        :receiver       <word>
        :reply-with     <word>
        :language       <word>
        :ontology       <word>
        :content        <word>)
```

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

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

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

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

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

See Figure 4.12 and Figure 4.13 for an example of an exchange that involves the *forward* performative.

NOTE: The `:in-reply-to` parameter for *forward* is optional and as far as we know only makes sense in the context of responding to *recommend-one*, *recommend-all*, *broker-one* and *broker-all* in which case the *forward* is a *direct* response to the `<performative>`. In the case of *forward* being used to respond to *broker-one* and *broker-all*, the `:sender` value of the embedded performative is omitted (see Section 5.4.3).

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

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

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

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

Let us consider the following situation: agent C knows of agent A, agent A knows of agent B and agent B knows of agent D ("knows of" is synonymous to "is able to deliver messages to"). Agent C wants agent D to process an *ask-if* for which agent D has advertised its ability and commitment to do so (it is possible for C to know that agent D exists but still not being able to deliver messages to it, *e.g.*, C learned about D after a *recommend-one* message similar to that of Figure 4.15). So, agent C sends the following *forward* message to agent A.

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

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

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

See Figure 4.13 for the continuation of this exchange.

Figure 4.12: A conversation involving the *forward* performative. See Figure 4.13, also.

Continuing the exchange that is shown in Figure 4.12, agent B sends to agent D the following *forward* message.

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

There are two possible scenarios for D upon receiving this last message.
<u>**Scenario 1**</u>: D can deliver directly to A, *i.e.*, D knows of C even though C does not know of D. In this case C sends the following message

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

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

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

that is followed by messages similar to those of Figure 4.12.

Figure 4.13: The rest of the exchange of Figure 4.12.

The constraint is that `performative_name` can be one of the performatives that can be used with *advertise* (see page 40). This is a request to find an agent that *can* and *will* process the `:content`, (*i.e.*, an agent that has sent an *advertise* with such a `:content`) in the name of the receiver of the *broker-one* (so all responses from the third party will be directed to the broker, *i.e.*, the $\texttt{receiver}_{broker-one}$). After receiving the response, the broker will sent it to the `:sender` of the *broker-one*, wrapped in a *forward* originating from the broker-ed agent. See Figure 4.14 for an example of an exchange that involves the *broker-one* performative.

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

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

This performative is a request to find **all** agents that *can* and *will* process the content (similar to *broker-one*). The constraint is again that `performative_name` can be one of those that may be used with advertise (see page 40).

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

Agent facilitator has received an *advertise* message from agent A, identical to the first message in Figure 4.6, except for $\texttt{receiver}_{advertise} = facilitator$ and $\texttt{sender}_{ask-if} = facilitator$). Later, agent C sends the following message to the facilitator

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

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

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

Agent A responds with the following message

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

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

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

The `:from` of the *forward*, which is also the value of the `:sender` of the *tell*, is omitted for reasons that are made clear in the semantic description (see Section 5.4.3).

Figure 4.14: An example of a *broker-one* performative and the follow-up

The constraint is that `performative_name` be one of the performatives that can be used in *advertise* (see page 40). This is a request to suggest an agent that *can* process the `:content` (again, as is the case with *broker-one*, use is made of the *advertise* messages that the $\texttt{receiver}_{recommend-one}$ has received). Since more than just an agent name is needed in order for $\texttt{sender}_{recommend-one}$ to be able to contact this agent, the appropriate response of $\texttt{receiver}_{recommend-one}$ will be to *forward* the *advertise* message that satisfies the request. See Figure 4.15 for an example of an exchange that involves the *recommend-one* performative.

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

The constraint is that `performative_name` can be one of the performatives that can be used in *advertise* (see page 40). This is a request to suggest **all** agents that *can* process the content (similar to *recommend-one*).

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

The constraint is that `performative_name` can be one of the performatives that can be used in *advertise* (see page 40). This performative is like a *broker-one* but responses will be directed back to the issuer of the *recruit-one*. In effect, *recruit-one* is equivalent to

Agent *facilitator* has received an *advertise* message from agent A, identical to the first message in Figure 4.6 (except $\texttt{receiver}_{advertise}$ = *facilitator* and $\texttt{sender}_{ask-if}$ = *facilitator*). Later, agent C sends the following message to the *facilitator*

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

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

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

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

Figure 4.15: An example of a *recommend-one* and a response to it.

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

with the additional constraint that $\texttt{to}_{forward} = \texttt{receiver}_{performative\_name} = X$, where X is to be provided by the $\texttt{receiver}_{forward}$, *i.e.*, the $\texttt{receiver}_{recruit-one}$, making use of the *advertise* performatives known to it (likewise for the $\texttt{in} - \texttt{reply} - \texttt{to}_{performative\_name}$) See Figure 4.16 for an example of an exchange that involves the *recruit-one* performative.

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

The constraint is that `performative_name` can be one of the performatives that can be used in *advertise* (see page 40). This performative is like a *broker-all* but responses will be directed to the issuer of the *recruit-all*. In effect *broker-all* is equivalent to a series of *forward* messages, like those mentioned in the description of *recruit-one*.

Agent *facilitator* has received an *advertise* message from agent A, identical to the first message in Figure 4.6 (except for $\texttt{receiver}_{advertise} = facilitator$ and $\texttt{sender}_{ask-if} = facilitator$). Later, agent C sends the following message to the *facilitator*

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

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

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

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

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

Figure 4.16: An example of a *recruit-one* and its follow-up.

Summary

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

- In each domain of KQML–speaking agents there is at least one agent with a special status called *facilitator* that can always handle the networking and facilitation performatives. Agents advertise to their facilitator, *i.e.*, they send *advertise* messages to their facilitators, thus announcing the messages that they are committed to accepting and properly processing. Advertising to a facilitator is like advertising to the community (either of their own domain or of some other domain). Agents can still advertise on a one-to-one basis, if they so wish, and such advertisements do not commit them to processing messages from agents other than the `:receiver` of the *advertise*. Actually, such advertisements will never be shared with other agents, because of the "personal" nature of the *advertisements*, *i.e.*, they are addressed to particular agents and only *facilitators* can supersede that; see Table 4.5, also. Agents can use their facilitator either
  - to have their queries properly dispatched to other agents, using *recruit-one*, *recruit-all*, *broker-one* or *broker-all*, or
  - to send a *recommend-one* or a *recommend-all* to get the relevant *advertise* messages and directly contact agent(s) that may process their queries.
- Agents can access agents in other domains either through their facilitator, or directly. This implies that a smart facilitator may be built in such a way that whenever it cannot find a useful, relevant *advertise* from an agent in its domain, it may query another facilitator, in some other domain. Such an action initiates a sub–dialogue-b-dialogue with another facilitator in order to serve the original query. Elaborate protocols of this kind are examples of conversations (interactions) that be built on top of the conversation policies presented in Chapter 6.
- Facilitators may request the services of other facilitators in the same way that regular agents may request the services of their facilitator. Facilitators do *not advertise*, not even to other facilitators. The model we imply is one where regular agents *advertise* their services to their facilitators and thus facilitators become providers of *query–processing* information about the agents in their domain; such information can then be accessed by any agent (regular or facilitator), using the *facilitation performatives.*
- We use the term *facilitator* to refer to all kinds of special services that may be provided by *specialized* agents, such as *Agent Name Servers* (ANS), *proxy agents*, or *brokers* ([34]).

Chapter 5

Semantics for KQML Performatives

5.1 Speech act theory as a context for KQML semantics

It is fair to ask whether one can talk about an agent communication language without referring to the properties of agency. Instead of providing a comprehensive definition of agency, we suggest that agents are commonly taken to be "high–level" (*i.e.*, they use symbolic representation, display cognitive–like function, and/or have a belief and/or a knowledge store), and are commonly viewed as having an intentional level description (*i.e.*, their state is viewed as consisting of mental components such as beliefs, capabilities, desires, choices, commitments, *etc.*). We take this "description" (an *intentional* description) to be a helpful *abstract model* for *viewing* software agents, even if their actual implementation does not make claims to such ambitious concepts.[1] Speech act theory provides us with an adequate theory to talk about communication between such agents, and Searle's approach in particular is a very attractive framework.

Searle broke down the illocutionary force into seven components (presented in Section 2.1.2). 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. 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 Chapter 6) and the application programmer.[2] The *mode of achievement* refers to establishing certain relationships between speakers and listeners that make certain illocutionary acts meaningful. The mode of achievement is set

[1] Agents reside at the *knowledge level* ([56, 57]) and cannot therefore be accommodated by languages or protocols that appear in *Distributed Computing* and focus on processes rather than on programs or collection of programs that constitute the agents; the *knowledge level* constitutes a paradigm that resides above processes and data structures.

[2] 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.

by the "organizational" hierarchy or interaction protocol that the agents may use in their interaction. In Contract Net [68], the fact that some agents act as managers and others as potential contractors, creates a context for the negotiation ([20]) , through bidding, that characterizes the protocol. The *preparatory conditions*, for an agent to use a performative, are viewed as *preconditions* on the cognitive state of the agent.

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 the 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 a satisfiability (completion) condition, which indicates the state of affairs *after* the completion of the illocutionary act (performative).

5.2 A framework for the semantics of KQML

We want 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 a *First Order Predicate Calculus* kind of language, to do that. In these expressions we use predicates that stand for *epistemic* operators and have a reserved meaning. 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 ([9]) and Singh's ([67]). 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. In effect, this operators states that P "exists" in the agent's *virtual knowledge base*.
2. **Know**, like the following two operators, refers to the cognitive state of the agents;[3] $know(A, P)$ expresses a state of knowledge awareness of A, about the state 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 and thus is committed to a course of action towards achieving P in the future.

Also, we consider two stages in the handling of a *received* message and we assign predicates to refer to them. First, it is physically *received* (something we implicitly assume throughout the analysis), and then *processed*, in the sense that it is a valid KQML message and will be delivered to the application for processing, *i.e.*, passed to the appropriate *handler function* for processing. The predicates refer to the stages when completed and reference to each,

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

assumes that the prior stages have occurred. Reference to the message being handled is made through the value of the `:reply-with` parameter and we will use M to refer to the message as a whole.

Roughly, *know*, *want* and *intend* stand for the psychological states of knowledge, desire and intention, respectively. For the *bel* predicate only, 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. It is usually the case in the literature that $know(A, P)$ has the meaning $bel(A, P) \wedge P$, *i.e.*, A believes P and P is *true* (in a *model-theoretic* sense). In our analysis, we are not interested in the truth of propositions that are part of the agents knowledge store. We assume that an agent has beliefs about its knowledge store that are true for the agent and might be (or might not be) true for the *world*; an agent believes everything that can be inferred in its knowledge store. Agents are assumed to be experts on their knowledge store and truthful, and they communicate (among other things) their *beliefs* about propositions. So, it is perfectly fine to have $know(A, bel(B, P))$, if $bel(B, P)$ has been communicated to A with some message, under the assumptions we made. Also, $know(A, bel(B,''foo(a, b)''))$ is allowed (if B "speaks" Prolog) but $know(A,''foo(a, b)'')$ (or even $know(B,''foo(A, B)'')$ do not make much sense in our framework. 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 (which 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.

5.3 The semantic description

We will use expressions with the operators described in the previous section to provide the following:

1. A natural language description of the performative's intuitive meaning.
2. An expression that describes the illocutionary act. For all practical purposes, this is a formalization of the natural language description.
3. Preconditions that indicate the necessary state for an agent in order to send a performative (**Pre(A)**) and for the receiver to accept it and successfully process it (**Pre(B)**). If the preconditions do not hold a *error* or *sorry* will be the most likely response.
4. Postconditions that describe the states of both interlocutors after the utterance of a performative (by the sender) and after the receipt (but before a counter utterance) of a message (by the receiver).[4] The postconditions (**Post(A)** and **Post(B)**, respectively) hold unless a *sorry* or an *error* is sent as a *response*.
5. A completion condition for the performative (**Completion**) that indicates the final state, after possibly a conversation has taken place and the intention suggested by the performative that started the conversation, has been fulfilled.

[4]After the receiver replies, a new *cycle* starts.

6. Any comments that we might find suitable to enhance the understanding of the performative.

Table 5.1 summarizes some of the performatives' properties that relate to the semantic description outlined above. If there are non-null preconditions for the receiver, this will mean that the performative can only be some-kind of response to another performative that established those preconditions No preconditions are necessary for the receiver of a performative that starts a conversation (see *advertise*).

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*). The preconditions have to be established somehow and often they can only be established (at least in part) by a communication act (KQML message).

Whenever 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.

Name	*Illocutionary Point*	**Pre(A)**	**Pre(B)**	**Post(A)**	**Post(B)**	**Compl.**
ask-if	directive	Yes	Yes	Yes	Yes	Yes
ask-all	directive	Yes	Yes	Yes	Yes	Yes
ask-one	directive	Yes	Yes	Yes	Yes	Yes
stream-all	directive	Yes	Yes	Yes	Yes	Yes
eos	n/a					
tell	assertive	Yes	Yes	Yes	Yes	Post(B)
untell	assertive	Yes	Yes	Yes	Yes	Post(B)
deny	assertive	Yes	Yes	Yes	Yes	Post(B)
insert	directive	Yes	Yes	Yes	Yes	Post(B)
uninsert	directive	Yes	Yes	Yes	Yes	Post(B)
delete-one	directive	Yes	Yes	Yes	Yes	Post(B)
delete-all	directive	Yes	Yes	Yes	Yes	Post(B)
undelete	directive	Yes	Yes	Yes	Yes	Post(B)
achieve	directive	Yes	Yes	Yes	Yes	Post(B)
unachieve	directive	Yes	Yes	Yes	Yes	Post(B)
advertise	commisive	Yes	No	Yes	Yes	Post(B)
subscribe	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
error	assertive	Yes	No	Yes	Yes	Post(B)
sorry	assertive	Yes	No	Yes	Yes	Post(B)
standby	n/a					
ready	n/a					
next	n/a					
rest	n/a					
discard	n/a					
register	directive	Yes	Yes	Yes	Yes	Post(B)
unregister	directive	Yes	Yes	Yes	Yes	Post(B)
forward (for `:from`, `:to`)	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
forward (for `:sender`, `:receiver`)	directive	Yes	No	No	Yes	No
broadcast	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
transport-address	directive directive	Yes	Yes	Yes	Yes	Yes
broker-one	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
broker-all	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
recommend-one	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
recommend-all	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
recruit-one	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`
recruit-all	directive	`<p>`	`<p>`	`<p>`	`<p>`	`<p>`

Table 5.1: These are properties of the performatives that relate to their semantic description. "N/a" stands for not applicable, "`<p>`" stands for `<performative>`, *i.e.*, the respective conditions are those of the `<performative>` in the content, and "Post(B)" refers to the postconditions for the receiver.

5.4 Semantics for KQML performatives

We first introduce the notation that we will use for the semantic description of the performatives. **A** is the `:sender`, **B** is the `:receiver` and **X** is the `:content` of the performative (KQML message). From a *speech act* point of view, **A** is the speaker, **B** is the listener and **X** is the propositional content of the speech act that takes place. Occasionally we use **M** to refer to an instance of a KQML message. Capital case letters stand for universally quantified variables; we will use letters in the beginning of the alphabet (*e.g.*, A, B, *etc.*) for agents' names and letters towards the end of the alphabet (*e.g.*, X, Y,Z) for propositional contents of performatives. All underscores (_) are unnamed, universally quantified variables (they stand for performative parameters that do not have values in the KQML message). We will use capital case letters preceded by a question mark (?), *e.g.*, $?B$, for existentially quantified variables.

All expressions consisting of the epistemic operators of Section 5.2 and the various predicates introduced in this chapter stand for mental states of agents. We allow conjunctions (we will use $\wedge$) and disjunctions (we will use $\vee$) of expressions that stand for mental states (the resulting expressions represent mental states, also), but we do not allow $\wedge$ and $\vee$ in the scope of the epistemic operators. A negation (we will use $\neg$) of a mental state is taken to mean that the mental state does not hold. When "$\neg$" qualifies the **bel** predicate, *e.g.*, $\neg(bel(A,X))$, it is taken to mean that the `:content` expression X is not true for agent A, *i.e.*, it is not provable in A's knowledge base. Negation may also be used to mean the logical negation (in a model theoretic sense) of a `:content` expression, *e.g.*, $\neg(X)$. We use the same symbol for all three different uses of negation.

All expressions mentioned as preconditions, postconditions and completion conditions are the minimum necessary for our specification of KQML.

5.4.1 Discourse performatives

- **ask-if(A,B,X)**
  1. A wants to know what B believes regarding the truth status of the content X.
  2. $want(A, know(A,Y))$
     where Y may be any of $bel(B,X)$, $bel(B,\neg(X))$, or $\neg(bel(B,X))$
  3. **Pre(A)**: $want(A, know(A,Y)) \wedge know(A, intend(B, process(B,M)))$
     where M is **ask-if(A,B,X)**
     Pre(B): $intend(B, process(B,M))$
  4. **Post(A)**: $intend(A, know(A,Y))$
     Post(B): $know(B, want(A, know(A,Y)))$
  5. **Completion**: $know(A, Y_i))$
     where Y_i is one of the Y mentioned above.
  6. Not believing something is not necessarily the same with believing its negation, although this may be the case for certain systems. The **Pre(A)** and **Pre(B)** suggest that a proper advertisement is needed to establish them (see *advertise* on page 72); this is the case with many other performatives.
- **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.
2. $want(A, know(A, Z))$
where Z may be:(1) $bel(B, Y)$, where Y is the finite collection of all possible Y_1, Y_2, ..., where each Y_i is an instance of X with values for the variables in X and each Y_i appears once in this collection (the collection might be empty), or (2) $\neg(bel(B, X))$, or finally, (3) $bel(B, \neg(X))$.
3. **Pre(A)**: $want(A, know(A, Z)) \wedge know(A, intend(B, process(B, M)))$
where M is **ask-all(A,B,X)**.
Pre(B): $intend(B, process(B, M))$
4. **Post(A)**: $intend(A, know(A, Z))$
Post(B): $know(B, want(A, know(A, Z)))$
5. **Completion**: $know(A, Z_i)$
where Z_i is one of the possible Z mentioned above.
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 to use *ask-all* when X has free variables. An *ask-if* with content *foo(X,b)* makes perfect sense (for Prolog–speaking agents), if one wants to know if there exists X such that $foo(X, b)$ is true. But if the same expression is used with an *ask-all*, one expects something like $[foo(a, b), foo(c, b)]$. 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).

- **ask-one(A,B,X)** Everything said about *ask-all* holds for *ask-one* also, but in this case A is interested in receiving exactly *one* response although there might be more than one responses to its query.

- **stream-all(A,B,X)** Everything mentioned for *ask-all* holds for *stream-all*, too. The only difference is in the expected delivery format of the response. Either because the sender cannot (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 fact for B).

- **eos(A,B,M)** This performative is somewhat unusual with respect to the other performatives mentioned because its only purpose is to notify B that there are no more responses to a *positive* response to a *stream-all*. This performative is just an end–of–stream marker.

- **tell(A,B,X)**

1. A states to B that A believes the content to be true.
2. $bel(A, X)$
3. **Pre(A)**: $bel(A, X) \wedge know(A, want(B, know(B, bel(A, X))))$
Pre(B): $intend(B, know(B, bel(A, X)))$
4. **Post(A)**: $know(A, know(B, bel(A, X)))$
Post(B): $know(B, bel(A, X))$

5. **Completion**: $know(B, bel(A, X))$
6. The completion condition holds, unless a *sorry* or *error* suggests B's inability to acknowledge the *tell* properly, as is the case with any other performative (see Section 5.4.2 for more details).

- **untell(A,B,X)**

1. A states to B that A believes the negation of the content to be true.
2. $\neg(bel(A, X))$
3. **Pre(A)**: $\neg(bel(A, X)) \wedge know(A, want(B, know(B, \neg(bel(A, X)))))$
Pre(B): $intend(B, know(B, \neg bel(A, X)))$
4. **Post(A)**: $know(A, know(B, \neg(bel(A, X))))$
Post(B): $know(B, \neg(bel(A, X)))$
5. **Completion**: $know(B, \neg(bel(A, X)))$
6. The completion condition holds, unless a *sorry* or *error* suggests B's inability to acknowledge the *tell* properly. The semantic description of *untell* is similar to that of *tell*, if $bel(A, X)$ is replaced with $\neg(bel(A, X))$.

- **deny(A,B,X)**

1. A states to B that A believes the negation of the content to be true.
2. $bel(A, \neg(X))))$
3. **Pre(A)**: $bel(A, \neg(X)) \wedge know(A, want(B, know(B, bel(A, \neg(X)))))$
Pre(B): $intend(B, know(B, bel(A, \neg(X))))$
4. **Post(A)**: $know(A, know(B, bel(A, \neg(X))))$
Post(B): $know(B, bel(A, \neg(X)))$
5. **Completion**: $know(B, bel(A, \neg(X)))$
6. It is the case that **deny(A,B,X)** $\equiv$ **tell(A,B,¬(X))**.

- **insert(A,B,X)**

1. A wants B to believe the content (in other words, to make the content a part of its knowledge store).
2. $want(A, (bel(B, X)))$
3. **Pre(A)**:$know(A, intend(B, process(B, M)))$
where M is **insert(A,B,X)**
Pre(B): $intend(B, process(B, M)) \wedge \neg(bel(B, X))$
4. **Post(A)**: $know(A, bel(B, X))$
Post(B): $bel(B, X)$
5. **Completion**: $bel(B, X)$
6. The content X should be a ground expression. Note that it is not *necessarily* the case that $bel(A, X)$. Also, note that $bel(B, X)$ refers to X's VKB, *i.e.*, what might already exist in the KB or *inferred*. This means that it is not enough for B to add the expression X in its KB, but the agent also has to make sure that such an action does not create a conflict with what might be inferred in its VKB.

Agents will either use *insert* along with some *truth maintenance* mechanism, or will only *advertise* for *insert* messages that have a `:content` that cannot be possibly inferred in their KB. Similar arguments apply to *uninsert*, *delete-one*, *delete-all* and *undelete*.

- **uninsert(A,B,X)**

  1. A wants B to remove X from its knowledge store, where X was inserted to B's knowledge store upon A's request in the past.
  2. $want(A, \neg(bel(B, X)))$
  3. **Pre(A)**: $know(A, process(B, M))$
     where M is **insert(A,B,X)**.
     It is always an implicit postcondition that $know(A, process(B, M_i))$, for any M_i that A sent to B, **unless** a *sorry* or *error* is sent in response to M. In other words, a message is always delivered to its destination (one of the assumptions mentioned in Chapter 4) and the sending agent knows that.
     Pre(B): $process(B, M)) \wedge bel(B, X)$
     $bel(B, X)$ suggests that B is asked to remove something that is in the database already. Since M was processed (as implied by **Pre(A)**, *i.e.*, no *error* or *sorry* was sent from B to A as a response to M, it must be the case that $bel(B, X)$, which is the **Completion** for M. In a sense, explicitly mentioning this precondition is redundant since processing a message leads to the satisfiability of its postconditions; we include it in order to establish that no *error* or *sorry* had been the response to the related *insert*, *i.e.*, the *insert* was indeed processed by B (same argument applies to **Pre(A)**).
