1 **FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS**

2

3

4 **FIPA 97 Specification**

5 **Part 2**

6

7 # Agent Communication Language

8

9 ***Obsolete***

10

11 Publication date: 10th October, 1997

Contents

129

130   **Foreword**

131   The Foundation for Intelligent Physical Agents (FIPA) is a non-profit association registered in Geneva, Switzerland.
132   FIPA's purpose is to promote the success of emerging agent-based applications, services and equipment. This goal is
133   pursued by making available in a timely manner, internationally agreed specifications that maximise interoperability
134   across agent-based applications, services and equipment. This is realised through the open international collaboration
135   of member organisations, which are companies and universities active in the agent field. FIPA intends to make the
136   results of its activities available to all interested parties and to contribute the results of its activities to appropriate formal
137   standards bodies.

138   This specification has been developed through direct involvement of the FIPA membership. The 35 corporate members
139   of FIPA (October 1997) represent 12 countries from all over the world

140   Membership in FIPA is open to any corporation and individual firm, partnership, governmental body or international
141   organisation without restriction. By joining FIPA each Member declares himself individually and collectively committed
142   to open competition in the development of agent-based applications, services and equipment. Associate Member
143   status is usually chosen by those entities who do want to be members of FIPA without using the right to influence the
144   precise content of the specifications through voting.

145   The Members are not restricted in any way from designing, developing, marketing and/or procuring agent-based
146   applications, services and equipment. Members are not bound to implement or use specific agent-based standards,
147   recommendations and FIPA specifications by virtue of their participation in FIPA.

148   This specification is published as FIPA 97 ver. 1.0 after two previous versions have been subject to public comments
149   following disclosure on the WWW. It has undergone intense review by members as well non-members. FIPA is now
150   starting a validation phase by encouraging its members to carry out field trials that are based on this specification.
151   During 1998 FIPA will publish FIPA 97 ver. 2.0 that will incorporate whatever adaptations will be deemed necessary to
152   take into account the results of field trials.

153   This document forms part two of the FIPA '97 specification. It should be read in conjunction with parts one (Agent
154   Management) and three (Agent/Software Interaction). Part One, *Agent Management* details standards for agent
155   naming, message transport mechanisms and possible failures, and agent registration. Part Three, *Agent/Software*
156   *Integration* details how agent systems can inter-operate successfully with non-agent software systems, such as
157   databases and legacy applications.

158   This release of part two of the FIPA 97 specification cancels and replaces all previous draft versions.

159

159 Introduction

160 The FIPA 97 specification is the first output of the Foundation for Intelligent Physical Agents. It provides specification of
161 basic agent technologies that can be integrated by agent systems developers to make complex systems with a high
162 degree of interoperability.

163 FIPA specifies the interfaces of the different components in the environment with which an agent can interact, i.e.
164 humans, other agents, non-agent software and the physical world.

165

166
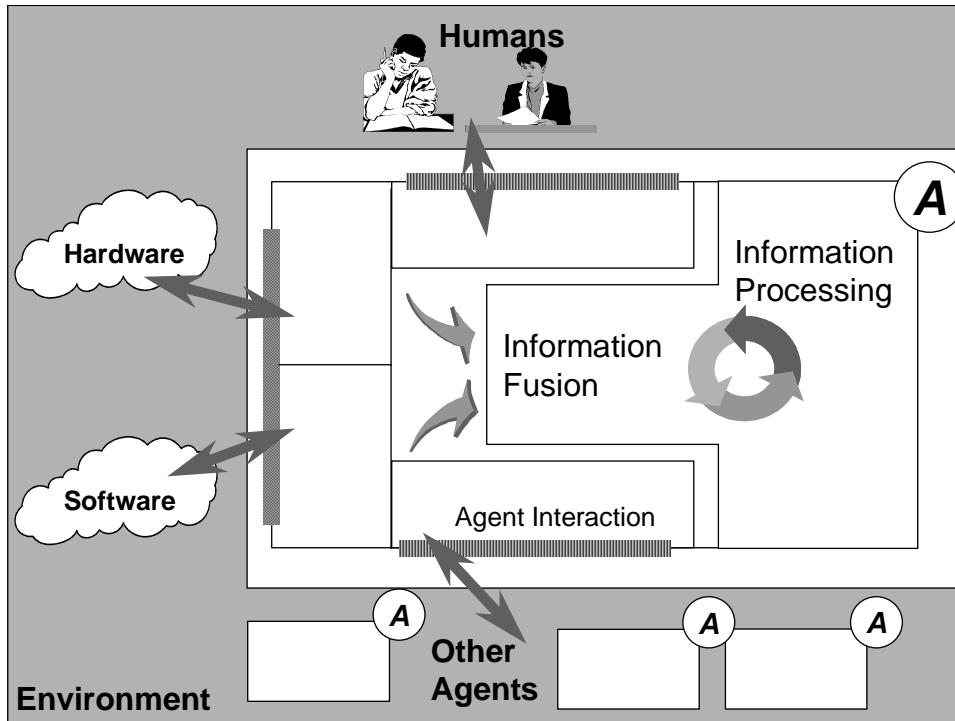
167   *Summary of agent interactions with their environment*

168

169 FIPA produces two kinds of specification:

170   **normative** specifications that mandate the external behaviour of an agent and ensure interoperability with other
171   FIPA-specified subsystems;

172   **informative** specifications of applications for guidance to industry on the use of FIPA technologies.

173 The first set of specifications – called FIPA 97 – has seven parts:

174   three normative parts for basic agent technologies: agent management, agent communication language and
175   agent/software integration

176   four informative application descriptions that provide examples of how the normative items can be applied:
177   personal travel assistance, personal assistant, audio-visual entertainment and broadcasting and network
178   management and provisioning.

179 Overall, the three FIPA 97 technologies allow:

180   the construction and management of an agent system composed of different agents, possibly built by different
181   developers;

182   agents to communicate and interact with each other to achieve individual or common goals;

183   legacy software or new non-agent software systems to be used by agents.

184

185   A brief summary of the FIPA 97 specification is given below.

Part 1 Agent Management

187   This part of FIPA 97 provides a normative framework within which FIPA compliant agents can exist, operate and be
188   managed. It defines an agent platform reference model containing such capabilities as white and yellow pages,
189   message routing and life-cycle management. True to the FIPA approach, these capablities are themselves intelligent
190   agents using formally sound communicative acts based on special message sets. An appropriate ontology and content
191   language allows agents to discover each other's capabilities.

Part 2 Agent Communication Language

193   The FIPA Agent Communication Language (ACL) is based on speech act theory: messages are actions, or
194   *communicative acts*, as they are intended to perform some action by virtue of being sent. The specification consists of
195   a set of message types and the description of their pragmatics, that is the effects on the mental attitudes of the sender
196   and receiver agents. Every communicative act is described with both a narrative form and a formal semantics based on
197   modal logic.

198   The specifications include guidance to users who are already familiar with KQML in order to facilitate migration to the
199   FIPA ACL.

200   The specification also provides the normative description of a set of high-level interaction protocols, including
201   requesting an action, contract net and several kinds of auctions.

Part 3 Agent/Software Integration

203   This part applies to any other non-agentised software with which agents need to "connect". Such software includes
204   legacy software, conventional database systems, middleware for all manners of interaction including hardware drivers.
205   Because in most significant applications, non-agentised software may dominate software agents, part 3 provides
206   important normative statements. It suggests ways by which Agents may connect to software via "wrappers" including
207   specifications of the wrapper ontology and the software dynamic registration mechanism. For this purpose, an Agent
208   Resource Broker (ARB) service is defined which allows advertisement of non-agent services in the agent domain and
209   management of their use by other agents, such as negotiation of parameters (e.g. cost and priority), authentication
210   and permission.

Part 4 - Personal Travel Assistance

212   The travel industry involves many components such as content providers, brokers, and personalization services,
213   typically from many different companies. In applying agents to this industry, various implementations from various
214   vendors must interoperate and dynamically discover each other as different services come and go. Agents operating
215   on behalf of their users can provide assistance in the pre-trip planning phase, as well as during the on-trip execution
216   phase. A system supporting these services is called a PTA (Personal Travel Agent).

217   In order to accomplish this assistance, the PTA interacts with the user and with other agents, representing the available
218   travel services. The agent system is responsible for the configuration and delivery - at the right time, cost, Quality of
219   Service, and appropriate security and privacy measures - of trip planning and guidance services. It provides examples
220   of agent technologies for both the hard requirements of travel such as airline, hotel, and car arrangements as well as

221   the soft added-value services according to personal profiles, e.g. interests in sports, theatre, or other attractions and
222   events.

Part 5 - Personal Assistant

224   One central class of intelligent agents is that of a personal assistant (PA). It is a software agent that acts semi-
225   autonomously for and on behalf of a user, modelling the interests of the user and providing services to the user or other
226   people and PAs as and when required. These services include managing a user's diary, filtering and sorting e-mail,
227   managing the user's activities, locating and delivering (multimedia) information, and planning entertainment and travel.
228   It is like a secretary, it accomplishes routine support tasks to allow the user to concentrate on the real job, it is
229   unobtrusive but ready when needed, rich in knowledge about user and work. Some of the services may be provided by
230   other agents (e.g. the PTA) or systems, the Personal Assistant acts as an interface between the user and these
231   systems.

232   In the FIPA'97 test application, a Personal Assistant offers the user a unified, intelligent interface to the management of
233   his personal meeting schedule. The PA is capable of setting up meetings with several participants, possibly involving
234   travel for some of them. In this way FIPA is opening up a road for adding interoperability and agent capabilities to the
235   already established

Part 6 - Audio/Video Entertainment & Broadcasting

237   An effective means of information filtering and retrieval, in particular for digital broadcasting networks, is of great
238   importance because the selection and/or storage of one's favourite choice from plenty of programs on offer can be very
239   impractical. The information should be provided in a customised manner, to better suit the user's personal preferences
240   and the human interaction with the system should be as simple and intuitive as possible. Key functionalities such as
241   profiling, filtering, retrieving, and interfacing can be made more effective and reliable by the use of agent technologies.

242   Overall, the application provides to the user an intelligent interface with new and improved functionalities for the
243   negotiation, filtering, and retrieval of audio-visual information. This set of functionalities can be achieved by
244   collaboration between a user agent and content/service provider agent.

Part 7 - Network management & provisioning

246   Across the world, numerous service providers emerge that combine service elements from different network providers
247   in order to provide a single service to the end customer. The ultimate goal of all parties involved is to find the best deals
248   available in terms of Quality of Service and cost. Intelligent Agent technology is promising in the sense that it will
249   facilitate automatic negotiation of appropriate deals and configuration of services at different levels.

250   Part 7 of FIPA 97 utilizes agent technology to provide dynamic Virtual Private Network (VPN) services where a user
251   wants to set up a multi-media connection with several other users.

252   The service is delivered to the end customer using co-operating and negotiating specialized agents. Three types of
253   agents are used that represent the interests of the different parties involved:

254   agents to communicate and interact with each other to achieve individual or common goals;

255   The Service Provider Agent (SPA) that represents the interests of the Service Provider.

256   The Network Provider Agent (NPA) that represents the interests of the Network Provider.

257   The service is established by the initiating user who requests the service from its PCA. The PCA negotiates in with
258   available SPAs to obtain the best deal available. The SPA will in turn negotiate with the NPAs to obtain the optimal
259   solution and to configure the service at network level. Both SPA and NPA communicate with underlying service- and
260   network management systems to configure the underlying networks for the service.

