8

Communication

Communication has long been recognized as a topic of central importance in computer science, and many formalisms have been developed for representing the properties of communicating concurrent systems (Hoare, 1978; Milner, 1989). Such formalisms have tended to focus on a number of key issues that arise when dealing with systems that can interact with one another.

Perhaps the characteristic problem in communicating concurrent systems research is that of *synchronizing* multiple processes, which was widely studied throughout the 1970s and 1980s (Ben-Ari, 1990). Essentially, two processes (cf. agents) need to be synchronized if there is a possibility that they can interfere with one another in a destructive way. The classic example of such interference is the 'lost update' scenario. Tn this scenario, we have two processes, p_1 and p_2, both of which have access to some shared variable *v*. Process p_1 begins to update the value of *v*, by first reading it, then modifying it (perhaps by simply incrementing the value that it obtained), and finally saving this updated value in *v*. But between p_1 reading and again saving the value of *v*, process p_2 updates *v*, by saving some value in it. When p_1 saves its modified value of *v*, the update performed by p_2 is thus lost, which is almost certainly not what was intended. The lost update problem is a very real issue in the design of programs that communicate through shared data structures.

So, if we do not treat communication in such a 'low-level' way, then how *is* communication treated by the agent community? In order to understand the answer, it is helpful to first consider the way that communication is treated in the object-oriented programming community, that is, communication as method invocation. Suppose we have a Java system containing two objects, o_1 and o_2, and that o_1 has a publicly available method m_1. Object o_2 can communicate with o_1 by invoking method m_1. In Java, this would mean o_2 executing an instruction that looks something like `o1.m1(arg)`, where arg is the argument that o_2 wants to communicate to o_1. But consider: which object makes the decision about the execution of

method m_1? Is it object o_1 or object o_2? In this scenario, object o_1 has *no control* over the execution of m_1: the decision about whether to execute m_1 lies entirely with o_2.

Now consider a similar scenario, but in an agent-oriented setting. We have two agents *i* and *j*, where *i* has the capability to perform action α, which corresponds loosely to a method. But there is no concept in the agent-oriented world of agent *j* 'invoking a method' on *i*. This is because i is an *autonomous agent:* it has control over both its state and its behaviour. It cannot be taken for granted that agent *i* will execute action α just because another agent *j* wants it to. Performing the action α may not be in the best interests of agent *i*. The locus of control with respect to the decision about whether to execute an action is thus very different in agent and object systems.

In general, agents can neither force other agents to perform some action, nor write data onto the internal state of other agents. This does not mean they cannot communicate, however. What they *can* do is perform actions - communicative actions - in an attempt to *influence* other agents appropriately. For example, suppose I say to you 'It is raining in London', in a sincere way. Under normal circumstances, such a communication action is an attempt by me to modify your beliefs. Of course, simply uttering the sentence 'It is raining in London' is not usually enough to bring about this state of affairs, for all the reasons that were discussed above. You have control over your own beliefs (desires, intentions). You may believe that I am notoriously unreliable on the subject of the weather, or even that I am a pathological liar. But in performing the communication action of uttering 'It is raining in London', I am attempting to change your internal state. Furthermore, since this utterance is an action that I perform, I am performing it for some purpose - presumably because I intend that you believe it is raining.

8.1 Speech Acts

Speech act theory treats communication as action. It is predicated on the assumption that speech actions are performed by agents just like other actions, in the furtherance of their intentions.

I begin with an historical overview of speech act theory, focusing in particular on attempts to develop formal theories of speech acts, where communications are modelled as actions that alter the mental state of communication participants.

8.1.1 Austin

The theory of speech acts is generally recognized to have begun with the work of the philosopher John Austin (Austin. 1962). He noted that a certain class of natural language utterances - hereafter referred to as *speech acts* - had the characteristics of *actions,* in the sense that they change the state of the world in a way

analogous to physical actions. It may seem strange to think of utterances changing the world in the way that physical actions do. If I pick up a block from a table (to use an overworked but traditional example), then the world has changed in an obvious way. But how does speech change the world? Austin gave as paradigm examples declaring war and saying 'I now pronounce you man and wife'. Stated in the appropriate circumstances, these utterances clearly change the state of the world in a very tangible way[1].

Austin identified a number of *performative verbs*, which correspond to various ferent types of speech acts. Examples of such performative verbs are *request*, *inform*, and *promise*. In addition, Austin distinguished three different aspects of speech acts: the *locutionary act*, or act of making an utterance (e.g. saying 'Please make some tea'), the *illocutionary act*, or action performed in saying something (e.g. 'He requested me to make some tea'), and *perlocution*, or effect of the act 'He got me to make tea').

Austin referred to the conditions required for the successful completion of performatives as *felicity conditions*. He recognized three important felicity conditions.

(1) There must be an accepted conventional procedure for the performative, and the circumstances and persons must be as specified in the procedure.

(2) The procedure must be executed correctly and completely.

(3) The act must be sincere, and any *uptake* required must be completed, insofar as is possible.

.1.2 Searle

Austin's work was extended by John Searle in his 1969 book *Speech Acts* (Searle, 1969). Searle identified several properties that must hold for a speech act performed between a hearer and a speaker to succeed. For example, consider a *request* by SPEAKER to HEARER to perform ACTION.

(1) Normal I/O conditions. Normal I/O conditions state that HEARER is able to hear the request (thus must not be deaf, etc.), the act was performed in normal circumstances (not in a film or play, etc.), etc.

(2) **Preparatory conditions.** The preparatory conditions state what must be true of the world in order that SPEAKER correctly choose the speech act. In this case, HEARER must be able to perform ACTION, and SPEAKER must believe that HEARER is able to perform ACTION. Also, it must not be obvious that HEARER will do ACTION anyway.

[1]Notice that when referring to the effects of communication, I am ignoring 'pathological' cases, such as shouting while on a ski run and causing an avalanche. Similarly, I will ignore 'microscopic' effects (such as the minute changes in pressure or temperature in a room caused by speaking).

(3) **Sincerity conditions.** These conditions distinguish sincere performances of the request; an insincere performance of the act might occur if SPEAKER did not really want ACTION to be performed.

Searle also attempted a systematic classification of possible types of speech acts, identifying the following five key classes.