  4. **Post(A)**: $know(A, \neg(bel(B, X)))$
     Post(B): $\neg(bel(B, X))$
  5. **Completion**: $\neg(bel(B, X))$
  6. Notice the similarity to *delete-one* but with different preconditions that suggest that B has processed an *insert* in the past.

- **delete-one(A,B,X)**

  1. A states to B that A wants B to remove the content from its knowledge store.
  2. $want(A, \neg(bel(B, X)))$
  3. **Pre(A)**: $know(A, intend(B, process(B, M)))$
     where M is **delete(A,B,X)**.
     Pre(B): $intend(B, process(B, M)) \wedge bel(B, X)$
     $bel(B, X)$ suggests that B is asked to remove something that is in the database (or can be inferred in it) already. As is the case with *insert*, if the request cannot be satisfied, an *error* or a *sorry* will be sent as a *response*.
  4. **Post(A)**: $know(A, \neg(bel(B, X)))$
     Post(B): $\neg(bel(B, X))$
  5. **Completion**: $\neg(bel(B, X))$
  6. Note that it is not *necessarily* the case that $\neg(bel(A, X))$.

- **delete-all(A,B,X)**

1. A wants B to remove the content from its knowledge store.
2. $want(A, \neg(bel(B, X)))$
3. **Pre(A)**: $know(A, intend(B, process(B, M)))$
   where M is **delete(A,B,X)**
   Pre(B): $intend(B, process(B, M)) \wedge bel(B, X)$
   (see the comment in **Pre(B)** for *delete-one*)
4. **Post(A)**: $know(A, \neg(bel(B, X)))$
   Post(B): $\neg(bel(B, X))$
5. **Completion**: $\neg(bel(B, X))$
6. The semantic description does not differ from that of *delete-one*, but for *delete-all*, X has to have at least one free variable.

- **undelete(A,B,X)**

1. A wants B to restore X in its knowledge store, where X was removed from B's knowledge store upon A's request in the past.
2. $want(A, (bel(B, X)))$
3. **Pre(A)**: $know(A, process(B, M))$
   where M is **delete(A,B,X)**.
   The reasoning is similar to that made about **Pre (A)** for *uninsert*.
   Pre(B): $process(B, M) \wedge \neg(bel(B, X))$
   The reasoning is similar to that made about **Pre (B)** for *uninsert*.
4. **Post(A)**: $know(A, bel(B, X))$
   Post(B): $bel(B, X)$
5. **Completion**: $bel(B, X)$
6. Note that it is not *necessarily* the case that $bel(A, X)$)

- **achieve(A,B,X)**

1. A wants B to take some action that will make B true in its VKB (see also page 38).
2. $want(A, bel(B, X))$
   $bel(B, X)$ can only be the result of some action outside A and presumably X is a representation of some entity or action in the real world.
3. **Pre(A)**: $know(A, intend(B, process(B, M)))$
   where M is **achieve(A,B,X)**
   Pre(B): $intend(B, process(B, M)) \wedge \neg(bel(B, X))$
4. **Post(A)**: $know(A, bel(B, X))$
   Post(B): $bel(B, X)$
5. **Completion**: $bel(B, X)$
6. The semantic description is the same with that of *insert* but in this case there are additional requirements for the content X.

- **unachieve(A,B,X)**

1. A wants to undo the effects of an achieve that A sent to B in the past.

2. $want(A, \neg(bel(B,X)))$
   $bel(B,X)$ can only be the result of some action outside A and presumably X is a representation of some entity or action in the real world.
3. **Pre(A)**: $know(A, process(B,M))$
   where M is **achieve(A,B,X)**. The reasoning is similar to that made about **Pre(A)** for *uninsert* or *undelete*.
   Pre(B): $process(B,M)) \wedge bel(B,X)$
   The reasoning is similar to that made about **Pre (B)** for *uninsert* or *undelete*.
   Post(A): $know(A, \neg(bel(B,X)))$
   Post(B): $\neg(bel(B,X))$
4. **Completion**: $\neg(bel(B,X))$
5. The semantic description is the same with that of *uninsert* but in this case there are additional requirements for the content X.

- **advertise(A,B,M)**

  1. A wants B to know that A can and will process a message like M from B, if it receives one (A commits himself to such a course of action).
  2. $want(A, know(B, intend(A, process(A,M))))$
     where M is a KQML message **performative_name(A,B,X)**.
  3. **Pre(A)**: $intend(A, process(A,M))$
     Pre(B): NONE
  4. **Post(A)**: $know(A, know(B, intend(A, process(A,M))))$
     Post(B): $know(B, intend(A, process(A,M)))$
  5. **Completion**: $know(B, intend(A, process(A,M)))$
  6. An *advertise* is a commisive act, in the sense that it commits its sender to process M, as suggested by the announcement of the intention to process. If B is a *facilitator* then B is interchangeable (in the semantic description) with the name of any agent the facilitator knows about.

- **subscribe(A,B,M)**
  A wants to be updated every time that the *response* to the message M is different to the previous such response. The semantics of this message are those of M, but additionally every time that B responds to M, it is as if M is sent again and B processes M but delivers the *response* if and only if it is different than the previous one. *subscribe* is like an infinite loop of processing M with a response sent if and only if being different than the last generated response.

  Furthermore the part of the preconditions that relate to the intention to process, refer to **subscribe(A,B,M)** (let us call it M_0) and not to M. So, the preconditions should have M_0 in place of M. In other words, if an agent sends **advertise(A,B,ask-if(B,A,X))** this does not mean that an agent has committed to process **subscribe(B,A,ask-if(B,A,X))**. For that to happen, an **advertise(A,B,subscribe(B,A,-ask-if(B,A,X)))** would have to be sent.

5.4.2 Intervention and mechanics of conversation performatives

Of the performatives in this category, we treat only *error* and *sorry* as speech acts. We introduce the following predicate:
sendMSG, as in *sendMSG(A,B,M)*, for the state of "A sending message M to B."

- **error(A,B,Id)**
  1. A states to B that A did not process the KQML message M identified by the `:reply-with` value Id.
  2. $\neg(process(A, M))$
  3. **Pre(A)**: $receive(A, M) \wedge \neg(process(A, M))$
     Pre(B): $sendMSG(B, A, M)$
  4. **Post(A)**: $know(A, know(B, \neg(process(A, M)))) \wedge \neg(Post_M(A))$
     where $Post_M(A)$ is **Post(A)** for message M.
     Post(B): $know(B, \neg(process(A, M))) \wedge \neg(Post_M(B))$
  5. **Completion**: $know(B, \neg(process(A, M))) \wedge \neg(Completion_M(A))$
  6. An agent might respond with an *error* if either it 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. The postconditions and completion conditions for M do not hold.

- **sorry(A,B,Id)**
  1. A states to B that although it processed the message, it has no (possibly further) response to provide to the KQML message M identified by the `:reply-with` value Id.
  2. $process(A, M)$
  3. **Pre(A)**: $process(A, M)$
     Pre(B): $sendMSG(B, A, M)$
  4. **Post(A)**: $know(A, know(B, process(A, M))) \wedge \neg(Post_M(A))$
     Post(B): $know(B, (process(A, M)) \wedge \neg(Post_M(B))$
  5. **Completion**: $know(B, process(A, M)) \wedge \neg(Completion_M(A))$
  6. The postconditions and completion conditions do not hold, even though A dispatched the performative to the appropriate handler function, because A could not (or did not want) to come up with a response that would result to their satisfiability.

The rest of the performatives of this category are not speech acts. Their sole purpose is to alter the normal course of a conversation (one of the assumptions of the KQML model) but this is an issue of pragmatics rather than semantics. We can view them as *signals* that do not affect the semantics of the performatives they employ. For example, the semantics of a *standby* are identical to those of the embedded performative, but *standby* overturns the *default protocol* and puts into effect the protocol described in Section 4.4.2. For the semantics of those performatives, we give *finite state machines* that show how the default protocol for a single–message response (Figure 5.1) changes when the performatives

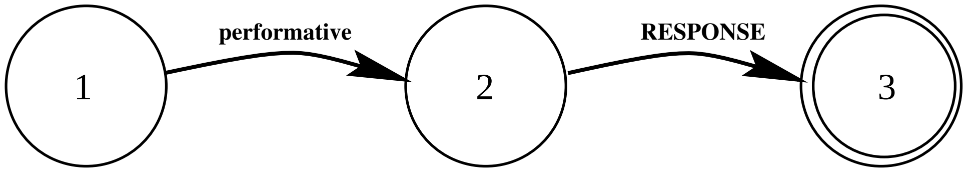

Figure 5.1: A Finite State Machine for the default protocol (single response). Messages referenced in capital–case letters belong to the interlocutor of the agent that initiates the conversation .

of this group are used (Figure 5.2). The *standby*, *next*, *rest* and *discard* performatives are only used by the agent that starts the conversation, since this is now the agent in control of the pace of the conversation, and *ready* is only used by its interlocutor. In a sense, the intuitive meaning of those performatives is irrelevant to their proper use. As a matter of fact, they may be handled automatically by the routers of the two interlocutors and the application programmer need not worry about them. If the agent cannot handle asynchronous interaction, its router can embed the outgoing queries in a *standby*. Similarly, for an incoming message, it can be dispatched to the handler function, and when the handler exits, the router can send a *ready* and wait before sending the *actual* response is sent. Figure 5.3 and Figure 5.4 (page 76), show the default and altered protocol for a *multi*–response, respectively.

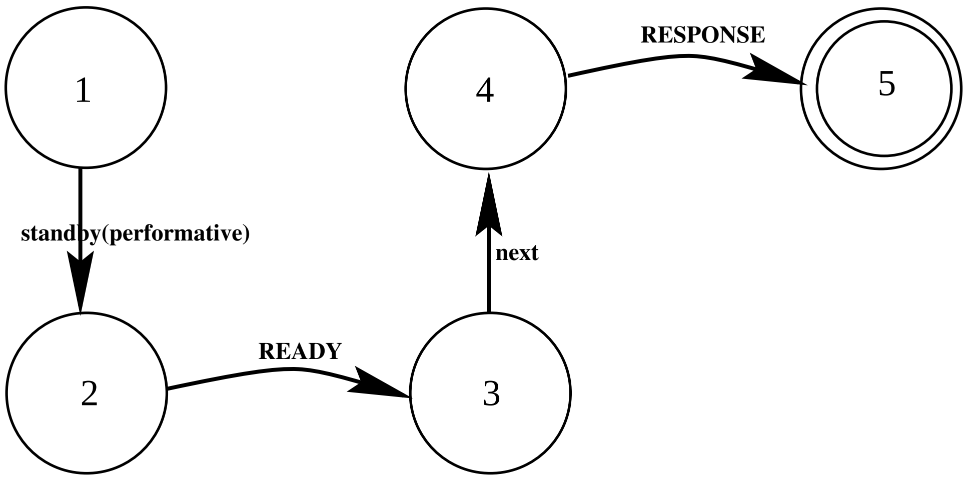

Figure 5.2: A Finite State Machine for the altered protocol, when *standby* is used (single response). Messages referenced in capital–case letters belong to the interlocutor of the agent that uses *standby*.

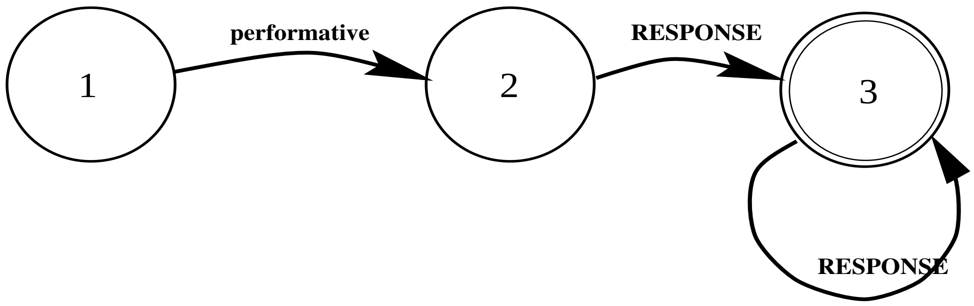

Figure 5.3: A Finite State Machine for the default protocol (multi–response). Messages referenced in capital–case letters belong to the interlocutor of the agent that initiates the conversation.

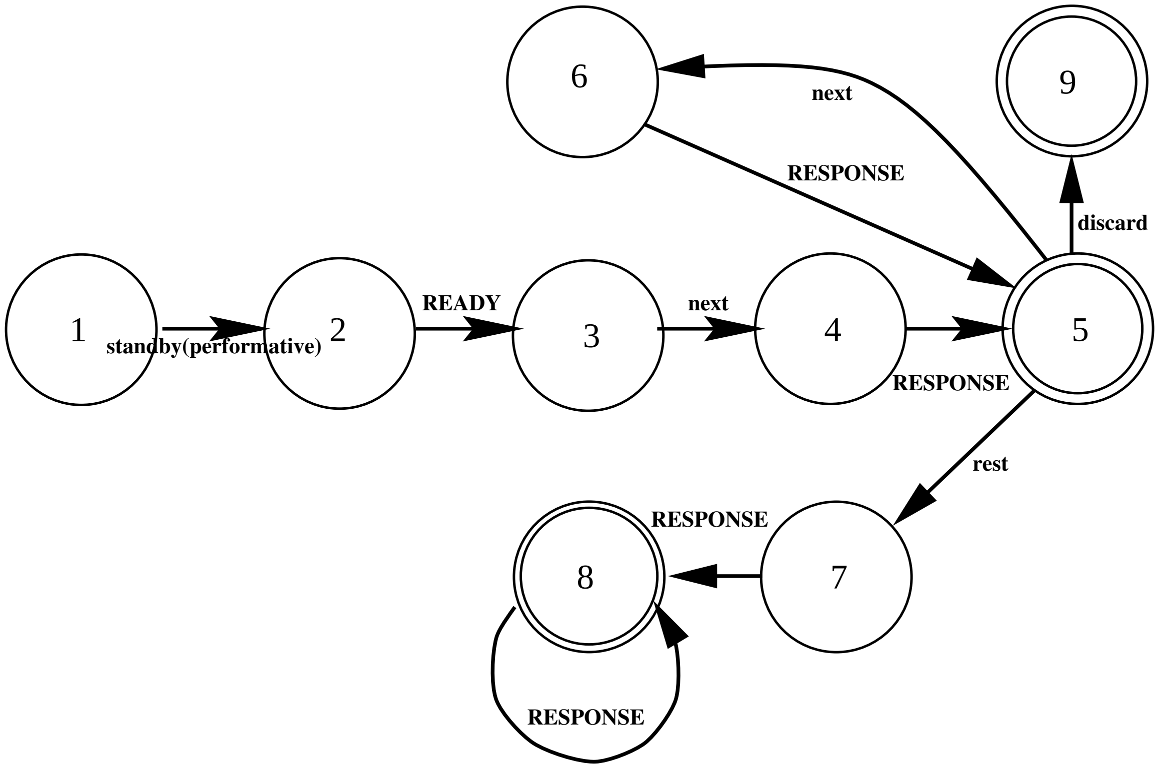

Figure 5.4: A Finite State Machine for the altered protocol, when *standby* is used (multi–response). Messages referenced in capital–case letters belong to the interlocutor of the agent that uses *standby*.

5.4.3 Networking and facilitation performatives

- **register(A,B,X)**
  This performative is just a shorthand for (and thus has the same semantic description as) **insert(A,B,X)**, where X is a list of properties of agent A, such as its physical address, e-mail address, symbolic name, network protocols that it can process, available proxy agents for A, *etc.* This performative is to be processed only by a *facilitator*, and given the special status of the facilitator, the *advertise*–related **Pre(A)** and **Pre(B)** are automatically established, *i.e.*, a facilitator need not advertise that it is going to process such an *insert*. The `:language` for this performative is KQML and the `:ontology` is the ontology of (KQML–speaking) agents. Work is being done to define such an ontology. The router of B (B can only be a *facilitator*), automatically captures the physical address of A, so the physical address mentioned in X may not be different from the physical address the message was sent from. In other words, an agent c an only *register* from its actual address, which is also the address linked to the symbolic names it is using.

- **unregister(A,B)**
  This is a shorthand for **uninsert(A,B,X))**, where X is the `:content` of the **register(A,B,X)** that A has been sent from the *same* physical address.

- **transport-address(A,B,X)**
  Let us assume that A has sent in the past **register(A,B,Y)** and that Y is a similar `:content` to the X in the *register* above. Then, *transport-address* signals the relocation of A in the network, to a new physical address and semantically, this act is like a *unregister* (**unregister(A,B,Y)**) from the old address followed by a *register* (**register(A,B,X)**) from the new address.

  > NOTE: Although it is beyond the scope of the semantics, we should note that a *transport-address* has to be a secure and authenticated message exchange. See [70] for a study on how to add security to KQML transactions.

For the rest of the performatives, we introduce three new predicates:

1. **canDeliver**, as in *canDeliver(A,B)*, for the state of "A being able to deliver a message (any message) to B." *canDeliver(A,B)* holds, but not exclusively, if a *register(B,A,X)* has been sent from B to A.

2. **canProcess**, as in *canProcess(A,M)*, for the state of "A being able to process message M." It is always the case that if **advertise(A,B,M)** then *canProcess(A,M)*, but it could very well be the case that *canProcess(A,M)* may be inferred in other ways.

- **forward(A,B,C,D,performative(A,B,X))**
  A, B, C, D stand for the `from`, `:to`, `:sender`, `:receiver` agents respectively and **performative(A,B,X)** is a complete KQML message. Furthermore, it should be true that $\neg((A = C) \wedge (B = D))$, *i.e.*, **forward(A,B,A,B,performative(A,B,X)))** is not a valid KQML message, since A can simply send a **performative(A,B,X)** message.

This performative actually involves at least three parties: the $\texttt{sender}_{performative}$ (and also $\texttt{from}_{forward}$ and maybe the $\texttt{sender}_{forward}$), the $\texttt{receiver}_{performative}$ (and also $\texttt{to}_{forward}$ but definitely not the $\texttt{receiver}_{forward}$, if $\texttt{from}_{forward} = \texttt{sender}_{forward}$)and finally the $\texttt{receiver}_{forward}$.

As far as the first two parties are concerned, *i.e.*, A and B, a **forward(A,B,C,D,performative(A,B,X)))** is the same as a **performative(A,B,X)**, from a semantics point of view, and the *forward* is a wrapper that provides for the delivery and successful transport of **performative(A,B,X)** from A to B. The response of B might be delivered either directly to A (this is the first option) or via a *forward* (see also page 89).

For C and D, semantically speaking, the situation is different.

1. C wants D to send a *forward* message either to B or to some other agent that can hopefully reach B.
2. $want(C, sendMSG(D, forward(A, B, D, B, performative(A, B, X))))$
3. **Pre(C)**:
   $want(C, sendMSG(D, forward(A, B, D, B, performative(A, B, X))))$
   $\wedge \neg canDeliver(C, B)$
   Pre(D): NONE
4. **Post(C)**: NONE
   Post(D):
   $sendMSG(D, forward(A, B, D, B, performative(A, B, X))) \vee$
   $sendMSG(D, forward(A, B, D, E, performative(A, B, X)))$,
   if there exists an agent E for which $bel(C, canDeliver(E, B))$.
5. **Completion(C)**: NONE
6. A significant point must be made here. There are no **Post(A)** and no **Completion** conditions, meaning that the $\texttt{sender}_{forward}$ has no guarantee that **performative(A,B,X)** will be delivered to B wrapped in a *forward*. This happens because in the case that D cannot reach B directly, D might (or might not, if not possible) send a *forward* request to some other agent E, that D believes (without any certainty of course) that can reach B either directly or indirectly. The reason for all that is the fact that agents in general do not advertise to what agents they can deliver, but there is the possibility of inferring such capabilities, for example by inspecting previous messages exchanged, that an agent happens to know about.

- **broadcast(A,B,performative(A,_,X))**
  This is semantically equivalent to **forward(A,B,A,C,performative(A,C,X))** for every agent C that $canDeliver(B, C)$. Of course, A does not know in advance who its conversants are; that is to be decided by B upon receipt of the *broadcast* message.

- **broker-one(A,B,performative(A,_,X)**
  Let D be an agent such that *canProcess(D,performative(B,D,X))* (this is to be provided or inferred by B). Then B sends **performative(B,D,X)** to D, receives some *response* (depending on the *performative*) from D, let us call it **response(D,B,X)**, and then B sends to A the message **forward(B,A,response(_,A,X))**.

Semantically this is a three–party situation. For B and D the semantics are those of **performative(B,D,X)**, meaning that as far as D knows of, the exchange has the meaning and repercussions of **performative(B,D,X)** (and whatever additional responses) being exchanged between B and D.

For A and B, the semantics are **not** those of a **performative(A,B,X)**, meaning that A is aware that whatever *response* comes from B is merely an "echo" of the utterance of D. So, for example, if the `:content` of the *broker-one* was **ask-if(A,_,X)**, A understands that the *response* **forward(B,A,tell(_,A,X))** does not imply that *bel(B,X)*, since D's response to B is wrapped in a *forward* and then sent to A. It is important to note though, that A does not know the identity of D, that B knows of, and only knows that such an agent D, that provided the response, exists. For A and B the *broker-one* is only a request to send the `:content` to D, as if sent by B and *forward* the *response*, coming from D, to A.

For A and D the semantics are those of **performative(A,D,X)** (let us call it M) but with the major difference that $Pre_M(D)$ and $Post_M(D)$ do not hold because D does not know that it has this exchange with A, instead believes that the exchange occurs with B. Additionally, all the conditions that apply to A lack the name of D, *e.g.*, in the case of the exchange mentioned above, eventually $know(A, bel(?D, X))$, meaning A knows that there is some agent that *believes* X but A does not know who that agent is, even though B does.

So, *broker-one* may be thought as an *anonymous* version of *recruit-one* (see below), in the sense that the *knowledge provider*, remains anonymous to the `:sender` of the *broker-one* and is only known to the *mediator* of the exchange, the `:receiver` of the *broker-one*.

- **broker-all(A,B,performative(A,_,X)** This performative is like multiple *broker-one* messages for every agent D, such that $canProcess(D, performative(A, B, X))$.

- **recommend-one(A,B,performative(A,_,X))**
 This is a request from A to have B *forward* to it one relevant *advertise* message from some agent D that suggests that $canProcess(D, performative(A, D, X))$, *i.e.*,
 $want(A, sendMSG(B, forward(D, A, B, A, advertise(D, A, Y))))$
 where Y is $performative(A, D, X)$
 Remember that if the `:receiver` of an *advertise* is a *facilitator* the `:receiver` value is interchangeable with the name of any agent that has sent a *register* to the *facilitator*.

- **recommend-all(A,B,performative(A,_,X))**
 This performative is like *recommend-one*, but
 $want(A, sendMSG(B, forward(D, A, B, A, advertise(D, A, Y))))$,
 where Y is $performative(A, D, X)$, for every agent D that has sent **advertise(D,-B,Y)**.

- **recruit-one(A,B,performative(A,_,X))**
 From a semantics point of view this is the same to a **forward(A,C,A,B,performative(A,C,X)** where C is an agent that at some point has sent a **advertise(C,B,performative(B,C,X)** (C has sent to B an *advertise* of X). We remind the reader (see also page 40) that advertising to a *facilitator* implies advertising to all agents that the *facilitator* knows (or might learn) about. Sending a *recruit-one* to a *non-facilitator*

agent D will result to something better than a *sorry* response only if some agent C has sent **advertise(C,D,performative(_,C,X))**, that suggests that C is willing to respond to **performative(_,C,X)** by any agent (as long as the other proper tags are in place).

Deciding the agent to be recruited, based only on *advertise* performatives is a somewhat limiting approach. Alternatively, we could allow for a more versatile but less safe approach, where agents may use their *subjective* and *non-standard* inferencing skills to suggest agents to be recruited.

- **recruit-all(A,B,performative(A,_,X))**
  From a semantics point of view this is the same to a series of **forward(A,C,A,B,performative(A,C,X)** messages to all agents C that at some point have sent a **advertise(C,B,performative(B,C,X)**. It should be noted that a **recruit-all(A,B,performative(A,_,X)** is not the same to a **broadcast(A,B,performative(A,_,X))** message, since a *broadcast* will result to **performative(A,_,X)** being sent (wrapped in a *forward*) to all agents that B knows of, regardless of whether they have sent relevant *advertise* messages or not. All observations made about *recruit-one* are valid for *recruit-all* too.

Chapter 6

The Conversation Policies for KQML Performatives

We use the term *conversation* to refer to a sequence of KQML messages that belong to the same thread of communication or interaction between two or possibly more agents. We assume some sort of (intuitive) causal relation between messages that are taken to belong in the same *conversation* and we use the `:in-reply-to` value as the indicator of such linkage. We use the term *conversation policies* to refer to the set of policies or rules that describe the domain of permissible *conversations* among KQML–speaking agents. The *conversation policies* that we provide do not describe *all* possible *conversations* because more complex interactions (and thus conversations) are possible between KQML–speaking agents. We view the conversations allowed by the policies we provide as the building blocks for the more complex interactions and as a starting point for any interaction that is consistent with the semantics of the KQML performatives.

6.1 Using a Definite Clause Grammar for the conversation policies

Next, we provide a brief overview of Definite Clause Grammars (DCGs) and explain how we use them to provide the *conversation policies* for the KQML performatives.

6.1.1 A brief introduction to Definite Clause Grammars

Definite Clause Grammars are an extension of Context Free Grammars (CFGs). A CFG has rules of the form

$$\mathbf{NT} \quad \longrightarrow \quad \mathbf{Body}$$

where **NT** is a non–terminal symbol and **Body** is a sequence of one or more terminal or non–terminal symbols separated by commas. In our notation we express terminals as strings enclosed in brackets. The notation we will use for DCGs extends the notation for CFGs in the following way ([60]):

- **Non–terminals** may be compound terms (instead of just atoms as in the CFG case)
- The body of a rule may contain **procedural attachments**, written within "{" and "}" (in addition to terminals and non–terminals), that express extra conditions that must be satisfied for the rule to be valid.

For example, a DCG rule might look like

$$\textbf{noun(N)} \quad \longrightarrow \quad \textbf{[W]}\ ,\ \textbf{\{RootForm(W,N), is\_noun(N)\}}$$

with the possible meaning that "a phrase identified as the noun **N** may consist of the single word **W**, where **N** is the root form of **W** and **N** is a noun" ([60]).

6.1.2 Definite Clause Grammars and KQML conversation policies

The term *conversation policies* includes both the permissible sequences (conversations) of KQML performatives and the constraints and dependencies on the values of the *reserved parameters* of those performative. In other words, we are not only interested in asserting that an *ask-if* might be followed by a *tell* (among other performatives) but we want to also capture constraints such as, the $\texttt{content}_{ask-if}$ being the same with the $\texttt{cont2ent}_{tell}$ or the $\texttt{reply}-\texttt{with}_{ask-if}$ being also the $\texttt{in}-\texttt{reply}-\texttt{to}_{tell}$. The DCG we provide in the next section fully describes all of the above in a declarative fashion.

Besides providing a lucid and convenient formalism for presenting the *conversation policies*, DCGs offer some additional advantages:

1. They have a straightforward implementation in Prolog (presented in Appendix A), which means that there is no additional work required to implement them into code.
2. The arguments of non–terminals can be used to carry and test contextual information.
3. They provide the means (the extra arguments of non–terminals) for building structure in the grammar rules and although this is feature we need not explore at that point, it might be a convenient extension for future implementations.

Our description and implementation of the conversation policies using a DCG makes extensive use of features (1) and (2) allowing as to provide a description that would not be possible had we chosen a CFG or a Finite State Machine for the task. Another formalism that would probably provide us with the same flexibility is that of Augmented Transition Networks[1] (ATNs), but DCGs have the advantage that they can be expressed directly in a general purpose programming language like Prolog (in fact our DCG is a Prolog program).

We chose to provide the *conversation policies* using a DCG. Each KQML message is a *terminal* in this grammar. A terminal is a list of the following values: `performative_name`, `:sender`, `:receiver`, `:in-reply-to`, `:reply-with`, `:language`, `:ontology`, `IO` (if `IO` is set to 1 the message is an incoming message and if it is set to O the current message is an outgoing message), `:content` and whenever the `:content` is a performative itself, then the

[1]Perreira and Warren claim that DCGs are at least as powerful of a formalism as ATNs ([6]), with DCGs having some considerable advantages over ATNs ([60]).

`:content` is going to be a list itself. Since *terminals* are enclosed in "[" and "]" a terminal in our DCG will look like:

[[ask-if,A,B,id1,id2,prolog,bar,foo(X,Y)]]

The conversation policies we present are tied to the semantics given in Section 5 in the sense that changes in the semantic description would result to different conversation policies. For example, as will become obvious in the next section, we only allow conversations that have query performatives (like *ask-if*, *ask-one*, *etc.*) if an appropriate *advertise* has taken place. Our conversation policies technically are not inferred from the semantic description, but we believe that they define the *minimal* set of conversations that are consistent with the semantics. The conversation policies have to obey the constraints imposed by the semantic description:

- If a performative has *preconditions* for the sender, then it cannot start a conversation since those preconditions have to be established by a preceding performative (see *tell*).
- If the *completion condition(s)* for a performative are not empty, then a performative cannot end a conversation since further (communicative) action has to take place to establish the *completion condition(s)* (see *ask-if* or *ask-all*).
- A performative may be preceded by a performative that can (partially) establish its preconditions (*e.g.*, a *tell* may be preceded by an *ask-if*).

6.2 The conversation policies for KQML performatives, in detail

Before presenting in detail the Definite Clause Grammar (DCG) describing the conversation policies for KQML–speaking agents, we should explain some of the idiosyncrasies and technicalities of the formalism we are using, which might look strange with respect to the canonical use of DCGs in Natural Language Processing:

- All non-terminals have parameters for the values for `performative_name`, `:sender`, `:receiver`, `:in-reply-to`, `:reply-with`, `:language`, `:ontology` and `:content`, even if often not all of them are necessary, for the purpose of consistency of the presentation.
- Sometimes certain variables might be uninstantiated and we chose to include them as place–holders for consistency and clarity purposes.
- The procedural attachments might make reference to predicates that are beyond the scope of the particular rule in which they appear (something that Prolog allows us to do but that in no way alters the essence of our presentation since we could pass such arguments along our non-terminals, but we chose not to, for more clarity).
- There are no predefined terminals; our terminals are all instances of lists of the values (`:keywords`) of KQML messages that are reachable from the start.
- The first letters of non-terminals have the following intuitive meaning: "s" has the meaning of a starting state for the conversation or sub–conversation (or sub–dialogue) of the particular performative; "r" has the meaning of a direct response; and "c" has the meaning of a continuation (or a follow–up) in the current thread. Of course, the choice of symbols has no significance on the workings of the grammar and only serves to ease understanding of it by the human reader (as opposed to the Prolog interpreter).
- The choice of making the *performative_name* a parameter in the non–terminals (instead of introducing distinct non–terminals for each performative) was based on merits of aesthetic and simplicity rather than functionality; a grammar of possible KQML conversations could be provided in this alternative way, but we think that our choice makes the grammar easier to follow.
- We use the values *IO* and *OI* in the terminals to signify whether a performative is incoming or outgoing. `IO` and `OI` can only take the values 0 and 1, and they always have complimentary values.

The DCG can be seen in full, as a Prolog program, in Appendix A.

ASK-IF, ASK-ONE and **ASK-ALL** Performatives

S $\longrightarrow$

    s(Cnv,Prf,S,R,IRt,Rw,IO,C),
    {member(Prf,[advertise,broker-one,broker-all,forward,
        broadcast,recommend-one,recommend-all,