Document history
262   This document is release 1.0 of part 2 of the FIPA 97 standard. Draft versions of this document have been reviewed

263   within FIPA and by the agent community. Changes from previous draft versions are not recorded here. However, future
264   revisions will be noted in this section.

265 **1 Scope**

266 **"Language is a very difficult thing to put into words" –** *Voltaire*

267 This document forms part two of the FIPA 97 specification for interoperable agents and agent societies. In particular,
268 this document lays out underlying principles and detailed requirements for agents to be able to communicate with each
269 other using messages representing communicative acts, independently of the specific agent implementations.

270 The document lays out, in the sections below, the following:

271 A core set of communicative acts, their meaning and means of composition;

272 Common patterns of usage of these communicative acts, including standard composite messages, and standard
273 or commonly used interaction protocols;

274 A detailed semantic description of the underlying meaning of the core set of message primitives;

275 A summary of the relationship between the FIPA ACL and widely used *de facto* standard agent communication
276 language KQML.

277 *Objectives of this document*

278 This document is intended to be directly of use to designers, developers and systems architects attempting to design,
279 build and test agent applications, particularly communities of multiple agents. It aims to lay out clearly the practical
280 components of inter-agent communication and co-operation, and explain the underlying theory. Beyond a basic
281 appreciation of the model of agent communication, readers can make practical use of the ACL specification without
282 necessarily absorbing the detail of the formal basis of the language.

283 However, the language does have a well-defined formal semantic foundation. The intention of this semantics is that it
284 both gives a deeper understanding of the meaning of the language to the formally inclined, and provides an
285 unambiguous reference point. This will be of increasing importance as agents, independently developed by separate
286 individuals and teams, attempt to inter-operate successfully.

287 This part of the FIPA 97 specification defines a language and supporting tools, such as protocols, to be used by
288 *intelligent software agents* to communicate with each other. The technology of software agents imposes a high-level
289 view of such agents, deriving much of its inspiration from social interaction in other contexts, such as human-to-human
290 communication. Therefore, the terms used and the mechanisms used support such a higher-level, often *task based*,
291 view of interaction and communication. The specification does not attempt to define the low and intermediate level
292 services often associated with communication between distributed software systems, such as network protocols,
293 transport services, etc. Indeed, the existence of such services used to physically convey the byte sequences
294 comprising the inter-agent communication acts are assumed.

295 No single, universal definition of a software agent exists, nor does this specification attempt to define one. However,
296 some characteristics of agent behaviour are commonly adopted, and the communication language defined in this
297 specification sets out to support and facilitate these behaviours. Such characteristics include, but are not limited to:

298 Goal directed behaviour;

299 Autonomous determination of courses of action;

300 Interaction by negotiation and delegation;

301 Modelling of anthropomorphic mental attitudes, such as beliefs, intentions, desires, plans and commitments;

302   Flexibility in responding to situations and needs.

303 No expectation is held that any given agent will necessarily embody any or all of these characteristics. However, it is
304 the intention of this part of the specification that such behaviours are supported by the communication language and its
305 supporting framework where appropriate.

306 **Note on conformance to the underlying semantic model**

307 The semantic model described in this document is given solely as an informative reference point for agent behaviour, as there is
308 currently no agreed technology for compliance testing against the semantics of the epistemic operators used in the model. This is
309 due to the difficulty of verifying that the mental attitudes of an agent conform to the specification, without dictating the agent's internal
310 architecture or underlying implementation model. As such, the semantics cannot be considered normative until the issue of
311 compliance testing is resolved. Such tests will be the subject of further FIPA work.

312 ## 2   Normative references

313 The following normative documents contain provisions which, through reference in this text, constitute provisions of this
314 specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.
315 However, parties to agreements based on this specification are encouraged to investigate the possibility of applying the
316 most recent editions of the normative documents indicated below. For undated references, the latest edition of the
317 normative document referred to applies. Members of ISO and IEC maintain registers of currently valid specifications.

318 ISO/IEC 2022: *Information technology - Character code.*

319 *FIPA 97 specification – Part 1: Agent Management.*

320 *FIPA 97 specification – Part 3: Agent/Software Integration.*

321

321   ## 3 Terms and definitions

322   For the purposes of this specification, the following terms and definitions apply:

323   **Action**

324   A basic construct which represents some activity which an agent may perform. A special class of actions is the
325   communicative acts.

326   **ARB Agent**

327   An agent which provides the Agent Resource Broker (ARB) service. There must be at least one such an agent in each
328   Agent Platform in order to allow the sharing of non-agent services.

329   **Agent**

330   An Agent is the fundamental actor in a domain. It combines one or more service capabilities into a unified and
331   integrated execution model which can include access to external software, human users and communication facilities.

332   **Agent Communication Language (ACL)**

333   A language with precisely defined syntax, semantics and pragmatics that is the basis of communication between
334   independently designed and developed software agents. ACL is the primary subject of this part of the FIPA
335   specification.

336   **Agent Communication Channel (ACC) Router**

337   The Agent Communication Channel is an agent which uses information provided by the Agent Management System to
338   route messages between agents within the platform and to agents resident on other platforms.

339   **Agent Management System (AMS)**

340   The Agent Management System is an agent which manages the creation, deletion, suspension, resumption,
341   authentication and migration of agents on the agent platform and provides a "white pages" directory service for all
342   agents resident on an agent platform. It stores the mapping between globally unique agent names (or GUID) and local
343   transport addresses used by the platform.

344   **Agent Platform (AP)**

345   An Agent Platform provides an infrastructure in which agents can be deployed. An agent must be registered on a
346   platform in order to interact with other agents on that platform or indeed other platforms. An AP consists of three
347   capability sets ACC, AMS and default Directory Facilitator.

348   **Communicative Act (CA)**

349   A special class of actions that correspond to the basic building blocks of dialogue between agents. A communicative
350   act has a well-defined, declarative meaning independent of the content of any given act. CA's are modelled on speech
351   act theory. Pragmatically, CA's are performed by an agent sending a message to another agent, using the message
352   format described in this specification.

353   **Content**

354   That part of a communicative act which represents the domain dependent component of the communication. Note that
355   "the content of a message" does not refer to "everything within the message, including the delimiters", as it does in
356   some languages, but rather specifically to the domain specific component. In the ACL semantic model, a content
357   expression may be composed from propositions, actions or IRE's.

358 **Conversation**

359 An ongoing sequence of communicative acts exchanged between two (or more) agents relating to some ongoing topic
360 of discourse. A conversation may (perhaps implicitly) accumulate context which is used to determine the meaning of
361 later messages in the conversation.

362 **Software System**

363 A software entity which is not conformant to the FIPA Agent Management specification.

364 **CORBA:**

365 *Common Object Request Broker Architecture*, an established standard allowing object-oriented distributed systems to
366 communicate through the remote invocation of object methods.

367 **Directory Facilitator (DF)**

368 The Directory facilitator is an agent which provides a "yellow pages" directory service for the agents. It store
369 descriptions of the agents and the services they offer.

370 **Feasibility Precondition (FP)**

371 The conditions (i.e. one or more propositions) which need be true before an agent can (plan to) execute an action.

372 **Illocutionary effect**

373 See speech act theory.

374 **Knowledge Querying and Manipulation Language (KQML)**

375 A de facto (but widely used) specification of a language for inter-agent communication. In practice, several
376 implementations and variations exist.

377 **Local Agent Platform**

378 The Local Agent Platform is the AP to which an aget is attached and which represents an ultimate destination for
379 messages directed to that agent.

380 **Message**

381 An individual unit of communication between two or more agents. A message corresponds to a communicative act, in
382 the sense that a message encodes the communicative act for reliable transmission between agents. Note that
383 communicative acts can be recursively composed, so while the outermost act is directly encoded by the message,
384 taken as a whole a given message may represent multiple individual communicative acts.

385 **Message content**

386 See content.

387 **Message transport service**

388 The message transport service is an abstract service provided by the agent management platform to which the agent is
389 (currently) attached. The message transport service provides for the reliable and timely delivery of messages to their
390 destination agents, and also provides a mapping from agent logical names to physical transport addresses.

391 **Ontology**

392 An ontology gives meanings to symbols and expressions within a given domain language. In order for a message from
393 one agent to be properly understood by another, the agents must ascribe the same meaning to the constants used in

394 the message. The ontology performs the function of mapping a given constant to some well-understood meaning. For a
395 given domain, the ontology may be an explicit construct or implicitly encoded with the implementation of the agent.

396 **Ontology sharing problem**

397 The problem of ensuring that two agents who wish to converse do, in fact, share a common ontology for the domain of
398 discourse. Minimally, agents should be able to discover whether or not they share a mutual understanding of the
399 domain constants. Some research work is addressing the problem of dynamically updating agents' ontologies as the
400 need arises. This specification makes no provision for dynamically sharing or updating ontologies.

401 **Perlocutionary Effect**

402 See speech act theory.

403 **Proposition**

404 A statement which can be either true or false. A closed proposition is one which contains no variables, other than those
405 defined within the scope of a quantifier.

406 **Protocol**

407 A common pattern of conversations used to perform some generally useful task. The protocol is often used to facilitate
408 a simplification of the computational machinery needed to support a given dialogue task between two agents.
409 Throughout this document, we reserve protocol to refer to dialogue patterns between agents, and networking protocol
410 to refer to underlying transport mechanisms such as TCP/IP.

411 **Rational Effect (RE)**

412 The rational effect of an action is a representation of the effect that an agent can expect to occur as a result of the
413 action being performed. In particular, the rational effect of a communicative act is the perlocutionary effect an agent
414 can expect the CA to have on a recipient agent.

415 Note that the recipient is not bound to ensure that the expected effect comes about; indeed it may be impossible for it
416 to do so. Thus an agent may use its knowledge of the rational effect in order to plan an action, but it is not entitled to
417 believe that the rational effect necessarily holds having performed the act.

418 **Speech Act Theory**

419 A theory of communications which is used as the basis for ACL. Speech act theory is derived from the linguistic
420 analysis of human communication. It is based on the idea that with language the speaker not only makes statements,
421 but also performs actions. A speech act can be put in a stylised form that begins "I hereby request …" or "I hereby
422 declare …". In this form the verb is called the performative, since saying it makes it so. Verbs that cannot be put into
423 this form are not speech acts, for example "I hereby solve this equation" does not actually solve the equation. [Austin
424 62, Searle 69].

425 In speech act theory, communicative acts are decomposed into locutionary, illocutionary and perlocutionary acts.
426 Locutionary acts refers to the formulation of an utterance, illocutionary refers to a categorisation of the utterance from
427 the speakers perspective (e.g. question, command, query, etc), and perlocutionary refers to the other intended effects
428 on the hearer. In the case of the ACL, the perlocutionary effect refers to the updating of the agent's mental attitudes.