(1) Representatives. A representative act commits the speaker to the truth of an expressed proposition. The paradigm case is *informing.*

(2) Directives. A directive is an attempt on the part of the speaker to get the hearer to do something. Paradigm case: *requesting.*

(3) **Commissives.** Commit the speaker to a course of action. Paradigm case: *promising.*

(4) **Expressives.** Express some psychological state (gratitude for example). Paradigm case: *thanking.*

(5) **Declarations.** Effect some changes in an institutional state of affairs. Paradigm case: *declaring war.*

8.1.3 The plan-based theory of speech acts

In the late 1960s and early 1970s, a number of researchers in AI began to build systems that could plan how to autonomously achieve goals (Allen *et al.*, 1990). Clearly, if such a system is required to interact with humans or other autonomous agents, then such plans must include *speech* actions. This introduced the question of how the properties of speech acts could be represented such that planning systems could reason about them. Cohen and Perrault (1979) gave an account of the semantics of speech acts by using techniques developed in AI planning research (Fikes and Nilsson, 1971). The aim of their work was to develop a theory of speech acts

> ...by modelling them in a planning system as operators defined...in terms of speakers' and hearers' beliefs and goals. Thus speech acts are treated in the same way as physical actions.
>
> (Cohen and Perrault, 1979)

The formalism chosen by Cohen and Perrault was the STRIPS notation, in which the properties of an action are characterized via preconditions and postconditions (Fikes and Nilsson, 1971). The idea is very similar to Hoare logic (Hoare, 1969). Cohen and Perrault demonstrated how the preconditions and postconditions of speech acts such as *request* could be represented in a multimodal logic containing operators for describing the *beliefs*, *abilities*, and *wants* of the participants in the speech act.

Consider the *Request* act. The aim of the *Request* act will be for a speaker to get a hearer to perform some action. Figure 8.1 defines the *Request* act. Two preconditions are stated: the 'cando.pr' (can-do preconditions), and 'want.pr' (want preconditions). The cando.pr states that for the successful completion of the *Request,* two conditions must hold. First, the speaker must believe that the hearer of the *Request* is able to perform the action. Second, the speaker must believe that the hearer also believes it has the ability to perform the action. The want.pr states that in order for the *Request* to be successful, the speaker must also believe it actually wants the *Request* to be performed. If the preconditions of the *Request* are fulfilled, then the *Request* will be successful: the result (defined by the 'effect' part of the definition) will be that the hearer believes the speaker believes it wants some action to be performed.

While the successful completion of the *Request* ensures that the hearer is aware of the speaker's desires, it is not enough in itself to guarantee that the desired action is actually performed. This is because the definition of *Request* only models the illocutionary force of the act. It says nothing of the perlocutionary force. What is required is a *mediating act.* Figure 8.1 gives a definition of $CauseToWant$, which is an example of such an act. By this definition, an agent will come to believe it wants to do something if it believes that another agent believes it wants to do it. This definition could clearly be extended by adding more preconditions, perhaps to do with beliefs about social relationships, power structures, etc.

The $Inform$ act is as basic as *Request.* The aim of performing an $Inform$ will be for a speaker to get a hearer to believe some statement. Like *Request,* the definition of $Inform$ requires an associated mediating act to model the perlocutionary force of the act. The cando.pr of $Inform$ states that the speaker must believe *qp* is true. The effect of the act will simply be to make the hearer believe that the speaker believes *qp.* The cando.pr of *Convince* simply states that the hearer must believe that the speaker believes *qp.* The effect is simply to make the hearer believe *qp.*

Speech acts as rational action

While the plan-based theory of speech acts was a major step forward, it was recognized that a theory of speech acts should be rooted in a more general theory of rational action. This observation led Cohen and Levesque to develop a theory in which speech acts were modelled as actions performed by rational agents in the furtherance of their intentions (Cohen and Levesque, 1990b). The foundation upon which they built this model of rational action was their theory of intention, described in Cohen and Levesque (1990a). The formal theory is summarized in Chapter 12, but, for now, here is the Cohen-Levesque definition of *requesting,* paraphrased in English.

> A request is an attempt on the part of $spkr$ by doing e, to bring about a state where, ideally (i) *addr* intends α (relative to the *spkr* still having

Figure 8.1 Definitions from Cohen and Perrault's plan-based theory of speech acts

$Request(S, H, \alpha)$

Preconditions	Cando.pr	*(S BELIEVE (H CANDO a)) A (S BELIEVE (H BELIEVE (H CANDO a)))*
	Want.pr	*{S BELIEVE (S WANT requestInstance))*
Effect		*(H BELIEVE (S BELIEVE {S WANT a)))*

$CauseToWant(A_1, A_2,$

Preconditions	Cando.pr	$(A_1$ *BELIEVE* $(A_2$ *BELIEVE* $(A_2$ *WANT* $\alpha)))$
	Want.pr	x
Effect		$(A_1$ *BELIEVE* $(A_1$ *WANT a))*

$Inform(S, H, \varphi)$

~~Preconditions~~	~~Cando.pr~~	*(S BELIEVE qp)*
	Want.pr	(5 *BELIEVE {S WANT informInstance))*
Effect		*(H BELIEVE {S BELIEVE* φ*))*

$Convince(A_1, A_2, \varphi)$

Preconditions	Cando.pr	$(A_1$ *BELIEVE* $(A_2$ *BELIEVE qp))*
	Want.pr	x
Effect		$(A_1$ *BELIEVE* $\varphi)$

that goal, and *addr* still being helpfully inclined to *spkr*), and (ii) *addr* actually eventually does α, or at least brings about a state where *addr* believes it is mutually believed that it wants the ideal situation.

(Cohen and Levesque, 1990b, p. 241)

8.2 Agent Communication Languages

As I noted earlier, speech act theories have directly informed and influenced a number of languages that have been developed specifically for agent communication. In the early 1990s, the US-based DARPA-funded Knowledge Sharing Effort (KSE) was formed, with the remit of

> [developing] protocols for the exchange of represented knowledge between autonomous information systems.
>
> (Finin *et al.*, 1993)

The KSE generated two main deliverables as follows.

- The *Knowledge Query and Manipulation Language* (KQML). KQML is an 'outer' language for agent communication. It defines an 'envelope' format for messages, using which an agent can explicitly state the intended illocutionary force of a message. KQML is not concerned with the *content* part of messages (Patil *et al.*, 1992; Mayfield *et al.*, 1996).