```
            recruit-one,recruit-all,register,
            unregister,transport-address])}
s(Cnv,ask-if,S,R,IRt,Rw,IO,C) ⟶
        [[ask-if ,S,R,IRt,Rw,IO,C]] |
        [[ask-if ,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
            r(Cnv,ask-if,S,R,_,Rw,OI,C)
r(Cnv,ask-if,R,S,_,IRt,IO,C) ⟶
        [[tell ,S,R,IRt,Rw,IO,C]] |
        [[untell ,S,R,IRt,Rw,IO,C]] |
        [[deny ,S,R,IRt,Rw,IO,C]] |
        problem(Cnv,R,S,IRt,_,IO)
s(Cnv,ask-one,S,R,IRt,Rw,IO,C) ⟶
        [[ask-one,S,R,IRt,Rw,IO,C]] |
        [[ask-one,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
            r(Cnv,ask-one,S,R,_,Rw,OI,_)
s(Cnv,ask-all,S,R,IRt,Rw,IO,C) ⟶
        [[ask-all,S,R,IRt,Rw,IO,C]] |
        [[ask-all,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
            r(Cnv,ask-all,S,R,_,Rw,OI,_)
r(Cnv,ask-one,S,R,IRt,Rw,IO,_) ⟶
        r(Cnv,ask,S,R,IRt,Rw,IO,_)
r(Cnv,ask-all,S,R,IRt,Rw,IO,_) ⟶
        r(Cnv,ask,S,R,IRt,Rw,IO,_)
r(Cnv,ask,R,S,_,IRt,IO,C) ⟶
        [[tell ,S,R,IRt,Rw,IO,C]] |
        [[untell ,S,R,IRt,Rw,IO,C]] |
        [[deny ,S,R,IRt,Rw,IO,C]] |
        problem(Cnv,R,S,IRt,_,IO)
```

Let us offer some help with reading these DCG rules. The rules are organized into groups that describe the sub–dialogues that may start with a performative, or a group of them. The rules are written in such a manner that any sequence of messages that is reachable from the start is also a conversation that will be accepted by the DCG. In other words, we do not subscribe to any notion of a *complete* KQML conversation, although it might be possible to define such conversations in some cases. Rules might be called by other rules. As a result, an *advertise* of an *ask-if* (the rules for *advertise* are given in page 88) is a conversation; if a proper *ask-if* follows the *advertise*, the sequence of *advertise* and *ask-if* is a conversation; and finally, if an appropriate *tell* follows the *ask-if*, the resulting sequence of the three messages will be a conversation that the DCG will accept. The values of the various terminals and non–terminals define what an *appropriate* follow–up is, at any point of a KQML exchange. We use the following variables for the various tokens that appear in the DCG (symbols that start with a capital–case letter are variables and those that start with small–case letters are constants): `Cnv` stands for the conversation that the DCG handles; `Prf` is the *performative_name*; `S` is the `:sender`; `R` is the `:receiver`; `IRt` is the `:in-reply-to` value; `Rw` is the `:reply-with`; `IO` is the variable that indicates if a message is an incoming or outgoing one; and `C` is the `:content`.

We take the position that all starting points for conversations are *advertise* performatives (see also Table 4.4 or page 40 for which performatives can appear in the `:content` of an *advertise*), except for *register*, *unregister* and *transport-address* (only *facilitators* can process them) and the *facilitation performatives* (when sent to, or processed by facilitators). The *query* performatives mentioned here (*ask-if*, *ask-all* and *ask-all*) may follow an *advertise* (see page 88) and may be responded to with any of *tell*, *untell* and *deny*. A response with a *sorry* or *error* is always a possibility of course. The `:in-reply-to` value of the response is the `:reply-with` of the *query* performative. This is going to be the case with all performatives that act as a *response* or a *follow-up* to some other performative. Also, notice that the `:content` of a response is the same as the `:content` of the querying performative in the case of the *ask-if*.

ERROR and SORRY Performatives

problem(Cnv,R,S,IRt,Rw,IO) $\longrightarrow$
  [[error ,S,R,IRt,Rw,IO,[]]] |
  [[sorry ,S,R,IRt,Rw,IO,[]]]

A *problematic* or a *non–positive* response, *i.e.*, a *sorry* or *error* is always a possibility and those two performatives may follow almost any performative (except for another *sorry* or *error*).

STREAM-ALL Performative

s(Cnv,stream-all,S,R,IRt,Rw,IO,C) $\longrightarrow$
  [[stream-all,S,R,IRt,Rw,IO,C]] |
  [[stream-all,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
    r(Cnv,stream-all,S,R,_,Rw,OI,_)
r(Cnv,stream-all,R,S,_,IRt,IO,C) $\longrightarrow$
  [[tell ,S,R,IRt,Rw,IO,C]] |
  [[tell,S,R,IRt,Rw,IO,C]] , r_stream1(Cnv,S,R,IRt,_,IO,_) |
  [[untell ,S,R,IRt,Rw,IO,C]] |
  [[deny ,S,R,IRt,Rw,IO,C]] |
  problem(Cnv,R,S,IRt,_,IO)
r_stream1(Cnv,S,R,IRt,_,IO,C) $\longrightarrow$
  [[tell,S,R,IRt,Rw,IO,C]] , r_stream1(Cnv,S,R,IRt,_,IO,_) |
  [[eos ,S,R,IRt,Rw,IO,[]]] |
  []

The possible responses to a *stream-all* are not different than those in the case of a *ask-all* performative except in the case of a *positive* response, which is going to be *at least* one *tell* (in a *complete* conversation) followed by the *eos* marker.

INSERT and **UNINSERT** Performatives

s(Cnv,insert,S,R,IRt,Rw,IO,C) $\longrightarrow$
  [[insert,S,R,IRt,Rw,IO,C]], c_insert1(Cnv,S,R,IRt,Rw,IO,C)