429 **Software Service**

430 An instantiation of a connection to a software system.

431 **TCP/IP**

432 A networking protocol used to establish connections and transmit data between hosts

433 **Wrapper Agent**

434 An agent which provides the FIPA-WRAPPER service to an agent domain on the Internet.

435 # 4 Symbols (and abbreviated terms)

436 ACC: Agent Communication Channel

437 ACL: Agent Communication Language

438 AMS: Agent Management System

439 AP: Agent Platform

440 API: Application Programming Interface

441 ARB: Agent Resource Broker

442 CA: Communicative Act

443 CORBA: Common Object Request Broker Architecture

444 DCOM: Distributed COM

445 DF: Directory Facilitator

446 FIPA: Foundation for Intelligent Physical Agents

447 FP: Feasibility Precondition

448 GUID: Global Unique Identifier

449 HAP: Home Agent Platform

450 HTTP: Hypertext Transmission Protocol

451 IDL: Interface Definition Language

452 IIOP: Internet Inter-ORB Protocol

453 OMG: Object Management Group

454 ORB: Object Request Broker

455 RE: Rational Effect

456 RMI: Remote Method Invocation, an inter-process communication method embodied in Java

457 SL: Semantic Language

458 SMTP: Simple Mail Transfer Protocol

459 TCP / IP: Transmission Control Protocol / Internet Protocol

460

460   **5   Overview of Inter-Agent Communication**

461   **5.1 Introduction**

462   This specification document does not define in a precise, prescriptive way what an agent is nor how it should be
463   implemented. Besides the lack of a general consensus on this issue in the agent research community, such definitions
464   frequently fall into the trap of being overly restrictive, ruling out some software constructs whose developers
465   legitimately consider to be agents, or else overly weak and of little assistance to the reader or software developer. A
466   goal of this specification is to be as widely applicable as possible, so the stance taken is to define the components as
467   precisely as possible, and allow applicability in any particular instance to be decided by the reader.

468   Nevertheless, some position must be taken on some of the characteristics of an agent, that it, on what an agent can
469   *do*, in order that the specification can specify a means of doing it. This position is outlined here, and consists of an
470   *abstract characterisation* of agent properties, and a simple abstract model of inter-agent communication.

471   The first characteristic assumed is that agents are communicating at a higher level of discourse, i.e. that the contents
472   of the communication are meaningful statements about the agents' environment or knowledge. This is one
473   characteristic that differentiates agent communication from, for example, other interactions between strongly
474   encapsulated computational entities such as method invocation in CORBA.

475   In order for this discourse to be given meaning, some assumptions have to be made about the agents. In this
476   specification, an abstract characterisation of agents is assumed, in which some core capabilities of agents are
477   described in terms of the agent's *mental attitudes*. This characterisation or model is intended as an abstract
478   specification, i.e. it does not pre-determine any particular agent implementation model nor a cognitive architecture.

479   More specifically, this specification characterises an agent as being able to be described as though it has mental
480   attitudes of:

481   **Belief**, which denotes the set of propositions (statements which can be true or false) which the agent accepts
482   are (currently) true; propositions which are believed false are represented by believing the negation of the
483   proposition.

484   **Uncertainty**, which denotes the set of propositions which the agent accepts are (currently) not known to be
485   certainly true or false, but which are held to be more likely to be true than false; propositions which are
486   uncertain but more likely to be false are represented by being uncertain of the negation of the proposition.
487   Note that this attitude does not prevent an agent from adopting a specific uncertain information formalism,
488   such as probability theory, in which a proposition is believed to have a certain degree of support. Rather the
489   uncertainty attitude provides a least commitment mechanism for agents with differing representation schemes
490   to discuss uncertain information.

491   **Intention**, which denotes a *choice*, or property or set of properties of the world which the agent desires to be
492   true and which are not currently believed to be true. An agent which adopts an intention will form a plan of
493   action to bring about the state of the world indicated by its choice.

494   Note that, with respect to some given proposition *p*, the attitudes of believing *p*, believing *not p*, being uncertain of *p*
495   and being uncertain of *not p* are mutually exclusive.

496   In addition, agents understand and are able to perform certain *actions*. In a distributed system, an agent typically will
497   only be able to fulfil its intentions by influencing other agents to perform actions.

498   Influencing the actions of other agents is performed by a special class of actions, denoted *communicative acts*. A
499   communicative act is performed by one agent towards another. The mechanism of performing a communicative act is
500   precisely that of sending a message encoding the act. Hence the roles of initiator and recipient of the communicative
501   act are frequently denoted as the *sender* and *receiver* of the message, respectively.

502 Building from a well-defined core, the messages defined in this specification represent a set of communicative acts that
503 attempt to seek a balance between generality, expressive power and simplicity, together with perspicuity to the agent
504 developer. The message type defines the communicative action that is being performed. Together with the appropriate
505 domain knowledge, the communicative act allows the receiver to determine the meaning of the contents of the
506 message.

507 The meanings of the communicative acts given in §6.5 are given in terms of the pre-conditions in respect to the
508 sender's mental attitudes, and the expected (from the sender's point of view) consequences on the receiver's mental
509 attitudes. However, since the sender and receiver are independent, there is no guarantee that the expected
510 consequences come to pass. For example, agent *i* may believe that "it is better to read books than to watch TV", and
511 may intend *j* to come to believe so also. Agent i will, in the ACL, *inform j* of its belief in the truth of that statement. Agent
512 *j* will then know, from the semantics of *inform*, that *i* intends it to believe in the value of books, but whether *j* comes
513 itself to believe the proposition is a matter for *j* alone to decide.

514 This specification concerns itself with inter-agent communication through message passing. Key sections of the
515 discussion are as follows:

516   §5.2 discusses the transportation of messages between agents;

517   §6.3 introduces the structure of messages;

518   §6.4 gives a standard transport syntax for transmitting ACL messages over simple byte streams;

519   §6.5 catalogues the standardised communicative acts and their representation as messages;

520   §7 introduces and defines a set of communication protocols to simplify certain common sequences of
521   messages;

522   §8 formally defines the underlying communication model.

523 ## 5.2 Message Transport Mechanisms

524 For two agents to communicate with each other by exchanging messages, they must have some common meeting
525 point through which the messages are delivered. The existence and properties of this *message transport service* are
526 the remit of FIPA Technical Committee 1: Agent Management.

527 The ACL presented here takes as a position that the contribution of agent technology to complex system behaviour
528 and inter-operation is most powerfully expressed at what, for the lack of a better term, may be called the higher levels
529 of interaction. For example, this document describes communicative acts for informing about believed truths,
530 requesting complex actions, protocols for negotiation, etc. The interaction mechanisms presented here do not compete
531 with, nor should they be compared to, low-level networking protocols such as TCP/IP, the OSI seven layer model, etc.
532 Nor do they directly present an alternative to CORBA, Java RMI or Unix RPC mechanisms. However, the functionality
533 of ACL does, in many ways overlap with the foregoing examples, not least in that ACL messages may often be
534 expected to be delivered via such mechanisms.

535 The ACL's role may be further clarified by consideration of the FIPA goal of general open agent systems. Other
536 mechanisms, notably CORBA, share this goal, but do so by imposing certain restrictions on the interfaces exposed by
537 objects. History suggests that agents and agent systems are typically implemented with a greater variety of interface
538 mechanisms; existing example agents include those using TCP/IP sockets, HTTP, SMTP and GSM short messages.
539 ACL respects this diversity by attempting to minimise requirements on the message delivery service. Notably, the
540 minimal message transport mechanism is defined as a textual form delivered over a simple byte stream, which is also
541 the approach taken by the widely used KQML agent communication language. A potential penalty for this inclusive
542 approach is upon very high-performance systems, where message throughput is pre-eminent. Future versions of this
543 specification may define alternative transport mechanism assumptions, including other transport syntaxes, which meet
544 the needs of very high performance systems.

545   Currently, the ACL imposes a minimal set of requirements on the message transport service, as shown below:

546   a)   The message service is able to deliver a message, encoded in the transport form below, to a destination as a
547   sequence of bytes. The message service exposes through its interface whether it is able to cope reliably with 8-bit
548   bytes whose high-order bit may be set.

549   b)   The normal case is that the message service is *reliable* (well-formed messages will arrive at the destination)
550   *accurate* (the message is received in the form in which it was sent), and *orderly* (messages from agent a to agent
551   b arrive at b in the order in which they were sent from a[1]). Unless informed otherwise, an agent is entitled to
552   assume that these properties hold.

553   c)   If the message delivery service is unable to guarantee any or all of the above properties, this fact is exposed in
554   some way through the interface to the message delivery service

555   d)   An agent will have the option of selecting whether it suspends and waits for the result of a message (synchronous
556   processing) or continues with other unrelated tasks while waiting for a message reply (asynchronous processing).
557   The availability of this behaviour will be implementation specific, but it must be made explicit where either
558   behaviour is not supported.

559   e)   Parameters of *the act of delivering a message*, such as time-out if no reply, are not codified at the message level
560   but are part of the interface exposed by the message delivery service.

561   f)   The message delivery service will detect and report error conditions, such as: ill-formed message, undeliverable,
562   unreachable agent, etc., back to the sending agent. Depending on the error condition, this may be returned either
563   as a return value from the message sending interface, or through the delivery of an appropriate error message.

564   g)   An agent has a name which will allow the message delivery service to deliver the message to the correct
565   destination. The message delivery service will be able to determine the correct transport mechanism (TCP/IP,
566   SMTP, http, etc.), and will allow for changes in agent location, as necessary.

567   The agent will, in some implementation specific way, have an structure which corresponds to a message it wishes to
568   send or has received. The syntax shown below in this document defines a *transport form*, in which the message is
569   mapped from its internal form to a character sequence, and can be mapped back to the internal message form from a
570   given character sequence. Note again the absence of architectural commitment: the internal message form *may* be a
571   explicit data structure, or it *may* be implicit in the way that the agent handles its messages.

572   For the purposes of the transport services, the message may be assumed to be an opaque byte stream, with the
573   exception that it is possible to extract the destination of the message.

574   At this transport level, messages are assumed to be encoded in 7-bit characters according to the ISO/IEC 2022
575   standard. This specification allows the expression of characters in extended character sets, such as Japanese. The
576   FIPA specification adopts the position that the default character mapping is US ASCII. More specifically, all ACL
577   compliant agents should assume that, when communication is commenced:

578   ISO/IEC 646 (US ASCII) is designated to G0;

579   ISO/IEC 6429 C0 is designated;

580   G0 is invoked in GL;

581   C0 is invoked in CL;

---

[1] Though possibly interspersed with messages from some other agent c.

582   SPACE in 2/0 (0x20) and

583   DELETE in 7/15 (0x7f)

584   Some transport services will be able to transport 8-bit characters safely, and, where this service is available, the agent
585   is free to make use of it. However, safe transmission of 8-bit characters is not universally assumed.

586

586   **6   FIPA ACL Messages**

587   **6.1 Preamble**

588   This section defines the individual message types that are central to the ACL specification. In particular, the form of the
589   messages and meaning of the message types are defined. The message types are a reference to the semantic acts
590   defined in this specification. These types impart a meaning to the whole message, that is, the act and the content of the
591   message, which extends any intrinsic meaning that the content itself may have.