  The *Knowledge Interchange Format* (KIF). KIF is a language explicitly intended to allow the representation of knowledge about some particular 'domain of discourse'. It was intended primarily (though not uniquely) to form the content parts of KQML messages.

2.1 KIF

I will begin by describing the Knowledge Interchange Format - KIF (Genesereth and Fikes, 1992). This language was originally developed with the intent of being a common language for expressing properties of a particular domain. It was *not* intended to be a language in which messages themselves would be expressed, but rather it was envisaged that the KIF would be used to express message *content*. KIF is closely based on first-order logic (Enderton, 1972; Genesereth and Nilsson, 1987). (In fact, KIF looks very like first-order logic recast in a LISP-like notation; to fully understand the details of this section, some understanding of first-order logic is therefore helpful.) Thus, for example, by using KIF, it is possible for agents to express

> properties of things in a domain (e.g. 'Michael is a vegetarian' - Michael has the property of being a vegetarian);
>
> relationships between things in a domain (e.g. 'Michael and Janine are married' - the relationship of marriage exists between Michael and Janine);
>
> general properties of a domain (e.g. 'everyhody has a mother').

In order to express these things, KIF assumes a basic, fixed logical apparatus, which contains the usual connectives that one finds in first-order logic: the binary Boolean connectives and, or, not, and so on, and the universal and existential quantifiers `forall` and `exists`. In addition, KIF provides a basic vocabulary of objects - in particular, numbers, characters, and strings. Some standard functions and relations for these objects are also provided, for example the 'less than' relationship between numbers, and the 'addition' function. A LISP-like notation is also provided for handling lists of objects. Using this basic apparatus, it is possible to *define* new objects, and the functional and other relationships between

these objects. At this point, some examples seem appropriate. The following KIF expression asserts that the temperature of ml is 83 Celsius:

```
(= (temperature ml) (scalar 83 Celsius))
```

In this expression, = is equality: a relation between two objects in the domain; temperature is a function that takes a single argument, an object in the domain (in this case, ml), and scalar is a function that takes two arguments. The = relation is provided as standard in KIF, but both the temperature and scalar functions must be defined.

The second example shows how definitions can be used to introduced new concepts for the domain, in terms of existing concepts. It says that an object is a bachelor if this object is a man and is not married:

```
(defrelation bachelor (?x) :=
  (and (man ?x)
    (not (married ?x))))
```

In this example, ?x is a variable, rather like a parameter in a programming language. There are two relations: man and married, each of which takes a single argument. The :=: symbol means 'is, by definition'.

The next example shows how relationships between individuals in the domain can be stated - it says that any individual with the property of being a person also has the property of being a mammal:

```
(defrelation person (?x) :=> (mammal ?x))
```

Here, both person and mammal are relations that take a single argument.

8.2.2 KQML

KQML is a message-based language for agent communication. Thus KQML defines a common format for messages. A KQML message may crudely be thought of as an object (in the sense of object-oriented programming): each message has a *performative* (which may be thought of as the class of the message), and a number of *parameters* (attribute/value pairs, which may be thought of as instance variables).

Here is an example KQML message:

```
(ask-one
   :content  (PRICE IBM ?price)
   :receiver stock-server
   :language LPROLOG
   :ontology NYSE-TICKS
)
```

The intuitive interpretation of this message is that the sender is asking about the price of IBM stock. The performative is ask-one, which an agent will use to

Table 8.1 Parameters for KQML messages.

Parameter	Meaning
:content	content of the message
:force	whether the sender of the message will ever deny the content of the message
:reply-with	whether the sender expects a reply, and, if so, an identifier for the reply
:in-reply-to	reference to the :reply-with parameter
:sender	sender of the message
:receiver	intended recipient of the message

ask a question of another agent where exactly one reply is needed. The various other components of this message represent its attributes. The most important of these is the :content field, which specifies the message content. In this case, the content simply asks for the price of IBM shares. The :receiver attribute specifies the intended recipient of the message, the :1anguage attribute specifies that the language in which the content is expressed is called LPROLOG (the recipient is assumed to 'understand' LPROLOG), and the final :ontology attribute defines the *terminology* used in the message - we will hear more about ontologies later in this chapter. The main parameters used in KQML messages are summarized in Table 8.1; note that different performatives require different sets of parameters.

Several different versions of KQML were proposed during the 1990s, with different collections of performatives in each. In Table 8.2, I summarize the version of KQML performatives that appeared in Finin *et al.* (1993); this version contains a total of 41 performatives. In this table, *S* denotes the :sender of lessages, *R* denotes the :receiver, and *C* denotes the content of the message.

To more fully understand these performatives, it is necessary to understand the notion of a *virtual knowledge base* (VKB) as it was used in KQML. The idea was that agents using KQML to communicate may be implemented using different programming languages and paradigms - and, in particular, any information that agents have may be *internally* represented in many different ways. No agent can assume that another agent will use the same internal representation; indeed, no actual 'representation' may be present in an agent at all. Nevertheless, for the purposes of communication, it makes sense for agents to treat other agents *as if* they had some internal representation of knowledge. Thus agents *attribute* knowledge to other agents; this attributed knowledge is known as the virtual knowledge base.

Table 8.2 KQML performatives.

Performative	Meaning
achieve	*S* wants *R* to make something true of their environment
advertise	*S* claims to be suited to processing a performative
ask-about	*S* wants all relevant sentences in *R's* VKB
ask-all	*S* wants all of *R's* answers to a question C
ask-if	*S* wants to know whether the answer to *C* is in *R's* VKB
ask-one	*S* wants one of *R's* answers to question C
break	*S* wants *R* to break an established pipe
broadcast	*S* wants *R* to send a performative over all connections
broker-all	*S* wants *R* to collect all responses to a performative
broker-one	5 wants *R* to get help in responding to a performative
deny	the embedded performative does not apply to *S* (anymore)
delete-all	*S* wants *R* to remove all sentences matching C from its VKB
delete-one	*S* wants *R* to remove one sentence matching *C* from its VKB
discard	*S* will not want *R's* remaining responses to a query
eos	end of a stream response to an earlier query
error	*S* considers *R's* earlier message to be malformed
evaluate	*S* wants *R* to evaluate (simplify) *C*
forward	*S* wants *R* to forward a message to another agent
generator	same as a standby of a stream-all
insert	*S* asks *R* to add content to its VKB
monitor	*S* wants updates to *R's* response to a stream-all
next	wants *R's* next response to a previously streamed performative
pipe	*S* wants *R* to route all further performatives to another agent
ready	*S* is ready to respond to *R's* previously mentioned performative
recommend-all	*S* wants all names of agents who can respond to C
recommend-one	*S* wants the name of an agent who can respond to a *C*
recruit-all	*S* wants *R* to get all suitable agents to respond to C
recruit-one	*S* wants *R* to get one suitable agent to respond to C
register	*S* can deliver performatives to some named agent
reply	communicates an expected reply
rest	*S* wants *R's* remaining responses to a previously named performative
sorry	*S* cannot provide a more informative reply
standby	*S* wants *R* to be ready to respond to a performative
stream-about	multiple response version of ask-about
stream-all	multiple response version of ask-all
subscribe	*S* wants updates to *R's* response to a performative
tell	*S* claims to *R* that *C* is in *S's* VKB
transport-address	*S* associates symbolic name with transport address
unregister	the deny of a register
untell	*S* claims to *R* that *C* is *not* in *S's* VKB