```
c_insert1(Cnv,S,R,IRt_insert,Rw,IO,C) --->
        {OI is abs(1-IO)},
        problem(Cnv,S,R,Rw,_,OI) |
        c_insert2(Cnv,S,R,IRt_insert,_,IO,C) |
        []
c_insert2(Cnv,S,R,IRt_insert,_,IO,C) --->
        [[uninsert,S,R,IRt_insert,Rw1,IO,C]],
              c_insert3(Cnv,S,R,_,Rw1,IO,C) |
        []
c_insert3(Cnv,S,R,_,Rw1,IO,_) --->
        {OI is abs(1-IO)},
        problem(Cnv,S,R,Rw1,_,OI) |
        []
```

The only interesting observation about *insert* is the possibility of "un-inserting", which is done by deleting (using *uninsert*) the same *:content* that was inserted. This can only occur if such an *insert* has taken place. Implicit in this is the idea that the advertising of a particular *insert*, also "advertises" the possibility of an "un-inserting" action.

DELETE-ONE, **DELETE-ALL** and **UNDELETE** Performatives

```
s(Cnv,delete-one,S,R,IRt,Rw,IO,C) --->
        [[delete-one,S,R,IRt,Rw,IO,C]], c_delete1(Cnv,S,R,IRt,Rw,IO,C)
s(Cnv,delete-all,S,R,IRt,Rw,IO,C) --->
        [[delete-all,S,R,IRt,Rw,IO,C]], c_delete1(Cnv,S,R,IRt,Rw,IO,C)
c_delete1(Cnv,S,R,IRt_delete,Rw,IO,C) --->
        {OI is abs(1-IO)},
        problem(Cnv,S,R,Rw,_,OI) |
        c_delete2(Cnv,S,R,IRt_delete,_,IO,C) |
        []
c_delete2(Cnv,S,R,IRt_delete,_,IO,C) --->
        [[undelete,S,R,IRt_delete,Rw1,IO,C]],
              c_delete3(Cnv,S,R,_,Rw1,IO,C) |
        []
c_delete3(Cnv,S,R,_,Rw1,IO,_) --->
        {OI is abs(1-IO)},
        problem(Cnv,S,R,Rw1,_,OI) |
        []
```

The comments mentioned about *insert* and *uninsert* holds for *delete-one*, *delete-all* and *undelete* too.

ACHIEVE and **UNACHIEVE** Performatives