592   For example, if *i* informs *j* that "Bonn is in Germany", the content of the message from *i* to *j* is "Bonn is in Germany",
593   and the act is the act of *informing*. "Bonn is in Germany" has a certain meaning, and is true under any reasonable
594   interpretation of the symbols "Bonn" and "Germany", but the meaning of the message includes effects on (the mental
595   attitudes of) agents *i* and *j*. The determination of this effect is essentially a private matter to both i and j, but for
596   meaningful communication to take place, some reasonable expectations of those effects must be fulfilled.

597   Clearly, the content of a message may range over an unrestricted range of domains. This specification does not
598   mandate any one formalism for representing message content. Agents themselves must arrange to be able to interpret
599   any given message content correctly. Note that this version of the specification does not address the *ontology sharing*
600   *problem*, though future versions may do so. The specification does set out to specify the meanings of the acts
601   independently of the content, that is, extending the above example, what it means to inform or be informed. In
602   particular, a set of standard communicative acts and their meanings is defined.

603   It may be noted, however, that there is a trade-off between the power and specificity of the acts. Notionally, a large
604   number of very specific act types, which convey nuances of meaning, can be considered equivalent to a smaller
605   number of more general ones, but they place different representational and implementation constraints on the agents.
606   The goals of the set of acts presented here are (i) to cover, overall, a wide range of communication situations, (ii) not to
607   overtax the design of simpler agents intended to fulfil a specific, well-defined purpose, and (iii) to minimise redundancy
608   and ambiguity, to facilitate the agent to choose which communicative act to employ. Succinctly, the goals are:
609   completeness, simplicity and conciseness.

610   The fundamental view of messages in ACL is that a message represents a *communicative act*. For purposes of
611   elegance and coherency, the treatment of communicative acts during dialogue should be consistent with the treatment
612   of other actions; a given communicative action is just one of the actions that an agent can perform. The term *message*
613   then plays two distinct roles within this document, depending on context. *Message* can be a synonym for
614   *communicative act*, or it may refer to the computational structure used by the message delivery service to convey the
615   agent's utterance to its destination.

616   The communication language presented in this specification is based on a precise formal semantics, giving an
617   unambiguous meaning to communicative actions. In practice, this formal basis is supplemented with pragmatic
618   extensions that serve to ease the practical implementation of effective inter-agent communications. For this reason, the
619   message *parameters* defined below are not defined in the formal semantics in §8, but are defined in narrative form in
620   the sections below. Similarly, conventions that agents are expected to adopt, such as protocol of message exchange,
621   are given an operational semantics in narrative form only.

622   **6.2 Requirements on agents**

623   This document introduces a set of pre-defined message types and protocols that are available for all agents to use.
624   However, it is not required for all agents to implement all of these messages. In particular, the minimal requirements on
625   FIPA ACL compliant agents are as follows:

> **Requirement 1:**
> Agents should send *not-understood* if they receive a message that they do not recognise or they are unable to
> process the content of the message. Agents must be prepared to receive and properly handle a *not-understood*
> message from other agents.

Requirement 2:
An ACL compliant agent may choose to implement any subset (including all, though this is unlikely) of the pre-defined message types and protocols. The implementation of these messages must be correct with respect to the referenced act's semantic definition.

Requirement 3:
An ACL compliant agent which uses the communicative acts whose names are defined in this specification must implement them correctly with respect to their definition.

Requirement 4:
Agents may use communicative acts with other names, not defined in this document, and are responsible for ensuring that the receiving agent will understand the meaning of the act. However, agents should not define new acts with a meaning that matches a pre-defined standard act.

Requirement 5:
An ACL compliant agent must be able to correctly generate a syntactically well formed message in the transport form that corresponds to the message it wishes to send. Symmetrically, it must be able to translate a character sequence that is well-formed in the transport syntax to the corresponding message.

626

627   6.3 Message structure

628   This section introduces the various structural elements of a message.

629   6.3.1   Overview of ACL messages


```
                                                ACL message
Begin message structure ──────►(inform
                                :sender agent1                      Message content expression
                                :receiver hpl-auction-server
Communicative act type ─────────:content
                                  (price (bid good02) 150)
                                :in-reply-to round-4
                                :reply-with bid04                   Parameter expression
                                :language sl
Message parameter ──────────────:ontology hpl-auction
                                )
```
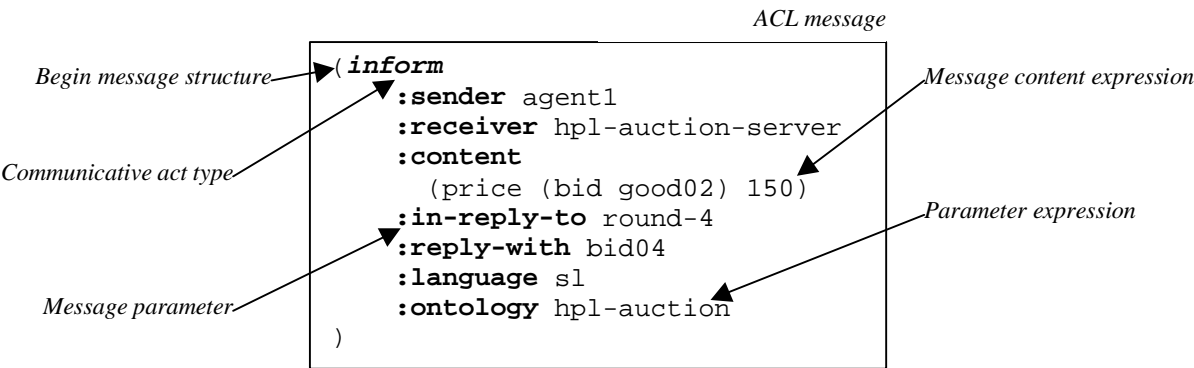
630   The following figure summarises the main structural elements of an ACL message:

631   **Figure 1 — Components of a message**

632   In their transport form, messages are represented as s-expressions. The first element of the message is a word which
633   identifies the communicative act being communicated, which defines the principal meaning of the message. There then
634   follows a sequence of message parameters, introduced by parameter keywords beginning with a colon character. No
635   space appears between the colon and the parameter keyword. One of the parameters contains the content of the
636   message, encoded as an expression in some formalism (see below). Other parameters help the message transport
637   service to deliver the message correctly (e.g. sender and receiver), help the receiver to interpret the meaning of the
638   message (e.g. language and ontology), or help the receiver to respond co-operatively (e.g. reply-with, reply-by).

639   It is this transport form that is serialised as a byte stream and transmitted by the message transport service. The
640   receiving agent is then responsible for decoding the byte stream, parsing the components message and processing it
641   correctly.

642   Note that the message's communicative act type corresponds to that which in KQML is called the *performative*[2]).

643   **6.3.2   Message parameters**

644   As noted above, the message contains a set of one or more parameters. Parameters may occur in any order in the
645   message. The only parameter that is mandatory in all messages is the *:receiver* parameter, so that the message
646   delivery service can correctly deliver the message. Clearly, no useful message will contain only the receiver. However,
647   precisely which other parameters are needed for effective communication will vary according to the situation.

648   The full set of pre-defined message parameters is shown in the following table:

649   **Table 1 — Pre-defined message parameters**

Message Parameter:	Meaning:
`:sender`	Denotes the identity of the sender of the message, i.e. the name of the agent of the communicative act.
`:receiver`	Denotes the identity of the intended recipient of the message. Note that the recipient may be a single agent name, or a tuple of agent names. This corresponds to the action of multicasting the message. Pragmatically, the semantics of this multicast is that the message is sent to each agent named in the tuple, and that the sender intends each of them to be recipient of the CA encoded in the message. For example, if an agent performs an inform act with a tuple of three agents as receiver, it denotes that the sender intends each of these agent to come to believe the content of the message.
`:content`	Denotes the content of the message; equivalently denotes the object of the action.

`:reply-with`	Introduces an *expression* which will be used by the agent responding to this message to identify the original message. Can be used to follow a conversation thread in a situation where multiple dialogues occur simultaneously.
	
	E.g. if agent i sends to agent j a message which contains
	        `:reply-with query1,`
	agent j will respond with a message containing
	        `:in-reply-to query1.`

`:in-reply-to`	Denotes an expression that references an earlier action to which this message is a reply.

---

[2] Note that the use of *performative* with respect to all of the messages defined in KQML has been challenged. The term is repeated here only because it will be familiar to many readers.

:envelope	Denotes an expression that provides useful information about the message as seen by the message transport service. The content of this parameter is not defined in the specification, but may include time sent, time received, route, etc. The structure of the envelope is a list of keyword value pairs, each of which denotes some aspect of the message service.
:language	Denotes the encoding scheme of the content of the action.
:ontology	Denotes the ontology which is used to give a meaning to the symbols in the content expression.
:reply-by	Denotes a time and/or date expression which indicates a guideline on the latest time by which the sending agent would like a reply.
:protocol	Introduces an identifier which denotes the *protocol* which the sending agent is employing. The protocol serves to give additional context for the interpretation of the message. Protocols are discussed in §7.
:conversation-id	Introduces an expression which is used to identify an ongoing sequence of communicative acts which together form a conversation. A conversation may be used by an agent to manage its communication strategies and activities. In addition the conversation may provide additional context for the interpretation of the meaning of a message.

650

651   **6.3.3   Message content**

652   The *content* of a message refers to whatever the communicative act applies to. If, in general terms, the communicative
653   act is considered as a sentence, the content is the grammatical object of the sentence. In general, the content can be
654   encoded in any language, and that language will be denoted by the :language parameter. The only requirement on
655   the content language is that it has the following properties:

> **Requirement 6:**
> In general, a content language must be able to express propositions, objects and actions. No other properties
> are required, though any given content language may be much more expressive than this. More specifically, the
> content of a message must express the data type of the action: propositions for inform, actions for request, etc.

656   A *proposition* states that some sentence in a language is true or false. An *object*, in this context, is a construct
657   which represents an identifiable "thing" (which may be abstract or concrete) in the domain of discourse.
658   Object in this context does not necessarily refer to the specialised programming constructs that appear in
659   *object-oriented* languages like C++ and Java. An *action* is a construct that the agent will interpret as being an
660   activity which can be carried out by some agent. In general, an action does not produce a result which is
661   communicated   to   another   agent   (but   see,   for   example,   §(iota <variable> <term>)
662   The iota operator introduces a scope for the given *expression* (which denotes a term), in which the given
663   *identifier*, which would otherwise be free, is defined. An expression containing a free variable is not a well-

Page 14

664 formed SL expression. The expression "(iota x (P x)" may be read as "the x such that P [is true] of x. The *iota*
665 operator is a constructor for terms which denote objects in the domain of discourse.

666 B.2.5).

667 Except in the special case outlined below, there is no requirement that message content languages conform to any well
668 known (pre-defined) syntax. In other words, it is the *responsibility of the agents in a dialogue* to ensure that they are
669 using a mutually comprehensible content language. This FIPA specification does not mandate the use of any particular
670 content language. One suggested content language formalism is shown in Annex B. There are many ways to ensure
671 the use of a common content language. It may be arranged by convention (e.g. such-and-such agents are well known
672 always to use Prolog), by negotiation[3] among the parties, or by employing the services of an intermediary as a
673 translator. Similarly, the agents are responsible for ensuring that they are using a common ontology.