Dialogue (a)

```
(evaluate
  :sender A :receiver B
  :language KIF :ontology motors
  :reply-with q1 :content (val (torque m1)))
(reply
  :sender B :receiver A
  :language KIF :ontology motors
  :in-reply-to q1 :content (= (torque m1) (scalar 12 kgf)))
```

Dialogue (b)

```
(stream-about
  :sender A :receiver B
  :language KIF :ontology motors
  :reply-with q1 :content m1)
(tell
  :sender B :receiver A
  :in-reply-to q1 :content (= (torque m1) (scalar 12 kgf)))
(tell
  :sender B :receiver A
  :in-reply-to q1 :content (= (status m1) normal))
(eos
  :sender B :receiver A
  :in-reply-to q1)
```

Figure 8.2 Example KQML Dialogues.

Example KQML dialogues

To illustrate the use of KQML, we will now consider some example KQML dialogues (these examples are adapted from Finin *et ah* (1993)). In the first dialogue (Figure 8.2(a)), agent A sends to agent B a query, and subsequently gets a response to this query. The query is the value of the torque on ml; agent A gives the query the name `q1` so that B can later refer back to this query when it responds. Finally, the `:ontology` of the query is motors - as might be guessed, this ontology defines a terminology relating to motors. The response that B sends indicates that the torque of ml is equal to 12 kgf - a scalar value.

The second dialogue (Figure 8.2(b)) illustrates a *stream* of messages: agent A asks agent B for everything it knows about ml. Agent B responds with two `tell` messages, indicating what it knows about ml, and then sends an eos (end of stream) message, indicating that it will send no more messages about ml. The first `tell` message indicates that the torque of ml is 12 kgf (as in dialogue (a)); the second `tell` message indicates that the status of ml is normal. Note that there is no content to the eos message; eos is thus a kind of meta-message - a message about messages.

```
Dialogue (c)
(advertise
  :sender A
  :language KQML :ontology K10
  :content
    (subscribe
      :language KQML :ontology K10
      :content
        (stream-about
          :language KIF :ontology motors
          :content ml)))
(subscribe
  :sender B :receiver A
  :reply-with s1
  :content
    (stream-about
      :language KIF :ontology motors
      :content ml))
(tell
  :sender A :receiver B
  :in-reply-to s1 :content (= (torque ml) (scalar 12 kgf)))
(tell
  :sender A :receiver B
  :in-reply-to s1 :content (= (status ml) normal))
(untell
  :sender A :receiver B
  :in-reply-to s1 :content (= (torque ml) (scalar 12 kgf)))
(tell
  :sender A :receiver B
  :in-reply-to s1 :content (= (torque ml) (scalar 15 kgf)))
(eos
  :sender A :receiver B
  :in-reply-to s1)
```

Figure 8.3 Another KQML dialogue.

The third (and most complex) dialogue, shown in Figure 8.3, shows how KQML messages themselves can be the content of KQML messages. The dialogue begins when agent A advertises to agent B that it is willing to accept subscriptions relating to ml. Agent B responds by subscribing to agent A with respect to ml. Agent A then responds with sequence of messages about ml; as well as including `tell` messages, as we have already seen, the sequence includes an `untell` message, to the effect that the torque of ml is no longer 12 kgf, followed by a `tell` message indicating the new value of torque. The sequence ends with an end of stream message.

The take-up of KQML by the multiagent systems community was significant, and several KQML-based implementations were developed and distributed. Despite this success, KQML was subsequently criticized on a number of grounds as follows.

- The basic KQML performative set was rather fluid - it was never tightly constrained, and so different implementations of KQML were developed that could not, in fact, interoperate.
- Transport mechanisms for KQML messages (i.e. ways of getting a message from agent *A* to agent *B)* were never precisely defined, again making it hard for different KQML-talking agents to interoperate.
- The semantics of KQML were never rigorously defined, in such a way that it was possible to tell whether two agents claiming to be talking KQML were in fact using the language 'properly'. The 'meaning' of KQML performatives was only defined using informal, English language descriptions, open to different interpretations. (I discuss this issue in more detail later on in this chapter.)
- The language was missing an entire class of performatives - *commissives*, by which one agent makes a commitment to another. As Cohen and Levesque point out, it is difficult to see how many multiagent scenarios could be implemented without commissives, which appear to be important if agents are to *coordinate* their actions with one another.
- The performative set for KQML was overly large and, it could be argued, rather *ad hoc.*

These criticisms - amongst others - led to the development of a new, but rather closely related language by the FIPA consortium.

2.3 The FIPA agent communication languages

In 1995, the Foundation for Intelligent Physical Agents (FIPA) began its work on developing standards for agent systems. The centerpiece of this initiative was the development of an ACL (FIPA, 1999). This ACL is superficially similar to KQML: it defines an 'outer' language for messages, it defines 20 performatives (such as `inform`) for defining the intended interpretation of messages, and it does not mandate any specific language for message content. In addition, the concrete syntax for FIPA ACL messages closely resembles that of KQML. Here is an example of a FIPA ACL message (from FIPA, 1999, p. 10):

```
(inform
   :sender    agent1
   :receiver  agent2
   :content   (price good2 150)
   :language  sl
   :ontology  hpl-auction
)
```

Table 8.3 Performatives provided by the FIPA communication language.