```
s(Cnv,achieve,S,R,IRt,Rw,IO,C) --->
        [[achieve,S,R,IRt,Rw,IO,C]], c_achieve1(Cnv,S,R,IRt,Rw,IO,C)
```

```
c_achieve1(Cnv,S,R,IRt_achieve,Rw,IO,C) -->
        {OI is abs(1-IO)},
        problem(Cnv,S,R,Rw,_,OI) |
        c_achieve2(Cnv,S,R,IRt_achieve,_,IO,C) |
        []
c_achieve2(Cnv,S,R,IRt_achieve,_,IO,C) -->
        [[unachieve,S,R,IRt_achieve,Rw1,IO,C]],
              c_achieve3(Cnv,S,R,_,Rw1,IO,C) |
        []
c_achieve3(Cnv,S,R,_,Rw1,IO,_) -->
        {OI is abs(1-IO)},
        problem(Cnv,S,R,Rw1,_,OI) |
        []
```

Whatever was mentioned about *insert* and *uninsert* (or *delete-one*, *delete-all* and *undelete*) holds for *achieve* and *unachieve*.

ADVERTISE Performative

```
s(Cnv,advertise,S,R,_,Rw,IO,_) -->
        { OI is abs(1-IO) },
        [[advertise,S,R,_,Rw,IO,[Prf1,R,S,Rw,_,OI,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,subscribe,
              insert,delete-one,delete-all,achieve])},
        c_advertise(Cnv,Prf1,S,R,Rw,_,OI,C1)
c_advertise(Cnv,Prf,R,S,Rw_advertise,_,IO,C) -->
        s(Cnv,Prf,S,R,Rw_advertise,_,IO,C) |
        problem(Cnv,S,R,Rw_advertise,_,IO) |
        []
```

The procedural attachment restricts the performatives that might appear in the `:content` of an *advertise* (see also Table 4.4 or page 40). The `:content` has the form of the expected follow-up to the *advertise*. This follow–up is given by the part of the DCG that starts the sub–dialogue for the embedded performative. Note that it is possible to have a *sorry* or *error* response to the *advertise* itself, as well to the follow-ups to the *advertise*.

SUBSCRIBE Performative

```
s(Cnv,subscribe,S,R,IRt,Rw,IO,C) -->
        {OI is abs(1-IO)},
        [[subscribe,S,R,IRt,Rw,IO,[Prf1,S,R,IRt,Rw1,IO,C1]]],
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,
              recommend-one,recommend-all])},
        c_subscribe(Cnv,Prf1,S,R,Rw,Rw1,OI,C1)
c_subscribe(Cnv,Prf,S,R,Rw_subscribe,Rw,IO,C) -->
        [] |
```

```
    problem(Cnv,S,R,Rw_subscribe,_,IO) |
    r(Cnv,Prf,S,R,_,Rw,IO,_),
    c_subscribe(Cnv,Prf,S,R,Rw_subscribe,Rw,IO,_)
```

The possible performatives in the `:content` of a *subscribe* are restricted as in the case of *advertise* (see also Table 4.4 or page 42). Since *subscribe* results to $\texttt{receiver}_{subscribe}$ responding again and again to the performative in the `:content` of *subscribe*, the follow–up invokes the part of the DCG that points to the response for the performatives in the procedural attachment and this call may be invoked *ad infinitum*.

REGISTER, **UNREGISTER** and **TRANSPORT-ADDRESS** Performatives

```
s(Cnv,register,S,R,IRt,Rw,IO,C) —→
    [[register,S,R,IRt,Rw,IO,C]], c_register1(Cnv,S,R,IRt,Rw,IO,C)
c_register1(Cnv,S,R,IRt_register,Rw,IO,C) —→
    {OI is abs(1-IO)},
    problem(Cnv,S,R,Rw,_,OI) |
    c_register2(Cnv,S,R,IRt_register,_,IO,C) |
    []
c_register2(Cnv,S,R,IRt_register,_,IO,C) —→
    [[unregister,S,R,IRt_register,Rw1,IO,C]],
        c_register3(Cnv,S,R,_,Rw1,IO,C) |
    [[transport-address,S,R,IRt_register,Rw1,IO,_]],
        c_register3(Cnv,S,R,_,Rw1,IO,C) |
    []
c_register3(Cnv,S,R,_,Rw1,IO,_) —→
    {OI is abs(1-IO)},
    problem(Cnv,S,R,Rw1,_,OI) |
    []
```

A *register* may be followed by an *unregister* or a *transport-address*, *i.e.*, an agent that has already registered may unregister in the future or relocate to a new network address.

FORWARD Performative

```
s(Cnv,forward,S,R,IRt,Rw,IO,C) —→
    [[forward,S,R,IRt,Rw,IO,[Prf,From,To,IRt1,Rw1,IO,C1]]],
    c_forward(Cnv,From,To,S,R,IRt,Rw,IO,Prf,IRt1,Rw1,C1)
c_forward(Cnv,From,To,S,R,IRt,Rw,IO,Prf,IRt1,Rw1,C1) —→
    [] |
    {OI is abs(1-IO)},
    problem(Cnv,S,R,Rw,_,OI) |
    {(To = R, IO = 1) ; (From = S, IO = 0)} |
    { To ≠ R ; From ≠ S},
    [[forward,R,NewR,Rw,Rw2,OI,[Prf,From,To,IRt1,Rw1,IO,C1]]]
```

We can identify the following four possibilities that may occur in the handling of a *forward*.

1. `:to` = `:receiver` **and IO = 1**, *i.e.*, the local agent (the `:receiver` of the *forward*) is the final destination of the (incoming) *forward*. This case will never be encountered because such a *forward*, as far as the conversation policies are concerned, is treated as if the `:content` of the *forward*, *i.e.* the embedded KQML message was received, and the appropriate DCG rule applies.

2. `:to` ≠ `:receiver` **and IO = 1**, *i.e.*, the local agent (the `:sender` of the *forward*) is *not* the final destination of the (incoming) *forward* and as a result the local agent acts only as a relay, sending another *forward* message, either to the `:to` agent, or to some other agent that stands a better chance of delivering the `:content` of the *forward* to its final destination, the `:to` agent.

3. `:from` = `:sender` **and IO = 0**, *i.e.*, the local agent (the `:receiver` of the *forward*) is the originator of the (outgoing) *forward*. This case "mirrors" case 1, and is also treated (for the local agent) as if the embedded KQML message was sent out.

4. `:from` ≠ `:sender` **and IO = 0**, *i.e.*, the local agent (the `:sender` of the *forward*) is **not** the originator of the (outgoing) *forward*. This can only occur if previously the local agent received a *forward* that was intended for some other agent and as a result the local agent sent out another *forward* to the `:to` agent (case 2 above).

In the DCG portion for *forward*, cases 1 and 3 are just mentioned but nothing happens with them since they are dealt by other sections of the grammar and the "mirrored" cases 2 and 4 are treated by the two successive *forward* messages (terminals).

BROADCAST Performative

```
s(Cnv,broadcast,S,R,IRt,Rw,IO,C) -->
        [[broadcast,S,R,_,Rw,1,[Prf,S,_,_,_,1,C1]]],
        c_broadcast(Cnv,S,R,IRt,Rw,IO,Prf,C1)
c_broadcast(Cnv,S,R,IRt,Rw,IO,Prf,C) -->
        [] |
        problem(Cnv,S,R,Rw,_,OI) |
        {IO = 1},
        c_broadcast1(Cnv,S,R,IRt,Rw,IO,Prf,C) |
        {IO = 0}
c_broadcast1(Cnv,S,R,IRt,Rw,IO,Prf,C) -->
        c_broadcast2(Cnv,S,R,IRt,Rw,IO,Prf,C),
        c_broadcast1(Cnv,S,R,IRt,Rw,IO,Prf,C)
c_broadcast2(Cnv,S,R,IRt,Rw,IO,Prf,C) -->
        [] |
        [[forward,R,NewR,_,Rw1,IO,[Prf,S,NewR,IRt2,Rw2,IO,C]]]
```

When a *broadcast* is received, it is to be followed by multiple outgoing *forward* messages to all agents that the local agent knows of. The case of a an outgoing *broadcast* will never occur, as far as the conversation policies are concerned, because *broadcast* is treated as multiple versions of the `:content` of *broadcast*, a KQML message itself, sent out.

BROKER-ONE Performative

```
s(Cnv,broker-one,S,R,IRt,Rw,IO,C) ⟶
        {OI is abs(1-IO)},
        [[broker-one,S,R,IRt,Rw,IO,[Prf1,R,_R,Rw,Rw1,_,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,insert,
              delete-one,delete-all,achieve,subscribe])},
        c_broker_one(Cnv,Prf1,S,R,Rw,Rw1,OI,C1)
c_broker_one(Cnv,Prf,R,S,Rw_broker,Rw,1,C) ⟶
        [] |
        problem(Cnv,S,R,Rw_broker,_,1) |
        r(Cnv,Prf,Brk,R,_,Rw,1,C) |
        c_insert(Cnv,Brk,R,_,Rw,0,C) |
        c_delete(Cnv,Brk,R,_,Rw,0,C) |
        c_achieve(Cnv,Brk,R,_,Rw,0,C) |
        c_subscribe(Cnv,Prf,Brk,R,_,Rw,1,C)
c_broker_one(Cnv,Prf,R,S,Rw_broker,Rw,0,C) ⟶
        [] |
        problem(Cnv,S,R,Rw_broker,Rw,0) |
        s(Cnv,Prf,S,Brk,Rw_broker,Rw,0,C),
        c_broker_one1(Cnv,Prf,S,R,Brk,Rw_broker,Rw,0,C)
c_broker_one1(Cnv,Prf,S,R,Brk,Rw_broker,Rw,IO,C) ⟶
        [] |
        {OI is abs(1-IO), last(Cnv,[Prf1,Brk,S,Rw,Rw1,OI,C1]),
        assert(send_MSG([forward,S,R,Rw_broker,Rw2,OI,
              [Prf1,Brk,R,Rw_broker,Rw1,OI,C1]]))} |
        [[forward,S,R,Rw_broker,Rw2,OI,[Prf1,Brk,R,Rw_broker,Rw1,OI,C1]]]
```

The *broker-one* performative presents an interesting case because it involves a three-party interaction. The $\texttt{receiver}_{broker-one}$ sends the $\texttt{content}_{broker-one}$ (with the appropriate values) to some other agent and then passes the response(s) to it to the $\texttt{sender}_{broker-one}$. The last part of this exchange can be done automatically with a procedural attachment in the DCG instead of being taken care of by the handler function for *broker-one*. As the `c_broker_one1` rule suggests, a sub–dialogue (a new conversation) with the third agent starts and the response (or follow–up), *i.e.*, the last message in the conversation being handled by the DCG with the expected values for `:sender` and `:in-reply-to`, is sent to $\texttt{sender}_{broker-one}$ (this is the meaning of the procedural attachment in the `c_broker_one1` rule, that makes reference to predicates that are not a part of the DCG).

If the local agent sent a *broker-one*, the message expected is the prescribed response or follow-up to the *performative* in the `:content`. Technically this message (or messages) will arrive wrapped in a *forward* but from the DCG point of view will be stripped from their "forwarding" packaging (see Case 1 of *forward*, above).

This performative is a prime example of how complicated interactions might be composed from the simpler building blocks.

BROKER-ALL Performative