674 The most general case is that of negotiating (i.e. jointly deciding) a content language. However, the agent must
675 overcome the problem of being able to begin the conversation in the first place, in order that they can then negotiate
676 content language. There has to be a common point of reference, known in advance to both parties. Thus, for the
677 specific purpose of registering with a directory facilitator and performing other key agent management functions, the
678 specification does include the following content language definition:

679 **Definition 1:**
680 The FIPA specification agent management content language is an s-expression notation used to express the
681 propositions, objects and actions pertaining to the management of the agent's lifecycle. The terms in the expression
682 are defined operationally in part one of the FIPA 97 specification.

683

> **Requirement 7:**
> A compliant agent is required to exercise the standard agent management capabilities through the use of
> messages using the agent management content language and ontology. The language and ontology are each
> denoted by the reserved term `fipa-agent-management` in their respective parameters.

684 **6.3.4   Representing the content of messages**

685 As noted above, the content of a message refers to the domain expression which the communicative act refers to. It is
686 encoded in the message as the value of the *:content* parameter. The FIPA specification does not mandate any
687 particular content encoding language (i.e. the representation form of the :content expression) on a normative basis.
688 The SL content language is provided in Annex B on an informative basis.

689 To facilitate the encoding of simple languages (that is, simple in their syntactic requirements), the s-expression form
690 included in the ACL grammar shown below allows the construction of s-expressions of arbitrary depth and complexity.
691 A language which is defined as a sub-grammar of the general s-expression grammar is therefore admissible as a legal
692 ACL message without modification. The SL grammar shown in Annex B is an example of exactly this approach.

693 However, agents commonly need to embed in the body of the message an expression encoded in a notation other than
694 the simple s-expression form used for the messages themselves. The ACL grammar provides two mechanisms, both of
695 which avoid the problem of an ACL parser being required to parse any expression in any language:

696 Wrap the expression in double quotes, thus making it a string in ACL syntax, and protect any embedded double
697 quote in the embedded expression with a backslash. Note that backslash characters in the content expression
698 must also be protected. E.g.:

---

[3] The simplest case of such negotiations is where an agent publishes its admissible content language(s) in its registration entry, and
other agents simply adopt the use of the stated language or don't talk to it.

```
699        (inform :content "owner( agent1, \"Ian\" ) "
700                :language Prolog
701          … )
```

702   Prefix the expression with the appropriate length encoded string notation, thus ensuring that the expression will be
703   treated as one lexical token irrespective of its structure. E.g.:

```
704        (inform :content #22"owner( agent1, "Ian" )
705                :language Prolog
706          … )
```

707   As a result, an ACL parser will generate one lexical token, a string, representing the entire embedded language
708   expression. Once the message has been parsed, the token representing the content expression can be interpreted
709   according to its encoding scheme, which will by then be known from the *:language* parameter.

710   **6.3.5   Use of MIME for additional content expression encoding**

711   Sometimes, even the mechanisms in the previous section are not flexible enough to represent the full range of types of
712   expression available to an agent. An example may be when an agent wishes to express a concept such as "the sound
713   you asked for is <a digitised sound>". Alternatively, it may wish to express some content in a language or character set
714   encoding different from that used for the description of the content, such as "the translation of error message <some
715   English text> into Japanese is <some Japanese text>".

716   The Multipurpose Internet Mail Extensions (MIME) standard was developed to address similar issues in the context of
717   Internet mail messages [Freed & Borenstein 96]. The syntactic form of MIME headers is suited particularly to the
718   format of mail messages, and is not congruent with the transport syntax defined for FIPA ACL messages. However, the
719   capabilities provided by MIME, and in particular the now widely used notation for annotating content types is a
720   capability of great value to some categories of agent. To allow for this, an agent may optionally be able to process
721   MIME content expression descriptions as wrappers around a given expression, using an extension of the ACL
722   message syntax.

723   It is not a mandatory part of this specification that all agents be able to process MIME content descriptions. However,
724   MIME-capable agents can register this ability with their directory facilitator, and then proceed to use the format defined
725   in Annex D.

726   Note that, for the specific task of encoding language specific character sets, the ISO 2022 standard which is the base
727   level character encoding of the message stream is capable of supporting a full range of international character
728   encodings.

729   **6.3.6   Primitive and composite communicative acts**

730   This document defines a set of predefined communicative acts, each of which is given a specific meaning in the
731   specification. Pragmatically, each of these communicative acts may be treated equivalently: they have equal status.
732   However, in terms of definition and the determination of the formal meaning of the communicative acts, we distinguish
733   two classes: *primitive acts* and *composite acts.*

734   Primitive communicative acts are those whose actions are defined atomically, i.e. they are not defined in terms of other
735   acts. Composite communicative acts are the converse. Acts are *composed* by one of the following methods:

736   making one communicative act the object of another. For example, "I *request* you to *inform* me whether it is
737   raining" is the composite *query-if* act.

738   using the *composition operator* ";" to sequence actions

739   using the composition operator "|" to denote a non-deterministic choice of actions.

740   The sequencing operator is written as an infix semicolon. Thus the expression:

741   `a ; b`

742   denotes an action, whose meaning is that of action a followed by action b.

743   The non-deterministic choice operator is written as an infix vertical bar. Thus the expression:

744   `a | b`

745   denotes a *macro action*, whose meaning is that of either action `a`, or action `b`, but not both. The action may occur in the
746   past, present or future, or not at all.

747   Note that a macro action must be treated slightly differently than other communicative acts. A macro action can be
748   planned by an agent, and requested by one agent of another. However, a macro act will not appear as the outermost
749   (i.e. top-level) message being sent from one agent to another. Macro acts are used in the definition of new composite
750   communicative acts. For example, see the *inform-if* act in §6.5.10.

751   The definition of composite actions in this way is part of the underlying semantic model for the ACL, defined using the
752   semantic description language SL. Action composition as described above is not part of the concrete syntax for ACL.
753   However, these operators are defined in the concrete syntax for SL used as a content language in Annex B.

754   **6.4 Message syntax**

755   This section defines the message transport syntax. The syntax is expressed in standard EBNF format. For
756   completeness, the notation is as follows:

Grammar rule component	Example
Terminal tokens are enclosed in double quotes	`"("`
Non terminals are written as capitalised identifiers	`Expression`
Square brackets denote an optional construct	`[ "," OptionalArg ]`
Vertical bar denotes an alternative	`Integer	Real`
Asterisk denotes zero or more repetitions of the preceding expression	`Digit *`
Plus denotes one or more repetitions of the preceding expression	`Alpha +`
Parentheses are used to group expansions.	`( A | B ) *`
Productions are written with the non-terminal name on the lhs, expansion on the rhs, and terminated by a full stop.	`ANonTerminal = "an expansion".`

757   Some slightly different rules apply for the generation of lexical tokens. Lexical tokens use the same notation as above,
758   except:

Lexical rule component	Example
Square brackets enclose a character set	["a", "b", "c"]
Dash in a character set denotes a range	["a" – "z"]
Tilde denotes the complement of a character set if it is the first character	[~ "(", ")"]
Post-fix question-mark operator denotes that the preceding lexical expression is optional (may appear zero or one times)	["0" – "9"]? ["0" – "9"]

759

760   **6.4.1   Grammar rules for ACL message syntax**

761   This section defines the grammar for ACL.

```
762    ACLCommunicativeAct = Message.
763
764    Message             = "(" MessageType MessageParameter* ")".
765
766    MessageType         = "accept-proposal"
767                        | "agree"
768                        | "cancel"
769                        | "cfp"
770                        | "confirm"
771                        | "disconfirm"
772                        | "failure"
773                        | "inform"
774                        | "inform-if"
775                        | "inform-ref"
776                        | "not-understood"
777                        | "propose"
778                        | "query-if"
779                        | "query-ref"
780                        | "refuse"
781                        | "reject-proposal"
782                        | "request"
783                        | "request-when"
784                        | "request-whenever"
785                        | "subscribe".
786
787    MessageParameter    = ":sender" AgentName
788                        | ":receiver" RecipientExpr
789                        | ":content" ( Expression | MIMEEnhancedExpression )
790                        | ":reply-with" Expression
791                        | ":reply-by" DateTimeToken
792                        | ":in-reply-to" Expression
793                        | ":envelope" KeyValuePairList
794                        | ":language" Expression
795                        | ":ontology" Expression
796                        | ":protocol" Word
797                        | ":conversation-id" Expression.
798
799    Expression          = Word
```

```
800                            | String
801                            | Number
802                            | "(" Expression * ")".
803
804     MIMEEnhancedExpression – optional extension. See Annex D.
805
806     KeyValuePairList    = "(" KeyValuePair * ")".
807
808     KeyValuePair        = "(" Word Expression ")".
809


810     RecipientExpr       = AgentName
811                         | "(" AgentName + ")".
812
813     AgentName           = Word
814                         | Word "@" AgentAddress.
815     AgentAddress        = Word "://" InternetAddress ":" Number "/" ACCObj
816     AccObj              = Word.
817
818     InternetAddress     = DNSName | IPAddress.
819     IPAddress           = Integer "." Integer "." Integer "." Integer.
820     DNSName             = Word.
```

821   **Lexical rules**

```
822     Word                    = [~ "\0x00" - "\0x1f",
823                                "(", ")", "#", "0"-"9", "-"]
824                               [~ "\0x00" - "\0x1f",
825                                "(", ")"] *.

826     String                  = StringLiteral
827                             | ByteLengthEncodedString.
828
829     StringLiteral           = "\""
830                               ( [~ "\"" ] | "\\\"" )*
831                               "\"".

832     ByteLengthEncodedString = "#" ["0" - "9"]+ "\""
833                               <byte sequence>.

834     Number                  = Integer | Float.
835
836     DateTimeToken           = "+" ?
837                               Year Month Day "T"
838                               Hour Minute Second MilliSecond
839                               (TypeDesignator ?).
840
841     Year                = Digit Digit Digit Digit.
842     Month               = Digit Digit.
843     Day                 = Digit Digit.
844     Hour                = Digit Digit.
845     Minute              = Digit Digit.
846     Second              = Digit Digit.
847     MilliSecond         = Digit Digit Digit.
848     TypeDesignator      = AlphaCharacter.
849
850     Digit                   = ["0" – "9"].
```

851

6.4.2   Notes on grammar rules

a) The standard definitions for integers and floating point numbers are assumed.

b) All keywords are case-insensitive.

c) A length encoded string is a context sensitive lexical token. Its meaning is as follows: the header of the token is everything from the leading "#" to the separator " inclusive. Between the markers of the header is a decimal number with at least one digit. This digit then determines that *exactly* that number of 8-bit bytes are to be consumed as part of the token, without restriction. It is a lexical error for less than that number of bytes to be available*.*

Note that not all implementations of the agent communication channel (ACC) [see Part One of the FIPA 97 specification] will support the transparent transmission of 8-bit characters. It is the responsibility of the agent to ensure, by reference to the API provided for the ACC, that a given channel is able to faithfully transmit the chosen message encoding.

d) A well-formed message will obey the grammar, and in addition, will have at most one of each of the parameters. It is an error to attempt to send a message which is not well formed. Further rules on well-formed messages may be stated or implied the operational definitions of the values of parameters as these are further developed.

e) Strings encoded in accordance with ISO/IEC 2022 may contain characters which are otherwise not permitted in the definition of `Word`. These characters are ESC (0x1B), SO (0x0E) and SI (0x0F). This is due to the complexity that would result from including the full ISO/IEC 2022 grammar in the above EBNF description. Hence, despite the basic description above, a word may contain any well-formed ISO/IEC 2022 encoded character, other (representations of) parentheses, spaces, or the "#" character. Note that parentheses may legitimately occur as *part* of a well formed escape sequence; the preceding restriction on characters in a word refers only to the encoded characters, not the form of the encoding.

f) Time tokens are based on the ISO 8601 format, with extensions for relative time and millisecond durations. Time expressions may be absolute, or relative to the current time. Relative times are distinguished by the character "+" appearing as the first character in the construct. If no type designator is given, the local timezone is used. The type designator for UTC is the character "Z". UTC is preferred to prevent timezone ambiguities. Note that years must be encoded in four digits. As examples, 8:30 am on April 15th 1996 local time would be encoded as:
   19960415T083000000
the same time in UTC would be:
   19960415T083000000Z
while one hour, 15 minutes and 35 milliseconds from now would be:
   +00000000T011500035.

g) The format defined for agent names is taken from part one of the FIPA 97 standard. The option of simply using a word as the agent name is only valid where that word can be unambiguously resolved to an full agent name in the format given.