Performative	Passing information	Requesting information	Negotiation	Performing actions	Error handling
accept-proposal			×		
agree				×	
cancel		×		×	
cfp			×		
confirm	×				
disconfirm	×				
failure					×
inform	×				
inform-if	×				
inform-ref	×				
not-understood					×
propagate				×	
propose			×		
proxy				×	
query-if		×			
query-ref		×			
refuse				×	
reject-proposal			×		
request				×	
request-when				×	
request-whenever				×	
subscribe		×			

As should be clear from this example, the FIPA communication language is similar to KQML: the structure of messages is the same, and the message attribute fields are also very similar. The relationship between the FIPA ACL and KQML is discussed in FIPA (i999, pp. 68, 69). The most important difference between the two languages is in the collection of performatives they provide. The performatives provided by the FIPA communication language are categorized in Table 8.3.

Informally, these performatives have the following meaning.

accept-proposal The accept-proposal performative allows an agent to state that it accepts a proposal made by another agent.

agree An accept performative is used by one agent to indicate that it has acquiesced to a request made by another agent. It indicates that the sender of the agree message intends to carry out the requested action.

cancel A cancel performative is used by an agent to follow up to a previous request message, and indicates that it no longer desires a particular action to be carried out.

cfp A cfp (call for proposals) performative is used to initiate negotiation between agents. The content attribute of a `cfp` message contains both an

action (e.g. 'sell me a car') and a condition (e.g. 'the price of the car is less than US$10 000'). Essentially, it says 'here is an action that I wish to be carried out, and here are the terms under which I want it to be carried out - send me your proposals'. (We will see in the next chapter that the cfp message is a central component of *task-sharing* systems such as the *Contract Net.)*

confirm The confirm performative allows the sender of the message to confirm the truth of the content to the recipient, where, before sending the message, the sender believes that the recipient is unsure about the truth or otherwise of the content.

disconfirm Similar to confirm, but this performative indicates to a recipient that is unsure as to whether or not the sender believes the content that the content is in fact false.

failure This allows an agent to indicate to another agent that an attempt to perform some action (typically, one that it was previously requested to perform) failed.

inform Along with request, the inform performative is one of the two most important performatives in the FIPA ACL. It is the basic mechanism for communicating information. The content of an inform performative is a statement, and the idea is that the sender of the inform wants the recipient to believe this content. Intuitively, the sender is also implicitly stating that *it* believes the content of the message.

inform-if An inform-if implicitly says either that a particular statement is true or that it is false. Typically, an inform-if performative forms the content part of a message. An agent will send a request message to another agent, with the content part being an inform-if message. The idea is that the sender of the request is saying 'tell me if the content of the inform-if is either true or false'.

inform-ref The idea of inform-ref is somewhat similar to that of inform-if: the difference is that rather than asking whether or not an expression is true or false, the agent asks for the *value* of an expression.

not-understood This performative is used by one agent to indicate to another agent that it recognized that it performed some action, but did not understand why this action was performed. The most common use of not-understood is for one agent to indicate to another agent that a message that was just received was not understood. The content part of a not-understood message consists of both an action (the one whose purpose was not understood) and a statement, which gives some explanation of why it was not understood. This performative is the central error-handling mechanism in the FIPA ACL.

propagate The content attribute of a propagate message consists of two things: another message, and an expression that denotes a set of agents. The idea is that the recipient of the propagate message should send the embedded message to the agent(s) denoted by this expression.

propose This performative allows an agent to make a proposal to another agent, for example m response to a cfp message that was previously sent out.

proxy The proxy message type allows the sender of the message to treat the recipient of the message as a proxy for a set of agents. The content of a proxy message will contain both an embedded message (one that it wants forwarded to others) and a specification of the agents that it wants the message forwarded to.

query-if This performative allows one agent to ask another whether or not some specific statement is true or not. The content of the message will be the statement that the sender wishes to enquire about.

query-ref This performative is used by one agent to determine a specific value for an expression (cf. the evaluate performative in KQML).

refuse A refuse performative is used by one agent to state to another agent that it will not perform some action. The message content will contain both the action and a sentence that characterizes why the agent will not perform the action.

reject-proposal Allows an agent to indicate to another that it does not accept a proposal that was made as part of a negotiation process. The content specifies both the proposal that is being rejected, and a statement that characterizes the reasons for this rejection.

request The second fundamental performative allows an agent to request another agent to perform some action.

request-when The content of a request-when message will be both an action and a statement; the idea is that the sender wants the recipient to carry out the action when the statement is true (e.g. 'sound the bell when the temperature falls below 20 Celsius').

request-whenever Similar to request-when, the idea is that the recipient should perform the action *whenever* the statement is true.

subscribe Essentially as in KQML: the content will be a statement, and the sender wants to be notified whenever something relating to the statement changes.

Given that one of the most frequent and damning criticisms of KQML was the lack of an adequate semantics, it is perhaps not surprising that the developers of the FIPA agent communication language felt it important to give a comprehensive formal semantics to their language. The approach adopted drew heavily

on Cohen and Levesque's theory of speech acts as rational action (Cohen and Levesque, 1990b), but in particular on Sadek's enhancements to this work (Bretier and Sadek, 1997). The semantics were given with respect to a formal language called SL. This language allows one to represent *beliefs*, *desires*, and *uncertain beliefs* of agents, as well as the actions that agents perform. The semantics of the FIPA ACL map each ACL message to a formula of SL, which defines a constraint that the sender of the message must satisfy if it is to be considered as conforming to the FIPA ACL standard. FIPA refers to this constraint as the *feasibility* condition. The semantics also map each message to an SL-formula that defines the *rational effect* of the action - the 'purpose' of the message: what an agent will be attempting to achieve in sending the message (cf. perlocutionary act). However, in a society of autonomous agents, the rational effect of a message cannot (and should not) be guaranteed. Hence conformance does not require the recipient of a message to respect the rational effect part of the ACL semantics - only the feasibility condition.