```
s(Cnv,broker-all,S,R,IRt,Rw,IO,C) --->
        {OI is abs(1-IO)},
        [[broker-all,S,R,IRt,Rw,IO,[Prf1,R,_,Rw,Rw1,_,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,insert,
              delete-one,delete-all,achieve,subscribe])},
        c_broker_all(Cnv,Prf1,S,R,Rw,Rw1,OI,C1)
c_broker_all(Cnv,Prf,R,S,Rw_broker,Rw,1,C) --->
        problem(Cnv,S,R,Rw_broker,_,1) |
        c_broker_all1(Cnv,Prf,S,R,Brk,Rw_broker,Rw,1,C),
        c_broker_all(Cnv,Prf,R,S,Rw_broker,Rw,1,C)
c_broker_all1(Cnv,Prf,S,R,Brk,Rw_broker,Rw,IO,C) --->
        [] |
        r(Cnv,Prf,Brk,R,_,Rw,1,C) |
        c_insert(Cnv,Brk,R,_,Rw,0,C) |
        c_delete(Cnv,Brk,R,_,Rw,0,C) |
        c_achieve(Cnv,Brk,R,_,Rw,0,C) |
        c_subscribe(Cnv,Prf,Brk,R,_,Rw,1,C)
c_broker_all(Cnv,Prf,R,S,Rw_broker,Rw,0,C) --->
        [] |
        problem(Cnv,S,R,Rw_broker,_,0) |
        s(Cnv,Prf,S,Brk,Rw_broker,Rw,0,C),
        c_broker_all2(Cnv,Prf,S,R,Brk,Rw_broker,Rw,0,C),
        c_broker_all(Cnv,Prf,R,S,Rw_broker,Rw,0,C)
c_broker_all2(Cnv,Prf,S,R,Brk,Rw_broker,Rw,IO,C) --->
        [] |
        {OI is abs(1-IO), last(Cnv,[Prf1,Brk,S,Rw,Rw1,OI,C1]),
        assert(send_MSG([forward,S,R,Rw_broker,Rw2,OI,
              [Prf1,Brk,R,Rw_broker,Rw1,OI,C1]]))} |
        [[forward,S,R,Rw_broker,Rw2,OI,[Prf1,Brk,R,Rw_broker,Rw1,OI,C1]]]
```

A *broker-all* is similar to a *broker-one* with the necessary adjustments for the case of an incoming *broker-all* (multiple messages are sent out) and that of an outgoing *broker-all* (multiple *responses* might be received as a result of the multiple agents that might be broker-ed).

RECOMMEND-ONE Performative

```
s(Cnv,recommend-one,S,R,IRt,Rw,IO,C) --->
        {OI is abs(1-IO)},
        [[recommend-one,S,R,IRt,Rw,IO,[Prf1,_,S,_,_,OI,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,insert,
              delete-one,delete-all,achieve,subscribe])},
        c_recommend_one(Cnv,S,R,_,Rw,OI,Prf1,C1)
c_recommend_one(Cnv,R,S,_,Rw_rec,IO,Prf,C1) --->
        [] |
```

```
        problem(Cnv,S,R,Rw_rec,_,IO) |
        {OI is abs(1-IO)},
        [[forward,S,R,Rw_rec,Rw,IO,[advertise,Rec,_,R,Rw_adv,IO,
              [Prf,R,Rec,Rw_adv,_,OI,C]]]]
```

The response to a *recommend-one* is a *forward* of an *advertise* message that suits the `:content` of the *recommend-one*.

RECOMMEND-ALL Performative

```
s(Cnv,recommend-all,S,R,IRt,Rw,IO,C) −→
        {OI is abs(1-IO)},
        [[recommend-all,S,R,IRt,Rw,IO,[Prf1,_,S,_,_,OI,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,insert,
              delete-one,delete-all,achieve,subscribe])},
        c_recommend_all(Cnv,S,R,_,Rw,OI,Prf1,C1)
c_recommend_all(Cnv,R,S,_,Rw_rec,IO,Prf,C1) −→
        [] |
        problem(Cnv,S,R,Rw_rec,_,IO) |
        {OI is abs(1-IO)},
        c_recommend_all2(Cnv,R,S,_,Rw_rec,IO,Prf,C1)
c_recommend_all2(Cnv,R,S,_,Rw_rec,IO,Prf,C1) −→
        [[forward,R,S,Rw_rec,Rw,IO,[advertise,Rec,S,_,Rw_adv,IO,
              [Prf,S,Rec,Rw_adv,_,OI,C]]]],
        c_recommend_all(Cnv,R,S,_,Rw_rec,IO,Prf,C1)
```

Similarly to the case for *recommend-one*, a *recommend-all* is followed by multiple *forward* messages, one for each *advertise* message that matches the `:content` of the *recommend-all*.

RECRUIT-ONE Performative

```
s(Cnv,recruit-one,S,R,IRt,Rw,IO,C) −→
        {OI is abs(1-IO)},
        [[recruit-one,S,R,IRt,Rw,IO,[Prf1,_,_,_,Rw1,OI,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,insert,
              delete-one,delete-all,achieve,subscribe])},
        c_recruit_one(Cnv,S,R,_,Rw,OI,Prf1,Rw1,C1)
c_recruit_one(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C) −→
        [] |
        problem(Cnv,S,R,Rw_recruit,_,0) |
        {OI is abs(1-IO)},
        [[forward,R,S,Rw_rec,Rw1,0,[Prf,S,Rec,Rw_adv,Rw2,OI,C]]]
c_recruit_one(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C) −→
        [] |
        problem(Cnv,S,R,Rw_recruit,_,1) |
        r(Cnv,Prf,Rec,R,_,Rw,1,C) |
```

```
        c_insert(Cnv,Rec,R,_,Rw,0,C) |
        c_delete(Cnv,Rec,R,_,Rw,0,C) |
        c_achieve(Cnv,Rec,R,_,Rw,0,C) |
        c_subscribe(Cnv,Prf,Rec,R,_,Rw,1,C).
```

A *recruit-one* can be thought as a special case of *forward* and this is reflected in the DCG rules shown above (see also page 79). If the *recruit-one* was received by the local agent, a *forward* will be sent to some `:to` agent, which is decided by the local agent (usually a facilitator), using (supposedly) the relevant *advertise* messages. The `:from` agent of the *forward* will be the `:sender` of the *recruit-one*. If the local agent is the `:sender` of the *recruit-one*, *i.e.*, the message is outgoing, the *recruit-one* will be followed by whatever messages may succeed the embedded KQML message, *i.e.*, the `:content` of the *recruit-one*; the procedural attachment restricts the performatives that may appear in the `:content` of a *recruit-one*.

RECRUIT-ALL Performative

```
s(Cnv,recruit-all,S,R,IRt,Rw,IO,C) ⟶
        {OI is abs(1-IO)},
        [[recruit-all,S,R,IRt,Rw,IO,[Prf1,_,_,_,Rw1,OI,C1]]] ,
        {member(Prf1,[ask-if,ask-all,ask-one,stream-all,insert,
              delete-one,delete-all,achieve,subscribe])},
        c_recruit_all(Cnv,S,R,_,Rw,OI,Prf1,Rw1,C1)
c_recruit_all(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C) ⟶
        [] |
        problem(Cnv,S,R,Rw_recruit,_,0) |
        c_recruit_all_f(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C)
c_recruit_all_f(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C) ⟶
        c_recruit_all_f1(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C),
        c_recruit_all_f(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C)
c_recruit_all_f1(Cnv,R,S,_,Rw_recr,0,Prf,Rw,C) ⟶
        [] |
        [[forward,R,S,Rw_rec,Rw1,0,[Prf,S,Rec,Rw_adv,Rw2,OI,C]]]
c_recruit_all(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C) ⟶
        [] |
        problem(Cnv,S,R,Rw_recruit,_,1) |
        c_recruit_all_m(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C)
c_recruit_all_m(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C) ⟶
        c_recruit_all_m1(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C),
        c_recruit_all_m(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C)
c_recruit_all_m1(Cnv,R,S,_,Rw_recr,1,Prf,Rw,C) ⟶
        [] |
        r(Cnv,Prf,Rec,R,_,Rw,1,C) |
        c_insert(Cnv,Rec,R,_,Rw,0,C) |
        c_delete(Cnv,Rec,R,_,Rw,0,C) |
        c_achieve(Cnv,Rec,R,_,Rw,0,C) |
        c_subscribe(Cnv,Prf,Rec,R,_,Rw,1,C)
```

A *recruit-all* is similar to a *recruit-one* with the necessary adjustments for the case of an incoming *recruit-all* (multiple *forward* messages are sent out) and that of an outgoing *recruit-all* (multiple *responses* might be received as a result of the multiple agents that might be recruited).

Chapter 7

Implementing the Research

The research presented so far allows us to enhance the logical architecture of a KQML–speaking agent. A new component that implements the conversation policies, the so called *conversation* module, intercepts each incoming message after it leaves the router and before the appropriate handler function is dispatched (see Figure 3.1). Also, every outgoing message, before it reaches the router and is delivered to the `:receiver`, goes through the conversation module. The conversation module, along with the code for the handler functions and the router, can be combined to build a facilitator that handles all the networking and facilitation performatives or can be attached to an application to make it a KQML–ready agent.

7.1 The conversation module

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 following describe situations that the module should not allow to happen and handle accordingly:

- As a response to a previously received message, the application uses a performative that is not compatible with the conversation policies. The module should not deliver the message and signal to the application the unsuccessful attempt.[1] Normally, such a scenario would not take place, since the application programmer would be aware of the conversation policies, but there is always the possibility of an error, especially with performatives that take a performative as `:content`.

- The agent receives a message that either does not make any sense for the application, or does not fit into any of the ongoing conversations, nor can it be a starting point for a new one. In this case, the module should send to the `:sender` of the message an *error* reply, so that the original sender *knows* that the message was *not* processed by its intended receiver.

[1] The application programmer could provide a function in the handler functions' module to deal with such incidents automatically.

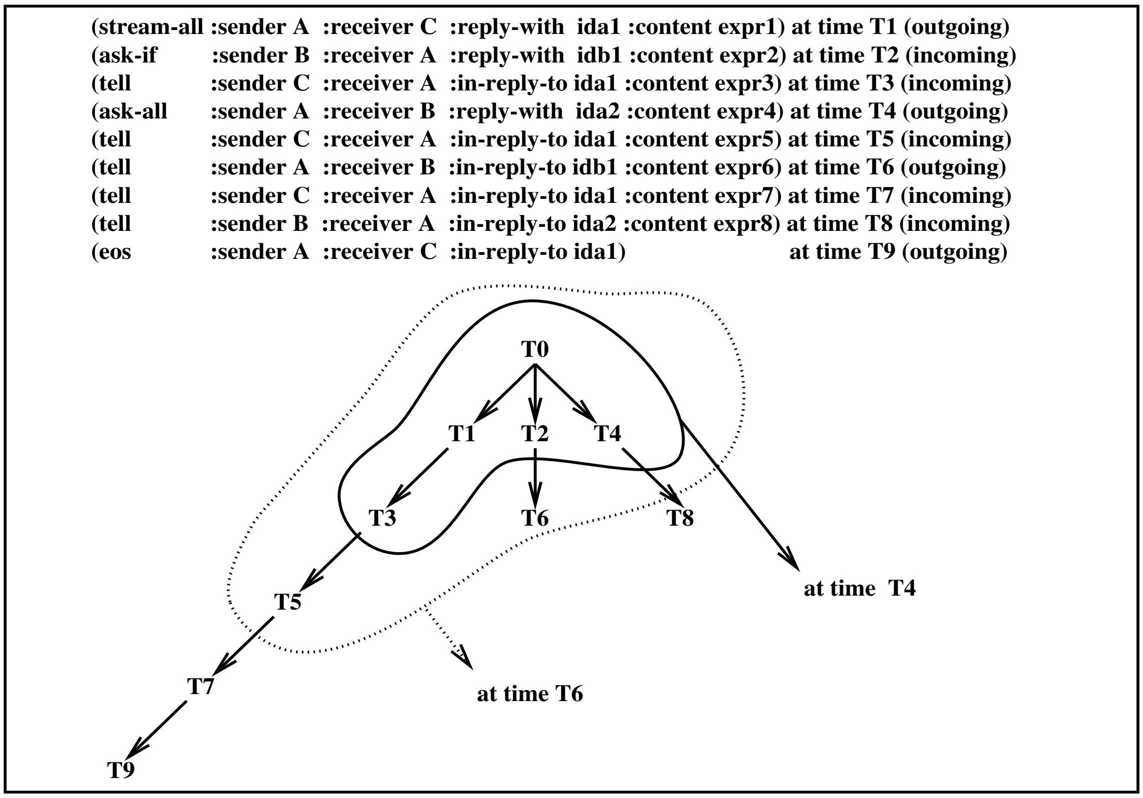

Figure 7.1: Sequence of imported and exported messages for an agent

Figure 7.1 gives an example of a series of messages sent and received by an agent named A during some time period. At any time between times T_1 and T_9 messages belonging three different conversations are handled. The three conversations evolve in parallel, though asynchronously. The conversation module should handle such parallel conversations 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.[2]
3. If no new thread can be started with the current message, a message with the *error* performative will be sent to the sender (if the message is incoming) or a signal will be delivered to the application (if the message is outgoing).

At the level of the DCG described in the previous chapter, every time a new message is to be handled by the module, the message becomes a potential new terminal, then it is appended to the first conversation (thread) and an attempt is made to successfully parse

[2]Not all performatives can be starting points for new threads. For illustrative reasons, in the example of Figure 7.1 we consider the performatives *ask-if*, *ask-all*, *stream-all* to be acceptable starting points.

the new conversation through the DCG. If this fails, the new message is appended to the second conversation, and so on.

We believe that by providing a conversation module that can cooperate with the router, the agent is enabled to better handle asynchronous behavior, keep track of its business and provide the application programmer the means to build more complex schemes of inter–agent communication (protocols like the Contract Net, see [68], [69]). The code that implements the conversation policies in Prolog can be seen in Appendix A; we have omitted that part of the top-level procedure, which also interfaces the module with the router (written in C) of the UMBC/Lockheed–Martin implementation.

7.2 The handler functions

The handler functions are language dependent interfaces to the application. They provide a single point of entry to the application (agent) for processing the performative; there is one such function for each performative. Handlers take whatever action is dictated by the semantics and the conversation policies and then exit. The performative semantics decide what actions the handler take on the agent's knowledge store and the conversation policies decide what the possible responses to a received performative might be, *i.e.*, what KQML messages might be sent as a response. The handler functions *are* the implementation of KQML, so it is important that the code for them is fully compliant with the semantic description and the conversation policies.

Chapter 8

Conclusions

KQML is a language for communication between (intelligent) software agents. Although the language has been in use for quite a while, the specification has been incomplete and no formal semantics have been offered. Revising the specification of the language and providing a semantic framework for KQML is the core of this research. The fundamental constraint in our semantic approach is that KQML should be defined in a manner unrelated to specific architectures, interaction protocols, models of agency, and problem solving strategies. Based on ideas from speech act theory, our semantic approach associates descriptions of the cognitive states of agents with the use and effects of each of the performatives (KQML's reserved message types). After covering the background material and referring to the related literature, we presented our revised KQML specification and the semantic framework along with the semantics for individual performatives, and the conversation policies, *i.e.*, the rules that describe the permissible conversations among KQML–speaking agents. As part of the new specification we also presented the logical architecture of agents in a KQML domain, and related it to the semantic description of the language.

Our research offers a fully specified language for agent communication along with a semantic description, the lack of which has been the source of incessant criticism of KQML, and conversation policies. KQML's success in the computing community remains to be seen, but we believe that the proposed research is a step in the right direction, and that there is both room and need for a language like KQML. Communication languages *per se* have long been neglected and currently there is no all–purpose communication language at KQML's linguistic level. Furthermore, the recent booming of network–centric computing has created the need for intelligent software agents and KQML can serve as a first–rate tool that will allow such programs to communicate with one another. KQML's basic concept, *i.e.*, that agents interact through *communication acts* (that communicate *attitudes* about knowledge) which may be combined into complex interactions, presents merit on its own. Our research offers a method to define the semantics and possible interactions for an agent communication language, irrespective of the particulars of KQML and and its assumptions.

Appendix A

Code for the Conversation Policies

This is the code that implements the conversation policies in Prolog. The code is a straightforward implementation of the DCG describing the conversation policies, which we presented in detail in Chapter 6.