6.5 Catalogue of Communicative Acts

This section defines all of the communicative acts that are part of this specification. Each message is defined by an informal narrative in this section, and more formally in §8. The narrative and formal definitions are intended to be equivalent. However, in the case of an ambiguity or inconsistency, the formal definition is the final reference point.

The following communicative acts and macro acts are standard components of the FIPA agent communication language. They are listed in alphabetical order. Communicative acts can be directly performed, can be planned by an agent, and can be requested of one agent by another. Macro acts can be planned and requested, but not directly performed.

896 **6.5.1   Preliminary notes**

897 The meanings of the communicative acts below frequently make reference to mental attitudes, such as belief, intention
898 or uncertainty. Whilst the formal semantics makes reference to formal operators which express these concepts, a given
899 agent implementation is not required to encode them explicitly, or to be founded on any particular agent model (e.g.
900 BDI). In the following narrative definitions:

901 *belief* means that, at least, the agent has a reasonable basis for stating the truth of a proposition, such as having
902 the proposition stored in a data structure or expressed implicitly in the construction of the agent software;

903 *intention* means that the agent wishes some proposition, not currently believed to be true, to become true, and
904 further that it will act in such a way that the truth of the proposition will be established. Again, this may not be
905 represented explicitly in the agent[4];

906 *uncertain* means that the agent is not sure that a proposition is necessarily true, but it is more likely to be true than
907 false. Believing a proposition and being uncertain of a proposition are mutually exclusive.

908 For ease of reference, a synopsis formal description of each act is included with the narrative text. The meaning of the
909 notation used may be found in §8.

910 **6.5.1.1   Category Index**

911 The following table identifies the communicative acts in the catalogue by category. This is provided purely for ease of
912 reference. Full descriptions of the messages can be found in the appropriate sections.

913   **Table 2 — Categories of communicative acts**

Communicative act	Information passing	Requesting information	Negotiation	Action performing	Error handling
accept-proposal			✓		
agree				✓	
cancel				✓	
cfp			✓		
confirm	✓				
disconfirm	✓				
failure					✓
inform	✓				
inform-if (macro act)	✓				

---

[4] For instance, an agent which is constructed with a simple loop which receives requests for information and always answers them immediately, can be said to be expressing an intention to be helpful; in other words to ensure that other agents who need information it possesses do indeed gain that information.

Communicative act	Information passing	Requesting information	Negotiation	Action performing	Error handling
inform-ref (macro act)	✓				
not-understood					✓
propose			✓		
query-if		✓			
query-ref		✓			
refuse				✓	
reject-proposal			✓		
request				✓	
request-when				✓	
request-whenever				✓	
subscribe		✓			

914

915

916

916 **6.5.2 accept-proposal**

<u>**Summary:**</u>	The action of accepting a previously submitted proposal to perform an action.
<u>**Message content:**</u>	A tuple, consisting of an action expression denoting the action to be done, and a proposition giving the conditions of the agreement.
<u>**Description:**</u>	*Accept-proposal* is a general-purpose acceptance of a proposal that was previously submitted (typically through a *propose* act). The agent sending the acceptance informs the receiver that it intends that (at some point in the future) the receiving agent will perform the action, once the given precondition is, or becomes, true. The proposition given as part of the acceptance indicates the preconditions that the agent is attaching to the acceptance. A typical use of this is to finalise the details of a deal in some protocol. For example, a previous offer to "hold a meeting anytime on Tuesday" might be accepted with an additional condition that the time of the meeting is 11.00. Note for future extension: i may intend that a becomes done without necessarily
<u>**Summary Formal Model**</u>	$<i,$ accept-proposal$(j, <j, a>, p(e, <j, a>))>$ $\qquad <i,$ inform$(j, I_i$ Done$(<j, a>, p(e, <j, a>)))>$ *Note: this summary is included here for completeness. For full details, see §8.*

<u>**Example**</u>	Agent i informs j that it accepts, without further preconditions, an offer from j to stream a given multimedia title to channel 19:
	
	`(accept-proposal`
	`    :sender i`
	`    :receiver j`
	`    :in-reply-to bid089`
	`    :content`
	`       (`
	`        (action j (stream-content movie1234 19))`
	`        true`
	`       )`
	`    :language sl)`
	
	Agent i informs j that it accepts an offer from j to stream a given multimedia title to channel 19 when the customer is ready. Agent i will *inform* j of this fact when appropriate:
	
	`(accept-proposal`
	`    :sender i`
	`    :receiver j`
	`    :in-reply-to bid089`
	`    :content`
	`       (`
	`        (action j (stream-content movie1234 19))`
	`        (B j (ready customer78))`
	`       )`
	`    :language sl)`

917

917 **6.5.3 agree**

<u>Summary:</u>	The action of agreeing to perform some action, possibly in the future.
<u>Message content:</u>	A tuple, consisting of an agent identifier, an action expression denoting the action to be done, and a proposition giving the conditions of the agreement.
<u>Description:</u>	*Agree* is a general purpose agreement to a previously submitted *request* to perform some action. The agent sending the agreement informs the receiver that it does intend to perform the action, but not until the given precondition is true. The proposition given as part of the *agree* act indicates the qualifiers, if any, that the agent is attaching to the agreement. This might be used, for example, to inform the receiver when the agent will execute the action which it is agreeing to perform. *Pragmatic note:* the precondition on the action being agreed to can include the perlocutionary effect of some other CA, such as an *inform* act. When the recipient of the agreement (e.g. a contract manager) wants the agreed action to be performed, it should then bring about the precondition by performing the necessary CA. This mechanism can be used to ensure that the contractor defers performing the action until the manager is ready for the action to be done.
<u>Summary Formal Model</u>	$<i$, agree(j, *a*, p)$> =$ $\qquad <i$, Inform(*j*, I$_i$ Done(*a*, p))$>$ *Note: this summary is included here for completeness. For full details, see §8.*

<u>Example</u>	Agent i (a job-shop scheduler) requests j (a robot) to deliver a box to a certain location. J answers that it agrees to the request but it has low priority.
	
	``
	(request
	    :sender i
	    :receiver j
	    :content (action j (deliver box017 (location 12 19)))
	    :protocol fipa-request
	    :reply-with order567
	)
	
	(agree
	    :sender j
	    :receiver i
	    :content ((deliver j box017 (location 12 19))
	            (priority order567 low))
	    :in-reply-to order567
	    :protocol fipa-request
	)
	``

918

918   **6.5.4   cancel**

<u>**Summary:**</u>	The action of cancelling some previously *request*ed action which has temporal extent (i.e. is not instantaneous).
<u>**Message content:**</u>	An action expression denoting the action to be cancelled.
<u>**Description:**</u>	Cancel allows an agent to stop another agent from continuing to perform (or expecting to perform) an action which was previously requested. Note that the action that is the object of the act of cancellation should be believed by the sender to be ongoing or to be planned but not yet executed. Attempting to *cancel* an action that has already been performed will result in a *refuse* message being sent back to the originator of the request.
<u>**Summary Formal Model**</u>	<*i*, cancel(*j*, a)> <*i*, disconfirm(*j*, I$_i$ Done(a)> *Note: this summary is included here for completeness. For full details, see §8.*

<u>**Example**</u>	Agent j0 asks i to cancel a previous *request-whenever* by quoting the action:
	
	<pre>(cancel
	    :sender j0
	    :receiver i
	    :content (request-whenever :sender j …)
	)</pre>
	Agent j1 asks i to cancel an action by cross-referencing the previous conversation in which the request was made:
	<pre>(cancel
	    :sender j1
	    :receiver i
	    :conversation-id cnv0087
	)</pre>

919

919   **6.5.5   cfp**

<u>**Summary:**</u>	The action of calling for proposals to perform a given action.
<u>**Message content:**</u>	A tuple containing an action expression denoting the action to be done and a proposition denoting the preconditions on the action.
<u>**Description:**</u>	*CFP* is a general-purpose action to initiate a negotiation process by making a call for proposals to perform the given action. The actual protocol under which the negotiation process is established is known either by prior agreement, or is explicitly stated in the *:protocol* parameter of the message. In normal usage, the agent responding to a *cfp* should answer with a proposition giving its conditions on the performance of the action. The responder's conditions should be compatible with the conditions originally contained in the *cfp*. For example, the *cfp* might seek proposals for a journey from Frankfurt to Munich, with a condition that the mode of travel is by train. A compatible proposal in reply would be for the 10.45 express train. An incompatible proposal would be to travel by 'plane. Note that *cfp* can also be used to simply check the availability of an agent to perform some action.

<u>**Summary Formal Model**</u>	$<i$, cfp(j, $<j$, a>, p(e, $<j$, a>))>
	       $<i$, query-ref(j, x
	             (I_j e Feasible(e, Done(e ; $<i$, inform(j, $I_i <j$, a>) >)
	                                    ((x = p'(e, $<j$, a>)) x p(e, $<j$, a>)
	
	                                       I_j Done($<j$, a>) Feasible($<j$, a>))))) >
	
	Note: this summary is included here for completeness. For full details, see §8.

<u>**Example**</u>	Agent j asks i to submit its proposal to sell 50 boxes of plums:
	
	``
	(cfp
	    :sender j
	    :receiver i
	    :content ((action i (sell plum 50)) true)
	    :ontology fruit-market
	)
	``

920

920   **6.5.6   confirm**

<u>**Summary:**</u>	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition.
<u>**Message content:**</u>	A proposition

<u>**Description:**</u>	The sending agent:
	
	        believes that some proposition is true
	
	        intends that the receiving agent also comes to believe that the proposition is true
	
	        believes that the receiver is *uncertain* of the truth of the proposition
	
	The first two properties defined above are straightforward: the sending agent is sincere[5], and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last pre-condition determines when the agent should use *confirm* vs. *inform* vs. *disconfirm*: *confirm* is used precisely when the other agent is already known to be uncertain about the proposition (rather than *uncertain* about the negation of the proposition).
	