As I noted above, the two most important communication primitives in the FIPA languages are `inform` and `request`. In fact, *all* other performatives in FIPA are defined in terms of these performatives. Here is the semantics for inform (FIPA, 1999, p. 25):

$$\langle i, inform(j, \varphi)\rangle$$
feasibility precondition: $B_i\varphi$ A v *Uifjqp)*
rational effect: $B_j\varphi$. (8.1)

The $B_i\varphi$ means 'agent i believes *qp'; $Bif_i\varphi$* means that 'agent i has a definite opinion one way or the other about the truth or falsity of *qp';* and $Uif_i\varphi$ means ent i is 'uncertain' about *qp.* Thus an agent i sending an *inform* message with content *qp* to agent j will be respecting the semantics of the FIPA ACL if it believes *qp,* and it is not the case that it believes of j either that j believes whether *qp* is true or false, or that j is uncertain of the truth or falsity of *qp.* If the agent is *successful* in performing the inform, then the recipient of the message - agent j - will believe *qp.*

The semantics of `request` are as follows[2]:

$$\langle i, request(j, \alpha)\rangle$$
feasibility precondition: $B_i Agent(\alpha, j)$A
rational effect: $Done(\alpha)$. (8.2)

The SL expression $Agent(\alpha, j)$ means that the agent of action α is j (i.e. j is the agent who performs α); and $Done(\alpha)$ means that the action α has been done. Thus agent i requesting agent j to perform action α means that agent i believes that the agent of α is j (and so it is sending the message to the right agent), and

[2] In the interests of comprehension, I have simplified the semantics a little.

agent *i* believes that agent *j* does not currently intend that α is done. The rational effect - what *i* wants to achieve by this - is that the action is done.

One key issue for this work is that of *semantic conformance testing.* The conformance testing problem can be summarized as follows (Wooldridge, 1998). We are given an agent, and an agent communication language with some well-defined semantics. The aim is to determine whether or not the agent respects the semantics of the language whenever it communicates. *Syntactic* conformance testing is of course easy - the difficult part is to see whether or not a particular agent program respects the *semantics* of the language.

The importance of conformance testing *has* been recognized by the ACL community (FIPA, 1999, p. 1). However, to date, little research has been carried out either on how verifiable communication languages might be developed, or on how existing ACLs might be verified. One exception is (my) Wooldridge (1998), where the issue of conformance testing is discussed from a formal point of view: I point out that ACL semantics are generally developed in such a way as to express *constraints* on the senders of messages. For example, the constraint imposed by the semantics of an 'inform' message might state that the sender believes the message content. This constraint can be viewed as a *specification.* Verifying that an agent respects the semantics of the agent communication language then reduces to a conventional program verification problem: show that the agent sending the message satisfies the specification given by the communication language semantics. But to solve this verification problem, we would have to be able to talk about the mental states of agents - what they believed, intended and so on. Given an agent implemented in (say) Java, it is not clear how this might be done.

8.3 Ontologies for Agent Communication

One issue that I have rather glossed over until now has been that of *ontologies.* The issue of ontologies arises for the following reason. If two agents are to communicate about some domain, then it is necessary for them to agree on the *terminology* that they use to describe this domain. For example, imagine an agent is buying a particular engineering item (nut or bolt) from another agent: the buyer needs to be able to unambiguously specify to the seller the desired properties of the item, such as its size. The agents thus need to be able to agree both on what 'size' means, and also what terms like 'inch' or 'centimetre' mean. An *ontology* is thus a specification of a set of terms as follows.

> An ontology is a formal definition of a body of knowledge. The most typical type of ontology used in building agents involves a structural component. Essentially a taxonomy of class and subclass relations coupled with definitions of the relationships between these things.
>
> (Jim Hendler)

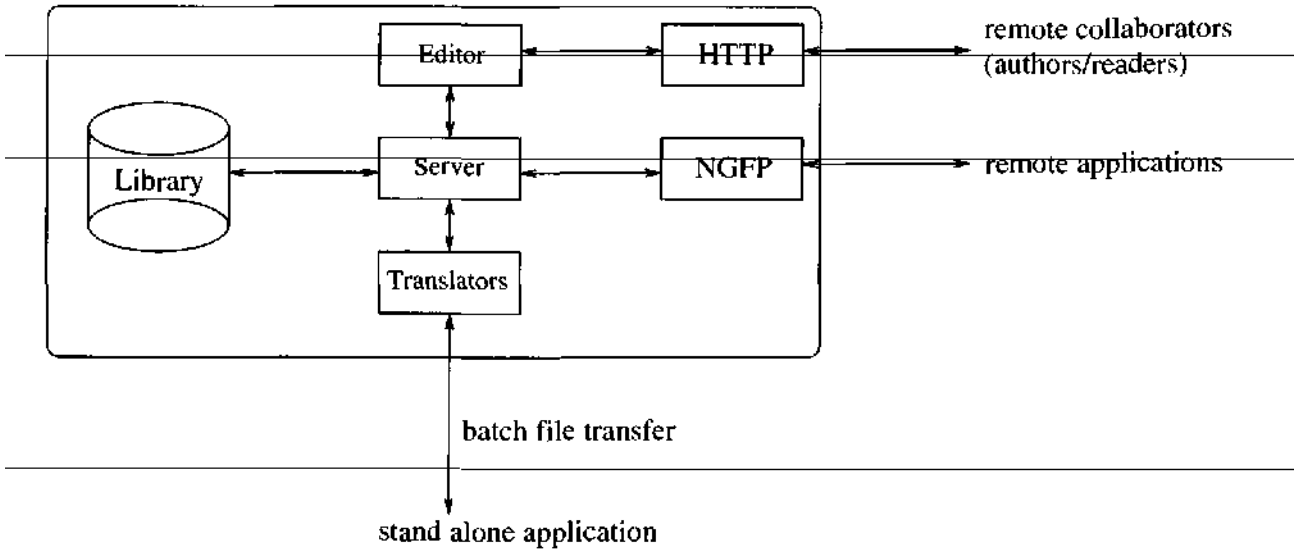

Figure 8.4 Architecture of the Ontolingua server.

In fact, we have already seen an example of a language for defining ontologies: KIF (Genesereth and Fikes, 1992). By using KIF, we can declaratively express the properties of a domain and the relationships between the things in this domain. As KIF was not primarily intended as a language for use by humans, but for processing by computers, software tools were developed that allow a user KIF ontologies - of these, the best known is the Ontolingua server (Farquhar *et al.*, 1997). The Ontolingua server is a Web-based service that is intended to provide a common platform in which ontologies developed by different groups can be shared, and perhaps a common view of these ontologies achieved.