```
% everything has to be ADVERTISEd first
try(Conv,IO) --> s(Conv,Perfo,S,R,IRt,Rw,IO,C),
                    {member(Perfo,[advertise,broker-one,broker-all,forward,
                                  broadcast,recommend-one,recommend-all,
                                  recruit-one,recruit-all,register,
                                  unregister,transport-address])}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ASK performatives, a.k.a --> ASK-IF, ASK-ONE, ASK-ALL
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,ask-if,S,R,IRt,Rw,IO,C) -->
       [[ask-if ,S,R,IRt,Rw,IO,C]] |
       [[ask-if ,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
                                   r(Conv,ask-if,S,R,_,Rw,OI,C).

r(Conv,ask-if,R,S,_,IRt,IO,C) -->
       [[tell   ,S,R,IRt,Rw,IO,C]] |
       [[untell ,S,R,IRt,Rw,IO,C]] |
       [[deny   ,S,R,IRt,Rw,IO,C]] |
       problem(Conv,R,S,IRt,_,IO).

s(Conv,ask-one,S,R,IRt,Rw,IO,C) -->
       [[ask-one,S,R,IRt,Rw,IO,C]] |
       [[ask-one,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
                                   r(Conv,ask-one,S,R,_,Rw,OI,_).

s(Conv,ask-all,S,R,IRt,Rw,IO,C) -->
       [[ask-all,S,R,IRt,Rw,IO,C]] |
       [[ask-all,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
                                   r(Conv,ask-all,S,R,_,Rw,OI,_).

r(Conv,ask-one,S,R,IRt,Rw,IO,_) -->
```

```
        r(Conv,ask,S,R,IRt,Rw,IO,_).

r(Conv,ask-all,S,R,IRt,Rw,IO,_) -->
        r(Conv,ask,S,R,IRt,Rw,IO,_).

r(Conv,ask,R,S,_,IRt,IO,C) -->
        [[tell   ,S,R,IRt,Rw,IO,C]] |
        [[untell ,S,R,IRt,Rw,IO,C]] |
        [[deny   ,S,R,IRt,Rw,IO,C]] |
        problem(Conv,R,S,IRt,_,IO).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ERROR and SORRY performatives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
problem(Conv,R,S,IRt,Rw,IO) -->
        [[error  ,S,R,IRt,Rw,IO,[]]] |
        [[sorry  ,S,R,IRt,Rw,IO,[]]] .

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% STREAM-ALL performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,stream-all,S,R,IRt,Rw,IO,C) -->
        [[stream-all,S,R,IRt,Rw,IO,C]] |
        [[stream-all,S,R,IRt,Rw,IO,C]], {OI is abs(1-IO)},
                                  r(Conv,stream-all,S,R,_,Rw,OI,_).

r(Conv,stream-all,R,S,_,IRt,IO,C) -->
        [[tell   ,S,R,IRt,Rw,IO,C]] |
        [[tell   ,S,R,IRt,Rw,IO,C]] , r_stream1(Conv,S,R,IRt,_,IO,_) |
        [[untell ,S,R,IRt,Rw,IO,C]] |
        [[deny   ,S,R,IRt,Rw,IO,C]] |
        problem(Conv,R,S,IRt,_,IO).

r_stream1(Conv,S,R,IRt,_,IO,C) -->
        [[tell   ,S,R,IRt,Rw,IO,C]] , r_stream1(Conv,S,R,IRt,_,IO,_) |
        [[eos    ,S,R,IRt,Rw,IO,[]]] |
        [].



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% INSERT, UNINSERT performatives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,insert,S,R,IRt,Rw,IO,C) -->
        [[insert,S,R,IRt,Rw,IO,C]], c_insert1(Conv,S,R,IRt,Rw,IO,C).

c_insert1(Conv,S,R,IRt_insert,Rw,IO,C) -->
        {OI is abs(1-IO)},
        problem(Conv,S,R,Rw,_,OI) |
        c_insert2(Conv,S,R,IRt_insert,_,IO,C) |
        [].
```

```
c_insert2(Conv,S,R,IRt_insert,_,IO,C) -->
       [[uninsert,S,R,IRt_insert,Rw1,IO,C]],
                           c_insert3(Conv,S,R,_,Rw1,IO,C) |
       [].

c_insert3(Conv,S,R,_,Rw1,IO,_) -->
       {OI is abs(1-IO)},
       problem(Conv,S,R,Rw1,_,OI) |
       [].



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DELETE-ONE, DELETE-ALL UNDELETE performatives
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,delete-one,S,R,IRt,Rw,IO,C) -->
       [[delete-one,S,R,IRt,Rw,IO,C]], c_delete1(Conv,S,R,IRt,Rw,IO,C).

s(Conv,delete-all,S,R,IRt,Rw,IO,C) -->
       [[delete-all,S,R,IRt,Rw,IO,C]], c_delete1(Conv,S,R,IRt,Rw,IO,C).

c_delete1(Conv,S,R,IRt_delete,Rw,IO,C) -->
       {OI is abs(1-IO)},
       problem(Conv,S,R,Rw,_,OI) |
       c_delete2(Conv,S,R,IRt_delete,_,IO,C) |
       [].

c_delete2(Conv,S,R,IRt_delete,_,IO,C) -->
       [[undelete,S,R,IRt_delete,Rw1,IO,C]],
                           c_delete3(Conv,S,R,_,Rw1,IO,C) |
       [].

c_delete3(Conv,S,R,_,Rw1,IO,_) -->
       {OI is abs(1-IO)},
       problem(Conv,S,R,Rw1,_,OI) |
       [].



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ACHIEVE, UNACHIEVE performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,achieve,S,R,IRt,Rw,IO,C) -->
       [[achieve,S,R,IRt,Rw,IO,C]], c_achieve1(Conv,S,R,IRt,Rw,IO,C).

c_achieve1(Conv,S,R,IRt_achieve,Rw,IO,C) -->
       {OI is abs(1-IO)},
       problem(Conv,S,R,Rw,_,OI) |
       c_achieve2(Conv,S,R,IRt_achieve,_,IO,C) |
       [].

c_achieve2(Conv,S,R,IRt_achieve,_,IO,C) -->
       [[unachieve,S,R,IRt_achieve,Rw1,IO,C]],
```

```
                           c_achieve3(Conv,S,R,_,Rw1,IO,C) |
       [].

c_achieve3(Conv,S,R,_,Rw1,IO,_) -->
       {OI is abs(1-IO)},
       problem(Conv,S,R,Rw1,_,OI) |
       [].




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ADVERTISE peformative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,advertise,S,R,_,Rw,IO,_) -->
       { OI is abs(1-IO) },
       [[advertise,S,R,_,Rw,IO,[Perfo1,R,S,Rw,_,OI,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,subscribe,
                       insert,delete-one,delete-all,achieve])},
       c_advertise(Conv,Perfo1,S,R,Rw,_,OI,C1).

c_advertise(Conv,Perfo,R,S,Rw_advertise,_,IO,C) -->
       s(Conv,Perfo,S,R,Rw_advertise,_,IO,C) |
       problem(Conv,S,R,Rw_advertise,_,IO) |
       [].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SUBSCRIBE performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,subscribe,S,R,IRt,Rw,IO,C) -->
       {OI is abs(1-IO)},
       [[subscribe,S,R,IRt,Rw,IO,[Perfo1,S,R,IRt,Rw1,IO,C1]]],
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,
                       recommend-one,recommend-all])},
       c_subscribe(Conv,Perfo1,S,R,Rw,Rw1,OI,C1).

c_subscribe(Conv,Perfo,S,R,Rw_subscribe,Rw,IO,C) -->
       [] |
       problem(Conv,S,R,Rw_subscribe,_,IO) |
       r(Conv,Perfo,S,R,_,Rw,IO,_),
       c_subscribe(Conv,Perfo,S,R,Rw_subscribe,Rw,IO,_).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% REGISTER, UNREGISTER performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,register,S,R,IRt,Rw,IO,C) -->
       [[register,S,R,IRt,Rw,IO,C]], c_register1(Conv,S,R,IRt,Rw,IO,C).

c_register1(Conv,S,R,IRt_register,Rw,IO,C) -->
       {OI is abs(1-IO)},
       problem(Conv,S,R,Rw,_,OI) |
```

```
        c_register2(Conv,S,R,IRt_register,_,IO,C) |
        [].

c_register2(Conv,S,R,IRt_register,_,IO,C) -->
        [[unregister,S,R,IRt_register,Rw1,IO,C]],
                           c_register3(Conv,S,R,_,Rw1,IO,C) |
        [[transport-address,S,R,IRt_register,Rw1,IO,_]],
                           c_register3(Conv,S,R,_,Rw1,IO,C) |
        [].

c_register3(Conv,S,R,_,Rw1,IO,_) -->
        {OI is abs(1-IO)},
        problem(Conv,S,R,Rw1,_,OI) |
        [].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% FORWARD performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,forward,S,R,IRt,Rw,IO,C) -->
        [[forward,S,R,IRt,Rw,IO,[Perfo,From,To,IRt1,Rw1,IO,C1]]],
        c_forward(Conv,From,To,S,R,IRt,Rw,IO,Perfo,IRt1,Rw1,C1).

c_forward(Conv,From,To,S,R,IRt,Rw,IO,Perfo,IRt1,Rw1,C1) -->
        [] |
        {OI is abs(1-IO)},
        problem(Conv,S,R,Rw,_,OI) |
        {(To  == R, IO == 1) ; (From  == S, IO == 0)} |
        { To \== R ; From \== S},
        [[forward,R,NewR,Rw,Rw2,OI,[Perfo,From,To,IRt1,Rw1,IO,C1]]].



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BROADCAST performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,broadcast,S,R,IRt,Rw,IO,C) -->
        [[broadcast,S,R,_,Rw,1,[Perfo,S,_,_,_,1,C1]]],
        c_broadcast(Conv,S,R,IRt,Rw,IO,Perfo,C1).

c_broadcast(Conv,S,R,IRt,Rw,IO,Perfo,C) -->
        [] |
        problem(Conv,S,R,Rw,_,OI) |
        {IO == 1},
        c_broadcast1(Conv,S,R,IRt,Rw,IO,Perfo,C) |
        {IO == 0}.

c_broadcast1(Conv,S,R,IRt,Rw,IO,Perfo,C) -->
        c_broadcast2(Conv,S,R,IRt,Rw,IO,Perfo,C),
        c_broadcast1(Conv,S,R,IRt,Rw,IO,Perfo,C).

c_broadcast2(Conv,S,R,IRt,Rw,IO,Perfo,C) -->
        [] |
        [[forward,R,NewR,_,Rw1,IO,[Perfo,S,NewR,IRt2,Rw2,IO,C]]].
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BROKER-ONE performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,broker-one,S,R,IRt,Rw,IO,C) -->
       {OI is abs(1-IO)},
       [[broker-one,S,R,IRt,Rw,IO,[Perfo1,R,_R,Rw,Rw1,_,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,insert,
                       delete-one,delete-all,achieve,subscribe])},
       c_broker_one(Conv,Perfo1,S,R,Rw,Rw1,OI,C1).

% if the broker-one was sent by the local agent, the response or
% follow-up from the brokered agent is expected
c_broker_one(Conv,Perfo,R,S,Rw_broker,Rw,1,C) -->
       [] |
       problem(Conv,S,R,Rw_broker,_,1) |
       r(Conv,Perfo,Brk,R,_,Rw,1,C)  |  %for the query perfo's
       c_insert(Conv,Brk,R,_,Rw,0,C) |
       c_delete(Conv,Brk,R,_,Rw,0,C) |
       c_achieve(Conv,Brk,R,_,Rw,0,C) |
       c_subscribe(Conv,Perfo,Brk,R,_,Rw,1,C).

% if the local agent (a facilitator) received a broker-one, he will
% send the content out, as if he is asking the "query" and will wrap
% the response in a forward, to forward it to the broker-one's sender
c_broker_one(Conv,Perfo,R,S,Rw_broker,Rw,0,C) -->
       [] |
       problem(Conv,S,R,Rw_broker,Rw,0) |
       s(Conv,Perfo,S,Brk,Rw_broker,Rw,0,C),
       c_broker_one1(Conv,Perfo,S,R,Brk,Rw_broker,Rw,0,C).

c_broker_one1(Conv,Perfo,S,R,Brk,Rw_broker,Rw,IO,C) -->
       [] |
       {OI is abs(1-IO), last(Conv,[Perfo1,Brk,S,Rw,Rw1,OI,C1]),
       assert(send_MSG([forward,S,R,Rw_broker,Rw2,OI,
                       [Perfo1,Brk,R,Rw_broker,Rw1,OI,C1]]))} |
       [[forward,S,R,Rw_broker,Rw2,OI,[Perfo1,Brk,R,Rw_broker,Rw1,OI,C1]]].



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% BROKER-ALL performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,broker-all,S,R,IRt,Rw,IO,C) -->
       {OI is abs(1-IO)},
       [[broker-all,S,R,IRt,Rw,IO,[Perfo1,R,_,Rw,Rw1,_,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,insert,
                       delete-one,delete-all,achieve,subscribe])},
       c_broker_all(Conv,Perfo1,S,R,Rw,Rw1,OI,C1).

% if the broker-all was sent by the local agent, the responses or
% follow-ups from the brokered agent are expected
```

```
c_broker_all(Conv,Perfo,R,S,Rw_broker,Rw,1,C) -->
       problem(Conv,S,R,Rw_broker,_,1) |
       c_broker_all1(Conv,Perfo,S,R,Brk,Rw_broker,Rw,1,C),
       c_broker_all(Conv,Perfo,R,S,Rw_broker,Rw,1,C).


c_broker_all1(Conv,Perfo,S,R,Brk,Rw_broker,Rw,IO,C) -->
       [] |
       r(Conv,Perfo,Brk,R,_,Rw,1,C)  |  %for the query perfo's
       c_insert(Conv,Brk,R,_,Rw,0,C) |
       c_delete(Conv,Brk,R,_,Rw,0,C) |
       c_achieve(Conv,Brk,R,_,Rw,0,C) |
       c_subscribe(Conv,Perfo,Brk,R,_,Rw,1,C).