	From the receiver's viewpoint, receiving a *confirm* message entitles it to believe that:
	
	        the sender believes the proposition that is the content of the message
	
	        the sender wishes the receiver to believe that proposition also.
	
	Whether or not the receiver does, indeed, change its mental attitude to one of belief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.

<u>**Summary Formal Model**</u>	$<i, \text{confirm}(j,)>$
	        FP: B_i B_iU_j
	        RE: B_j
	
	Note: this summary is included here for completeness. For full details, see §8.

<u>**Examples**</u>	Agent i confirms to agent j that it is, in fact, true that it is snowing today.
	
	``
	(confirm
	    :sender i
	    :receiver j
	    :content "weather(today, snowing)"
	    :language Prolog)
	``

921

---

[5] Arguably there are situations where an agent might not want to be sincere, for example to protect confidential information. We consider these cases to be beyond the current scope of this specification.

921   **6.5.7   disconfirm**

<u>**Summary:**</u>	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true.
<u>**Message content:**</u>	A proposition
<u>**Description:**</u>	The disconfirm act is used when the agent wishes to alter the known mental attitude of another agent. The sending agent: believes that some proposition is false intends that the receiving agent also comes to believe that the proposition is false believes that the receiver either believes the proposition, or is *uncertain* of the proposition. The first two properties defined above are straightforward: the sending agent is sincere *(note 5)*, and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last pre-condition determines when the agent should use *confirm*, *inform* or *disconfirm*: *disconfirm* is used precisely when the other agent is already known to believe the proposition or to be uncertain about it. From the receiver's viewpoint, receiving a *disconfirm* message entitles it to believe that: the sender believes that the proposition that is the content of the message is false; the sender wishes the receiver to believe the negated proposition also. Whether or not the receiver does, indeed, change its mental attitude to one of disbelief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.

<u>**Summary Formal Model**</u>	$<i$, disconfirm$(j,)>$
	        FP: B_i $B_i(U_j$ B_j)
	        RE: B_j
	
	Note: this summary is included here for completeness. For full details, see §8.

<u>**Examples**</u>	Agent i, believing that agent j thinks that a shark is a mammal, attempts to change j's belief:
	
	``
	(disconfirm
	    :sender i
	    :receiver j
	    :content (mammal shark))
	``

922

922   **6.5.8   failure**

<u>**Summary:**</u>	The action of telling another agent that an action was attempted but the attempt failed.
<u>**Message content:**</u>	A tuple, consisting of an action expression and a proposition giving the reason for the failure.
<u>**Description:**</u>	The failure act is an abbreviation for informing that an act was considered feasible by the sender, but was not completed for some given reason. The agent receiving a failure act is entitled to believe that: the action has not been done the action is (or, at the time the agent attempted to perform the action, was) feasible The (causal) reason for the refusal is represented by the proposition, which is the third term of the tuple. It may be the constant *true*. There is no guarantee that the reason is represented in a way that the receiving agent will understand: it could be a textual error message. Often it is the case that there is little either agent can do to further the attempt to perform the action.

<u>**Summary Formal Model**</u>	$<i,$ failure$(\ j,$ a, p $)>$
	     $<i,$ inform$(\ j, (\ e)$ Single(e) e a Done$(\ e,$ Feasible(a) I_i Done$(\ a\ )\ )$
	                                                      p $($ Done$(\ a\ )$ I_i Done$(a))$ $)>$
	
	Note: this summary is included here for completeness. For full details, see §8.

<u>**Example**</u>	Agent j informs i that it has failed to open a file:
	
	``
	(failure
	    :sender j
	    :receiver i
	    :content
	      (
	       (action j "open(\"foo.txt\")")
	       (error-message "No such file: foo.txt")
	      )
	    :language sl)
	``

923

923 **6.5.9   inform**

<u>**Summary:**</u>	The sender informs the receiver that a given proposition is true.
<u>**Message content:**</u>	A proposition

<u>**Description:**</u>	The sending agent:
	
	    holds that some proposition is true;
	
	    intends that the receiving agent also comes to believe that the proposition is true;
	
	    does not already believe that the receiver has any knowledge of the truth of the proposition.
	
	The first two properties defined above are straightforward: the sending agent is sincere, and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked). The last property is concerned with the semantic soundness of the act. If an agent knows already that some state of the world holds (that the receiver knows proposition *p*), it cannot rationally adopt an intention to bring about that state of the world (i.e. that the receiver *comes to know p* as a result of the inform act). Note that the property is not as strong as it perhaps appears. The sender *is not* required to *establish* whether the receiver knows p. It is only the case that, in the case that the sender already happens to know about the state of the receiver's beliefs, it should not adopt an intention to tell the receiver something it already knows.
	
	From the receiver's viewpoint, receiving an inform message entitles it to believe that:
	
	    the sender believes the proposition that is the content of the message
	
	    the sender wishes the receiver to believe that proposition also.
	
	Whether or not the receiver does, indeed, adopt belief in the proposition will be a function of the receiver's trust in the sincerity and reliability of the sender.

<u>**Summary Formal Model**</u>	$<i$, inform(j,)>$
	    FP: B_i $B_i(Bif_j$ $Uif_j)$
	    RE: B_j
	
	Note: this summary is included here for completeness. For full details, see §8.

<u>**Examples**</u>	Agent i informs agent j that (it is true that) it is raining today:
	
	``
	(inform
	    :sender i
	    :receiver j
	    :content "weather(today, raining)"
	    :language Prolog)
	``

924

924   **6.5.10   inform-if (macro act)**

Summary:	A macro action for the agent of the action to inform the recipient whether or not a proposition is true.
Message content:	A proposition.
Description:	The *inform-if* macro act is an abbreviation for informing whether or not a given proposition is believed. The agent which enacts an *inform-if* macro-act will actually perform a standard *inform* act. The content of the inform act will depend on the informing agent's beliefs. To *inform-if* on some closed proposition : if the agent believes the proposition, it will inform the other agent that if it believes the negation of the proposition, it informs that is false (i.e.) Under other circumstances, it may not be possible for the agent to perform this plan. For example, if it has no knowledge of , or will not permit the other party to know (that it believes) , it will send a *refuse* message.
Summary Formal Model	<*i*, inform-if(*j*, p)> <*i*, inform(*j*, p)> | <*i*, inform(*j*, p)> *Note: this summary is included here for completeness. For full details, see §8.*

Examples	Agent i requests j to inform it whether Lannion is in Normandy:
	
	``
	(request
	    :sender i
	    :receiver j
	    :content
	      (inform-if :sender j
	                  :receiver i
	                  :content "in(lannion, normandy)"
	                  :language Prolog)
	    :language sl)
	``
	
	Agent j replies that it is not:
	
	``
	(inform :sender j
	        :receiver i
	        :content "\+ in(lannion, normandy)"
	        :language Prolog)
	``

925

925   **6.5.11   inform-ref (macro act)**

Summary:	A macro action for sender to inform the receiver the object which corresponds to a definite descriptor (e.g. a name).
Message content:	An object description.
Description:	The *inform-ref* macro action allows the sender to inform the receiver some object that the sender believes corresponds to a definite descriptor, such as a name or other identifying description. *Inform-ref* is a macro action, since it corresponds to a (possibly infinite) disjunction of *inform* acts, each of which informs the receiver that "the object corresponding to *name* is *x*" for some given x. For example, an agent can plan an *inform-ref* of the current time to agent j, and then perform the act "*inform* j that the time is 10.45". The agent performing the act should believe that the object corresponding to the definite descriptor is the one that is given, and should not believe that the recipient of the act already knows this. The agent may elect to send a *refuse* message if it is unable to establish the preconditions of the act. Alternatively, it may choose to alter another agents known mental attitudes with respect to the given description by *confirm-ref* or *disconfirm-ref*.
Summary Formal Model	$<i$, inform-ref(j, x (x))> $<i$, inform(j, x (x) = r_1)> ... $<i$, inform(j, x (x) = r_k)> *Note: this summary is included here for completeness. For full details, see §8.*

Example	Agent i requests j to tell it the current Prime Minister of the United Kingdom:
	
	``
	(request
	  :sender i
	  :receiver j
	  :content
	    (inform-ref
	        :sender j
	        :receiver i
	        :content (iota ?x (UKPrimeMinister ?x))
	        :ontology world-politics
	        :language sl
	    )
	  :reply-with query0
	  :language sl)
	``
	
	Agent j replies:
	
	``
	(inform
	    :sender j
	    :receiver i
	    :content (= (iota ?x (UKPrimeMinister ?x))
	                "Tony Blair")
	    :ontology world-politics
	    :in-reply-to query0)
	``
	
	Note that a standard abbreviation for the *request* of *inform-ref* used in this example is

	the act *query-ref*.

926

926   **6.5.12   not-understood**

<u>**Summary:**</u>	The sender of the act (e.g. i) informs the receiver (e.g. j) that it perceived that j performed some action, but that i did not understand what j just did. A particular common case is that i tells j that i did not understand the message that j has just sent to i.
<u>**Message content:**</u>	A tuple consisting of an action or event (e.g. a communicative act) and an explanatory reason.
<u>**Description:**</u>	The sender received a communicative act which it did not understand. There may be several reasons for this: the agent may not have been designed to process a certain act or class of acts, or it may have been expecting a different message. For example, it may have been strictly following a pre-defined protocol, in which the possible message sequences are predetermined. The *not-understood* message indicates to that the sender of the original (i.e. misunderstood) action that nothing has been done as a result of the message.
	This act may also be used in the general case for i to inform j that it has not understood j's action.
	The second term of the content tuple is a proposition representing the reason for the failure to understand. There is no guarantee that the reason is represented in a way that the receiving agent will understand: it could be a textual error message. However, a co-operative agent will attempt to explain the misunderstanding constructively

<u>**Summary Formal Model**</u>	<*i*, not-understood(*j*, a)>
	        FP: to be completed
	        RE: to be completed
	
	Note: this summary is included here for completeness. For full details, see §8.