The structure of the Ontolingua server is illustrated in Figure 8.4. The central component is a library of ontologies, expressed in the Ontolingua ontology definition language (based on KIF). A server program provides access to this library. The library may be accessed through the server in several different ways: either by editing it directly (via a Web-based interface), or by programs that contact the server remotely via the NGFP interface. The Ontolingua server was capable of automatically transforming ontologies expressed in one format to a variety of others (e.g. the CORBA Interface Definition Language -

As I noted above, KIF is very closely based on first-order logic, which gives it a clean, well-understood semantics, and in addition means that it is extremely expressive (with sufficient ingenuity, pretty much any kind of knowledge can be expressed in first-order logic). However, many other languages and tools have been developed for expressing ontologies. Perhaps the most important of these at the time of writing is the Xtensible Markup Language (XML, 2001) and its close relative, the DARPA Agent Markup Language (DAML, 2001). To understand how XML and DAML came about, it is necessary to look at the history of the Web. The Web essentially comprises two things: a protocol (HTTP), which pro-

vides a common set of rules for enabling Web servers and clients to communicate with one another, and a format for documents called (as I am sure you know!) the Hypertext Markup Language (HTML). Now HTML essentially defines a grammar for interspersing documents with *markup commands*. Most of these markup commands relate to document layout, and thus give indications to a Web browser of how to display a document: which parts of the document should be treated as section headers, emphasized text, and so on. Of course, markup is not restricted to layout information: programs, for example in the form of JavaScript code, can also be attached. The grammar of HTML is defined by a *Document Type Declaration* (DTD). A DTD can be thought of as being analogous to the formal grammars used to define the syntax of programming languages. The HTML DTD thus defines what constitutes a syntactically acceptable HTML document. A DTD is in fact itself expressed in a formal language - the Standard Generalized Markup Language (SGML, 2001). SGML is essentially a language for defining other languages.

Now, to all intents and purposes, the HTML standard is fixed, in the sense that you cannot arbitrarily introduce tags and attributes into HTML documents that were not defined in the HTML DTD. But this severely limits the usefulness of the Web. To see what I mean by this, consider the following example. An e-commerce company selling CDs wishes to put details of its prices on its Web page. Using conventional HTML techniques, a Web page designer can only markup the document with layout information (see, for example, Figure 8.5(a)). But this means that a Web browser - or indeed any program that looks at documents on the Web - has no way of knowing which parts of the document refer to the titles of CDs, which refer to their prices, and so on. Using XML it is possible to define *new* markup tags - and so, in essence, to extend HTML. To see the value of this, consider Figure 8.5(b), which shows the same information as Figure 8.5(a), expressed using new tags (`catalogue`, `product`, and so on) that were defined using XML. Note that new tags such as these cannot be arbitrarily introduced into HTML documents: they must be defined. The way they are defined is by writing an XML DTD: thus XML, like SGML, is a language for defining languages. (In fact, XML is a subset of SGML.)

I hope it is clear that a computer program would have a much easier time *understanding the meaning* of Figure 8.5(b) than Figure 8.5(a). In Figure 8.5(a), there is nothing to help a program understand which part of the document refers to the price of the product, which refers to the title of the product, and so on. In contrast, Figure 8.5(b) makes all this explicit.

XML was developed to answer one of the longest standing critiques of the Web: the lack of *semantic markup*. Using languages like XML, it becomes possible to add information to Web pages in such a way that it becomes easy for computers not simply to display it, but to process it in meaningful ways. This idea led Tim Berners-Lee, widely credited as the inventor of the Web, to develop the idea of the *semantic Web*.

(a) Plain HTML

```
<ul>
   <li><em>Music</em>,
      <b>Madonna<b>,
      USD12<br><p>
   <li><em>Get Ready</em>,
      <b>New Order</b>,
      USD14<br><p>
</ul>
```

(b) XML

```
<catalogue>
   <product type="CD">
      <title>Music</title>
      <artist>Madonna</artist>
      <price currency="USD">12</price>
   </product>
   <product type="CD">
      <title>Get Ready</title>
      <artist>New Order</artist>
      <price currency="USD">14</price>
   </product>
</catalogue>
```

Figure 8.5 Plain HTML versus XML.

> I have a dream for the Web [in which computers] become capable of analysing all the data on the Web - the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialise.
>
> (Berners-Lee, 1999, pp. 169, 170)

In an attempt to realize this vision, work has begun on several languages and tools - notably the Darpa Agent Markup Language (DAML, 2001), which is based on XML. A fragment of a DAML ontology and knowledge base (from the DAML version of the CIA world fact book (DAML, 2001)) is shown in Figure 8.6.

Coordination Languages

One of the most important precursors to the development of multiagent systems was the blackboard model (Engelmore and Morgan, 1988). Initially developed as

```
<rdf:Description rdf:ID="UNITED-KINGDOM">
   <rdf:type rdf:resource="GEOREF"/>
   <HAS-TOTAL-AREA>
      (* 244820 Square-Kilometer)
   </HAS-TOTAL-AREA>
   <HAS-LAND-AREA>
      (* 241590 Square-Kilometer)
   </HAS-LAND-AREA>
   <HAS-COMPARATIVE-AREA-DOC>
      slightly smaller than Oregon
   </HAS-COMPARATIVE-AREA-DOC>
   <HAS-BIRTH-RATE>
      13.18
   </HAS-BIRTH-RATE>
   <HAS-TOTAL-BORDER-LENGTH>
      (* 360 Kilometer)
   </HAS-TOTAL-BORDER-LENGTH>
   <HAS-BUDGET-REVENUES>
      (* 3.255E11 Us-Dollars)
   </HAS-BUDGET-REVENUES>
   <HAS-BUDGET-EXPENDITURES>
      (* 4.009E11 Us-Dollars)
   </HAS-BUDGET-EXPENDITURES>
   <HAS-BUDGET-CAPITAL-EXPENDITURES>
      (* 3.3E10 Us-Dollars)
   </HAS-BUDGET-CAPITAL-EXPENDITURES>
   <HAS-CLIMATE-DOC>
      more than half of the days are overcast
   </HAS-CLIMATE-DOC>
   <HAS-COASTLINE-LENGTH>
      (* 12429 Kilometer)
   </HAS-COASTLINE-LENGTH>
   <HAS-CONSTITUTION-DOC>
      unwritten; partly statutes, partly common law
   </HAS-CONSTITUTION-DOC>
</rdf:Description>
```

Figure 8.6 Some facts about the UK, expressed in DAML.

part of the Hearsay speech understanding project, the blackboard model proposes that group problem solving proceeds by a group of 'knowledge sources' (agents) observing a shared data structure known as a blackboard: problem solving proceeds as these knowledge sources contribute partial solutions to the problem. In the 1980s, an interesting variation on the blackboard model was proposed within the programming language community. This variation was called Linda (Gelernter, 1985; Carriero and Gelernter, 1989).