% if the local agent (a facilitator) received a broker-all, he will
% sent the message out, as if he is asking the "query" and will wrap
% the response in a forward, to forward it to the sender of the broker-all
c_broker_all(Conv,Perfo,R,S,Rw_broker,Rw,0,C) -->
       [] |
       problem(Conv,S,R,Rw_broker,_,0) |
       s(Conv,Perfo,S,Brk,Rw_broker,Rw,0,C),
       c_broker_all2(Conv,Perfo,S,R,Brk,Rw_broker,Rw,0,C),
       c_broker_all(Conv,Perfo,R,S,Rw_broker,Rw,0,C).

c_broker_all2(Conv,Perfo,S,R,Brk,Rw_broker,Rw,IO,C) -->
       [] |
       {OI is abs(1-IO), last(Conv,[Perfo1,Brk,S,Rw,Rw1,OI,C1]),
       assert(send_MSG([forward,S,R,Rw_broker,Rw2,OI,
                       [Perfo1,Brk,R,Rw_broker,Rw1,OI,C1]]))} |
       [[forward,S,R,Rw_broker,Rw2,OI,[Perfo1,Brk,R,Rw_broker,Rw1,OI,C1]]].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RECOMMEND-ONE performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,recommend-one,S,R,IRt,Rw,IO,C) -->
       {OI is abs(1-IO)},
       [[recommend-one,S,R,IRt,Rw,IO,[Perfo1,_,S,_,_,OI,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,insert,
                       delete-one,delete-all,achieve,subscribe])},
       c_recommend_one(Conv,S,R,_,Rw,OI,Perfo1,C1).

c_recommend_one(Conv,R,S,_,Rw_rec,IO,Perfo,C1) -->
       [] |
       problem(Conv,S,R,Rw_rec,_,IO) |
       {OI is abs(1-IO)},
       [[forward,S,R,Rw_rec,Rw,IO,[advertise,Rec,_,R,Rw_adv,IO,
                                   [Perfo,R,Rec,Rw_adv,_,OI,C]]]].


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RECOMMEND-ALL performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,recommend-all,S,R,IRt,Rw,IO,C) -->
```

```
       {OI is abs(1-IO)},
       [[recommend-all,S,R,IRt,Rw,IO,[Perfo1,_,S,_,_,OI,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,insert,
                       delete-one,delete-all,achieve,subscribe])},
       c_recommend_all(Conv,S,R,_,Rw,OI,Perfo1,C1).

c_recommend_all(Conv,R,S,_,Rw_rec,IO,Perfo,C1) -->
       [] |
       problem(Conv,S,R,Rw_rec,_,IO) |
       {OI is abs(1-IO)},
       c_recommend_all2(Conv,R,S,_,Rw_rec,IO,Perfo,C1).


c_recommend_all2(Conv,R,S,_,Rw_rec,IO,Perfo,C1) -->
       [[forward,R,S,Rw_rec,Rw,IO,[advertise,Rec,S,_,Rw_adv,IO,
                                  [Perfo,S,Rec,Rw_adv,_,OI,C]]]],
       c_recommend_all(Conv,R,S,_,Rw_rec,IO,Perfo,C1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RECRUIT-ONE performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
s(Conv,recruit-one,S,R,IRt,Rw,IO,C) -->
       {OI is abs(1-IO)},
       [[recruit-one,S,R,IRt,Rw,IO,[Perfo1,_,_,_,Rw1,OI,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,insert,
                       delete-one,delete-all,achieve,subscribe])},
       c_recruit_one(Conv,S,R,_,Rw,OI,Perfo1,Rw1,C1).

% if the recruit-one was received by the local agent, a forward is
% appropriety sent to the recruited agent
% Rw_adv is derived from the advertise and Rw2 is given by the issuer of
% the recruit
c_recruit_one(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C) -->
       [] |
       problem(Conv,S,R,Rw_recruit,_,0) |
       {OI is abs(1-IO)},
       [[forward,R,S,Rw_rec,Rw1,0,[Perfo,S,Rec,Rw_adv,Rw2,OI,C]]].


% if the recruit-one was sent by the local agent, the responses or
% follow-ups from the recruited agent is expected
c_recruit_one(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C) -->
       [] |
       problem(Conv,S,R,Rw_recruit,_,1) |
       r(Conv,Perfo,Rec,R,_,Rw,1,C)  |  %for the query perfo's
       c_insert(Conv,Rec,R,_,Rw,0,C) |
       c_delete(Conv,Rec,R,_,Rw,0,C) |
       c_achieve(Conv,Rec,R,_,Rw,0,C) |
       c_subscribe(Conv,Perfo,Rec,R,_,Rw,1,C).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% RECRUIT-ALL performative
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
s(Conv,recruit-all,S,R,IRt,Rw,IO,C) -->
       {OI is abs(1-IO)},
       [[recruit-all,S,R,IRt,Rw,IO,[Perfo1,_,_,_,Rw1,OI,C1]]] ,
       {member(Perfo1,[ask-if,ask-all,ask-one,stream-all,insert,
                       delete-one,delete-all,achieve,subscribe])},
       c_recruit_all(Conv,S,R,_,Rw,OI,Perfo1,Rw1,C1).

% if the recruit-all was received by the local agent, multiple forwards is
% approprietly sent to the recruited agents
% Rw_adv is derived from the advertise and Rw2 is given by the issuer of
% the recruit
c_recruit_all(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C) -->
       [] |
       problem(Conv,S,R,Rw_recruit,_,0) |
       c_recruit_all_f(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C).

c_recruit_all_f(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C) -->
       c_recruit_all_f1(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C),
       c_recruit_all_f(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C).

c_recruit_all_f1(Conv,R,S,_,Rw_recr,0,Perfo,Rw,C) -->
       [] |
       [[forward,R,S,Rw_rec,Rw1,0,[Perfo,S,Rec,Rw_adv,Rw2,OI,C]]].


% if the recruit-all was sent by the local agent, the responses or
% follow-ups from the recruited agent are expected
c_recruit_all(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C) -->
       [] |
       problem(Conv,S,R,Rw_recruit,_,1) |
       c_recruit_all_m(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C).

c_recruit_all_m(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C) -->
       c_recruit_all_m1(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C),
       c_recruit_all_m(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C).

c_recruit_all_m1(Conv,R,S,_,Rw_recr,1,Perfo,Rw,C) -->
       [] |
       r(Conv,Perfo,Rec,R,_,Rw,1,C)  |  %for the query perfo's
       c_insert(Conv,Rec,R,_,Rw,0,C) |
       c_delete(Conv,Rec,R,_,Rw,0,C) |
       c_achieve(Conv,Rec,R,_,Rw,0,C) |
       c_subscribe(Conv,Perfo,Rec,R,_,Rw,1,C).
```

Bibliography

[1] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[2] External Interfaces Working Group ARPA Knowledge Sharing Initiative. Specification of the KQML agent-communication language. Working paper, December 1992.

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

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

[5] Nicholas M. Avouris and Les Gasser, editors. *Distributed Artificial Intelligence: Theory and Praxis.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[6] Madeleine Bates. The theory and practice of augmented transition network grammars. In Leonard Bolc, editor, *Natural Language Communication with Computers*, Lecture Notes in Computer Science, pages 191–260. Morgan Kaufmann, 1979.

[7] 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.

[8] Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufman Publishers, San Mateo, CA, 1988.

[9] 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.

[10] M. Caspari and E. Motta. Symbol-level requirements for agent level programming. (submitted during ECAI's review period), 1993.

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

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

[13] Philip R. Cohen and H.J. Levesque. Communicative actions for artificial agents. In *Proceedings of the International Conference on Multi–Agent Systems*. AAAI Press, June 1995.

[14] Philip R. Cohen, Jerry Morgan, and Martha Pollack, editors. *Intentions in Communication*. MIT Press, Cambridge, MA, 1990.

[15] 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.

[16] 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.

[17] Susan E. Conry, Robert A. Meyer, and Victor R. Lesser. Multistage negotiation in distributed planning. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 367–384. Morgan Kaufman, 1988.

[18] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1008–1014, Philadelphia, PA, August 1986. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 503–518, Addison-Wesley, 1988.).

[19] M. Cutkosky, E. Engelmore, R. Fikes, T. Gruber, M. Genesereth, and W. Mark. PACT: An experiment in integrating concurrent engineering systems. *IEEE Computer*, pages 28–38, January 1993.

[20] 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).

[21] Y. Demazeau and J.-P. Muller, editors. *Decentralized A.I.: Proc. of the First European Workshop on Modelling*. Elsevier Science Publishers B.V./North Holland, Amsterdam, 1990.

[22] Y. Demazeau and J.-P. Muller, editors. *Decentralized A.I. 2: Proc. of the Second European Workshop on Modelling*. Elsevier Science Publishers B.V./North Holland, Amsterdam, 1991.

[23] Edmund H. Durfee and Victor R. Lesser. Using partial global plans to coordinate distributed problem solvers. In *IJCAI*, pages 875–883, 1987. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 285–293, Morgan Kaufmann, 1988).

[24] Edmund H. Durfee and Thomas A. Montgomery. A hierarchical protocol for coordinating multiagent behaviors. In *Proceedings of the 1990 Conference of the American Association for Artificial Intelligence*, pages 86–93, 1990.

[25] E.H. Durfee. The distributed artificial intelligence melting pot. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1301–1306, Nov./Dec. 1991.

[26] 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.

[27] L.D. Erman, J.S. Lark, and Frederick Hayes-Roth. ABE: An environment for engineering intelligent systems. *IEEE Transactions on Software Engineering*, 14(12):1758–1770, Dec. 1988.

[28] Jacques Ferber and Patrice Carl. Actors and agents as reflective concurrent objects: A MERING IV perspective. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1420–1436, Nov./Dec. 1991.

[29] T. Finin, R. Fritzson, and D. McKay et. al. An overview of KQML: A knowledge query and manipulation language. Technical report, Department of Computer Science, University of Maryland Baltimore County, 1992.

[30] Tim Finin, Yannis Labrou, and James Mayfield. KQML as an agent communication language. In Jeffery M. Bradshaw, editor, *Software Agents*. MIT Press, 1996.

[31] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In *International Conference on Building and Sharing of Very Large-Scale Knowledge Bases*, December 1993.

[32] 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.

[33] Tim Finin, Charles Nicholas, and Yelena Yesha, editors. *Information and Knowledge Management: Expanding the Definition of Database*. Lecture Notes in Computer Science 752. Springer–Verlag, 1993. (ISBN 3–540–57419–0).

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

[35] 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.

[36] L. Gasser and J.P. Briot. Object-Based Concurrent Programming and DAI. In Nicholas M. Avouris and Les Gasser, editors, *Distributed Artificial Intelligence: Theory and Praxis*, pages 81–107. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[37] Les Gasser, Carl Braganza, and Nava Herman. MACE: A flexible testbed for distributed AI research. In Michael N. Huhns, editor, *Distributed Artificial Intelligence*, Research Notes in Artificial Intelligence, chapter 5, pages 119–152. Pitman, 1987.

[38] Les Gasser and Michael N. Huhns, editors. *Distributed Artificial Intelligence*. Morgan Kaufman Publishers, San Mateo, CA, 1987.

[39] Les Gasser and Michael N. Huhns, editors. *Distributed Artificial Intelligence, Volume 2*. Morgan Kaufman Publishers, San Mateo, CA, 1989.

[40] M. Genesereth and R. Fikes et. al. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical report, Computer Science Department, Stanford University, 1992.

[41] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, 1994.

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

[43] Matt Ginsberg. Knowledge Interchange Format: The KIF of death. *AI Magazine*, 1991.

[44] C. Hewitt. Viewing control structures as patterns on passing messages. *Artificial Intelligence*, 8(3):323–364, Fall 1977.

[45] 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).

[46] 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).

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

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

[49] Yannis Labrou and Tim Finin. A semantics approach for KQML—a general purpose communication language for software agents. In *Third International Conference on Information and Knowledge Management*, November 1994. Available as `http://www.cs.umbc.edu/kqml/papers/kqml-semantics.ps`.

[50] Douglas B. Lenat. Beings: Knowledge as interacting experts. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 126–133, Stanford, CA, August 1975. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 161–168, Morgan Kaufmann, 1988).

[51] Victor R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, Fall 1983. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 353–386, Addison-Wesley, 1988 and in *Readings from AI Magazine: Volumes 1–5*, Robert Engelmore, editor, pages 69–85, AAAI, Menlo Park, California, 1988).

[52] J. Mayfield, Y. Labrou, and T. Finin. Evaluating KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II—Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL–95)*, Lecture Notes in Artificial Intelligence. Springer–Verlag, 1996.

[53] James Mayfield, Yannis Labrou, and Tim Finin. Desiderata for agent communication languages. In *Proceedings of the 1995 AAAI Spring Symposium on Information Gathering in Distributed Environments*, March 1995.

[54] M.Tenenbaum, J. Weber, and T. Gruber. Enterprise integration: Lessons from SHADE and PACT. In C. Petrie, editor, *Enterprise Integration Modeling.* MIT Press, 1993.

[55] 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.

[56] Allen Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[57] Allen Newell. Reflections on the knowledge level. *Artificial Intelligence*, 59:31–38, 1993.

[58] H. Penny Nii. Blackboard application systems and a knowledge engineering perspective (part 1). *AI Magazine*, August 1986.

[59] Jon Pastor, Don McKay, and Tim Finin. View–concepts: Knowledge–based access to databases. In *Proceedings of the First International Conference on Information and Knowledge Management*, October 1992.

[60] F. Perreira and D. Warren. Definite clause grammars for language analysis. In Barbara J. Grosz, Karen Sparck Jones, and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 101–124. Morgan Kaufmann Publishers, 1986.

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

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

[63] Yoav Shoham. Agent–oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[64] M. P. Singh. Semantical considerations on some primitives for agent specification. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II—Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL–95)*, Lecture Notes in Artificial Intelligence. Springer–Verlag, 1996.

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

[66] M.P. Singh. On the semantics of protocols among distributed intelligent agents. In *Proceedings of the IEEE International Phoenix Conference on Computers and Communications*, 1992.

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

[68] 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).

[69] Reid G. Smith and Randall Davis. Framework for cooperation in distributed problem solving. *IEEE Transactions on System, Man, and Cybernetics*, SMC-11(1):61–70, January 1981.

[70] Chelliah Thirunavukkarasu, Tim Finin, and James Mayfield. Secret agents—a security architecture for the KQML agent communication language. In *Proceedings of the ACM CIKM Intelligent Information Agents Workshop*, December 1995.

[71] D. Vanderveken. On the unification of speech act theory and formal semantics. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication*, pages 195–220. MIT Press, Cambridge, MA, 1990.