Strictly speaking, Linda is not a programming language. It is the generic name given to a collection of programming language constructs, which can be used to

implement blackboard-like systems. The core of the Linda model - corresponding loosely to a blackboard - is the *tuple space.* A tuple space is a shared data structure, the components of which are *tagged tuples.* Here is an example of a tagged tuple:

⟨"person","mjw",35⟩.

A tuple may be thought of as a list of data elements. The first of these is the *tag* of the tuple, which corresponds loosely to a class in object-oriented programming. In the example above, the tag is 'person', suggesting that this tuple records information about a person. The remainder of the elements in the tuple are data values.

Processes (agents) who can see the tuple space can access it via three instructions (Table 8.4). The out operation is the simplest: the expressions that are parameters to the operation are evaluated in turn, and the tagged tuple that results is deposited into the tuple space. The i n and out operations allow a process to access the tuple space. The idea of the i n operation is that the parameters to it may either be expressions or parameters of the form ?v, where v is a variable name. When an instruction

```
in("tag", field1, ..., fieldN)
```

is executed, then each of the expressions it contains is evaluated in turn. When this is done, the process that is executing the instruction waits (*blocks*) until a *matching* tuple is in the tuple space. For example, suppose that the tuple space contained the single person tuple above, and that a process attempted to execute the following instruction:

```
in("person", "mjw", ?age).
```

Then this operation would succeed, and the variable age would subsequently have the value 35. If, however, a process attempted to execute the instruction

```
in("person", "sdp", ?age),
```

then the process would block until a tuple whose tag was "person" and whose first data element was "sdp" appeared in the tuple space. (If there is more than one matching tuple in the tuple space, then one is selected at random.)

The rd operation is essentially the same as i n except that it does not remove the tuple from the tuple space - it simply copies the data elements into fields.

Despite its simplicity, Linda turns out be a very simple and intuitive language for developing complex distributed applications that must be coordinated with one another.

Notes and Further Reading

The problems associated with communicating concurrent systems have driven a significant fraction of research into theoretical computer science since the early

Table 8.4 Operations for manipulating Linda tuple spaces.

Operation	Meaning
`out("tag", expr1, ..., exprN)`	evaluate `expr1,..., exprN` and deposit resulting tuple in tuple space
`in("tag", field1, ..., fieldN)`	wait until matching tuple occupies tuple space, then remove it, copying its values into fields
`rd("tag", field1, ..., fieldN)`	wait until matching tuple occupies tuple space, then copy its values into fields

1980s. Two of the best-known formalisms developed in this period are Tony Hoare's Communicating Sequential Processes (CSPs) (Hoare, 1978), and Robin Milner's Calculus of Communicating Systems (CCS) (Milner, 1989). Temporal logic has also been widely used for reasoning about concurrent systems - see, for example, Pnueli (1986) for an overview. A good reference, which describes the key problems in concurrent and distributed systems, is Ben-Ari (1990).

The plan-based theory of speech acts developed by Cohen and Perrault made speech act theory accessible and directly usable to the artificial intelligence community (Cohen and Perrault, 1979). In the multiagent systems community, this work is arguably the most influential single publication on the topic of speech act-like communication. Many authors have built on its basic ideas. For example, borrowing a formalism for representing the mental state of agents that was developed by Moore (1990), Douglas Appelt was able to implement a system that was capable of planning to perform speech acts (Appelt, 1982, 1985).

Many other approaches to speech act semantics have appeared in the literature. For example, Perrault (1990) described how Reiter's default logic (Reiter, 1980) could be used to reason about speech acts. Appelt gave a critique of Perrault's work (Appelt and Konolige, 1988, pp. 167, 168), and Konolige proposed a related technique using hierarchic auto-epistemic logic (HAEL) (Konolige, 1988) for reasoning about speech acts. Galliers emphasized the links between speech acts and AMG belief revision (Gardenfors, 1988): she noted that the changes in a hearer's state caused by a speech act could be understood as analogous to an agent revising its beliefs in the presence of new information (Galliers, 1991). Singh developed a theory of speech acts (Singh, 1991c, 1993) using his formal framework for representing rational agents (Singh, 1990a,b, 1991a,b, 1994, 1998b; Singh and Asher, 1991). He introduced a predicate $comm(i, m)$ to represent the fact that agent i communicates message m to agent j, and then used this predicate to define the semantics of assertive, directive, commissive, and permissive speech acts.

Dignum and Greaves (2000) is a collection of papers on agent communication languages. As I mentioned in the main text of the chapter, a number of KQML implementations have been developed: well-known examples are InfoSleuth(Nodine and Unruh, 1998), KAoS (Bradshaw *et al.*, 1997) and JATLite (Jeon *et*

al, 2000)). Several FIPA implementations have also been developed, of which the Java-based Jade system is probably the best known (Poggi and Rimassa, 2001).

A critique of KIF was published as Ginsberg (1991), while a critique of KQML appears in Cohen and Levesque (1995). A good general survey of work on ontologies (up to 1996) is Uschold and Gruninger (1996). There are many good online references to XML, DAML and the like: a readable published reference is Decker *et al.* (2000). The March/April 2001 issue of *IEEE Intelligent Systems* magazine contained a useful collection of articles on the semantic web (Fensel and Musen, 2001), in the semantic Web (Hendler, 2001), and the OIL language for ontologies on the semantic Web (Fensel *et al.*, 2001).

Recently, a number of proposals have appeared for communication languages with a verifiable semantics (Singh, 1998a; Pitt and Mamdani, 1999; Wooldridge, 1999). See Labrou *et al.* (1999) for a discussion of the state of the art in agent communication languages as of early 1999.

Coordination languages have been the subject of much interest by the theoretical computer science community: a regular conference is now held on the subject, the proceedings of which were published as Ciancarini and Hankin (1996). Interestingly, the Linda model has been implemented in the JavaSpaces package ian *et al,* 1999), making it possible to use the model with Java/JINI systems (Oaks and Wong, 2000).

Class discussion: Cohen and Perrault (1979). A nice introduction to speech acts and the semantics of speech acts, this paper was hugely influential, and although it was written for a natural language understanding audience, it is easy to make sense of.