<u>**Examples**</u>	Agent i did not understand an query-if message because it did not recognise the ontology:
	
	``
	(not-understood
	    :sender i
	    :receiver j
	    :content ((query-if :sender j :receiver i …)
	            (unknown (ontology www)))
	    :language sl)
	``

927

927 **6.5.13 propose**

<u>Summary:</u>	The action of submitting a proposal to perform a certain action, given certain preconditions.
<u>Message content:</u>	A tuple containing an action description, representing the action that the sender is proposing to perform, and a proposition representing the preconditions on the performance of the action.
<u>Description:</u>	*Propose* is a general-purpose action to make a proposal or respond to an existing proposal during a negotiation process by proposing to perform a given action subject to certain conditions being true. The actual protocol under which the negotiation process is being conducted is known either by prior agreement, or is explicitly stated in the *:protocol* parameter of the message. The proposer (the sender of the *propose*) informs the receiver that the proposer will adopt the intention to perform the action once the given precondition is met, and the receiver notifies the proposer of the receiver's intention that the proposer performs the action. A typical use of the condition attached to the proposal is to specify the price of a bid in an auctioning or negotiation protocol.
<u>Summary Formal Model</u>	<*i*, propose(*j*, <*i*, a>, p> <*i*, inform(*j*, I*j* Done(a, p) I*i* Done(a, p))> *Note: this summary is included here for completeness. For full details, see §8.*

<u>Example</u>	Agent j informs i that it will sell 50 boxes of plums for $200:
	
	`(propose`
	`    :sender j`
	`    :receiver i`
	`    :content ((action j (sell plum 50))(cost 200))`
	`    :ontology fruit-market`
	`    :in-reply-to proposal2`
	`    :language sl`
	`    …`
	`)`

928

928 **6.5.14 query-if**

Summary:	The action of asking another agent whether or not a given proposition is true.
Message content:	A proposition.
Description:	*Query-if* is the act of asking another agent whether (it believes that) a given proposition is true. The sending agent is requesting the receiver to *inform* it of the truth of the proposition. The agent performing the *query-if* act: has no knowledge of the truth value of the proposition believes that the other agent does know the truth of the proposition.
Summary Formal Model	$<i$, query-if(j,) $<i$, request(j, $<j$, inform-if(i,)>)> *Note: this summary is included here for completeness. For full details, see §8.*

Example	Agent i asks agent j if j is registered with domain server d1:
	
	``
	(query-if
	    :sender i
	    :receiver j
	    :content
	       (registered (server d1) (agent j))
	    :reply-with r09
	    …)
	``
	
	Agent j replies that it is not:
	
	``
	(inform
	    :sender j
	    :receiver i
	    :content (not (registered (server d1) (agent j)))
	    :in-reply-to r09
	)
	``

929

929   **6.5.15   query-ref**

<u>**Summary:**</u>	The action of asking another agent for the object referred to by an expression.
<u>**Message content:**</u>	A definite descriptor

<u>**Description:**</u>	*Query-ref* is the act of asking another agent to inform the requestor of the object identified by a definite descriptor. The sending agent is requesting the receiver to perform an *inform* act, containing the object that corresponds to the definite descriptor.
	
	The agent performing the *query-ref* act:
	
	    does not know which object corresponds to the descriptor
	
	    believes that the other agent does know which object corresponds to the descriptor.

<u>**Summary Formal Model**</u>	<*i*, query-ref(*j*, x (x)) <*i*, request(*j*, <*j*, inform-ref(*i*, x (x))>)> *Note: this summary is included here for completeness. For full details, see §8.*

<u>**Example**</u>	Agent i asks agent j for its available services:
	
	`(query-ref`
	`    :sender i`
	`    :receiver j`
	`    :content`
	`      (iota ?x (available-services j ?x))`
	`    …)`
	
	j replies that it can reserve trains, planes and automobiles:
	
	`(inform`
	`    :sender j`
	`    :receiver i`
	`    :content`
	`      (= (iota ?x (available-services j ?x))`
	`        ((reserve-ticket train)`
	`         (reserve-ticket plane)`
	`         (reserve automobile))`
	`      )`
	`    …)`

930

930 **6.5.16 refuse**

Summary:	The action of refusing to perform a given action, and explaining the reason for the refusal.
Message content:	A tuple, consisting of an action expression and a proposition giving the reason for the refusal.
Description:	The refuse act is an abbreviation for denying (strictly speaking, *disconfirm*ing) that an act is possible for the agent to perform, and stating the reason why that is so. The refuse act is performed when the agent cannot meet all of the preconditions for the action to be carried out, both implicit and explicit. For example, the agent may not know something it is being asked for, or another agent requested an action for which it has insufficient privilege. The agent receiving a refuse act is entitled to believe that: the action has not been done the action is not feasible (from the point of view of the sender of the refusal) the (causal) reason for the refusal is represented by the a proposition which is the third term of the tuple, (which may be the constant true). There is no guarantee that the reason is represented in a way that the receiving agent will understand: it could be a textual error message. However, a co-operative agent will attempt to explain the refusal constructively.
Summary Formal Model	<i, refuse(j, a,)> <i, disconfirm(j, Feasible(a))> ; <i, inform(j, ((Done(a) I_i Done(a))))> *Note: this summary is included here for completeness. For full details, see §8.*

Example	Agent j refuses to i reserve a ticket for i, since i there are insufficient funds in i's account:
	
	``
	(refuse
	    :sender j
	    :receiver i
	    :content
	      (
	        (action j (reserve-ticket LHR, MUC, 27-sept-97))
	        (insufficient-funds ac12345)
	      )
	    :language sl)
	``

931

931   **6.5.17   reject-proposal**

<u>**Summary:**</u>	The action of rejecting a proposal to perform some action during a negotiation.
<u>**Message content:**</u>	A tuple consisting of an action description and a proposition which formed the original proposal being rejected, and a further proposition which denotes the reason for the rejection.
<u>**Description:**</u>	*Reject-proposal* is a general-purpose rejection to a previously submitted proposal. The agent sending the rejection informs the receiver that it has no intention that the recipient performs the given action under the given preconditions. The additional proposition represents a reason that the proposal was rejected. Since it is in general hard to relate cause to effect, the formal model below only notes that the reason proposition was believed true by the sender at the time of the rejection. Syntactically the reason on the lhs should be treated as a causal explanation for the rejection, even though this is not established by the formal semantics.
<u>**Summary Formal Model**</u>	<*i*, reject-proposal(*j*, <*j*, a>, p(e, <j, a>),)> <*i*, inform(*j*, I$_i$ (Done(< *j*, a>) p(e, <j, a>)))> *Note: this summary is included here for completeness. For full details, see §8.*

<u>**Example**</u>	Agent i informs j that it rejects an offer from j to sell
	
	`(reject-proposal`
	`    :sender i`
	`    :receiver j`
	`    :content ((action j (sell plum 50)) (price-too-high 50))`
	`    :in-reply-to proposal13`
	`)`

932

932   **6.5.18   request**

<u>**Summary:**</u>	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.
<u>**Message content:**</u>	An action description.
<u>**Description:**</u>	The sender is requesting the receiver to perform some action. The content of the message is a description of the action to be performed, in some language the receiver understands. The action can be any action the receiver is capable of performing: pick up a box, book a plane flight, change a password etc. An important use of the request act is to build composite conversations between agents, where the actions that are the object of the request act are themselves communicative acts such as *inform*.
<u>**Summary Formal Model**</u>	$<i$, request(j, a)>$ FP: FP(a) [i\j] B_i Agent(j, a) $B_i I_j$ Done(a) RE: Done(a) *Note: this summary is included here for completeness. For full details, see §8.*

<u>**Examples**</u>	Agent i requests j to open a file:
	
	``
	(request
	    :sender i
	    :receiver j
	    :content "open \"db.txt\" for input"
	    :language vb)
	``

933

933 **6.5.19 request-when**

Summary:	The sender wants the receiver to perform some action when some given proposition becomes true.
Message content:	A tuple of an action description and a proposition.
Description:	*Request-when* allows an agent to inform another agent that a certain action should be performed as soon as a given precondition, expressed as a proposition, becomes true. The agent receiving a *request-when* should either refuse to take on the commitment, or should arrange to ensure that the action will be performed when the condition becomes true. This commitment will persist until such time as it is discharged by the condition becoming true, the requesting agent *cancel*s the *request-when*, or the agent decides that it can no longer honour the commitment, in which case it should send a *refuse* message to the originator. No specific commitment is implied by the specification as to how frequently the proposition is re-evaluated, nor what the lag will be between the proposition becoming true and the action being enacted. Agents which require such specific commitments should negotiate their own agreements prior to submitting the *request-when* act.
Summary Formal Model	<i, request-when(j, <j, a>, p)> <i, inform(j, I$_i$ (e) Enables(e, B$_j$p) After(e, <j, a>))> *Note: this summary is included here for completeness. For full details, see §8.*

Examples	Agent i tells agent j to notify it as soon as an alarm occurs.
	
	``
	(request-when
	    :sender i
	    :receiver j
	    :content (
	            (inform :sender j :receiver i
	             :content "something alarming!")
	            (Done(alarm))
	            )
	    …
	)
	``

934

934   **6.5.20 request-whenever**

<u>**Summary:**</u>	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.
<u>**Message content:**</u>	A tuple of an action description and a proposition.
<u>**Description:**</u>	*Request-whenever* allows an agent to inform another agent that a certain action should be performed as soon as a given precondition, expressed as a proposition, becomes true, and that, furthermore, if the proposition should subsequently become false, the action will be repeated as soon as it once more becomes true. *Request-whenever* represents a persistent commitment to re-evaluate the given proposition and take action when its value changes. The originating agent may subsequently remove this commitment by performing the *cancel* action. No specific commitment is implied by the specification as to how frequently the proposition is re-evaluated, nor what the lag will be between the proposition becoming true and the action being enacted. Agents who require such specific commitments should negotiate their own agreements prior to submitting the *request-when* act.
<u>**Summary Formal Model**</u>	<*i*, request-whenever(*j*, <*j*, a>, p)> <*i*, inform(*j*, I$_i$ Done(a, (e) Enabled(e, B$_j$ p)))> *Note: this summary is included here for completeness. For full details, see §8.*

<u>**Examples**</u>	Agent i tells agent j to notify it whenever the price of widgets rises from less than 50 to more than 50.
	
	<pre>(request-whenever
	    :sender i
	    :receiver j
	    :content ((inform :sender j :receiver i
	                 :content (price widget))
	              (> (price widget) 50))
	    …
	)</pre>

935

935   **6.5.21   subscribe**

Summary:	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.
Message content:	A definite descriptor
Description:	The *subscribe* act is a persistent version of *query-ref*, such that the agent receiving the *subscribe* will *inform* the sender of the value of the reference, and will continue to send further *inform*s if the object denoted by the definite description changes. A subscription set up by a *subscribe* act is terminated by a *cancel* act.

Summary Formal Model	Version 1 (Philippe):
	
	$<i$, subscribe(j, x (x))>
	    <i, request-whenever(j, <j, inform-ref(i, x (x)>,
	           (y) Bj ((x (x)) = y))>
	
	Version 2 (old):
	
	$<i$, subscribe(j, x (x))>
	    <i, inform(j, I$_i$ (e) (e') (y)
	        Feasible(e ; e',
	          Done(e', Bj (x (x))=y)
	              Bj (x (x))=y)
	            Feasible(e ; e', (e1) Feasible(e1)
	              (e2) (e3)
	        (e1 = (e2 ; <j, inform(i, (x (x)) = y)> ; e3)))))>
	
	We need a final decision on this – ed.
	
	Note: this summary is included here for completeness. For full details, see §8.

Examples	Agent i wishes to be updated on the exchange rate of Francs to Dollars, and makes a subscription agreement with j (an exchange rate server):
	
	``
	(subscribe
	    :sender i
	    :receiver j:
	    :content (iota ?x (= ?x (xch-rate FFr USD)))
	)
	``

936

936 **7 Interaction Protocols**

937 Ongoing conversations between agents often fall into typical patterns. In such cases, certain message sequences are
938 expected, and, at any point in the conversation, other messages are expected to follow. These typical patterns of
939 message exchange are called *protocols*. A designer of agent systems has the choice to make the agents sufficiently
940 aware of the meanings of the messages, and the goals, beliefs and other mental attitudes the agent possesses, that
941 the agent's planning process causes such protocols to arise spontaneously from the agents' choices. This, however,
942 places a heavy burden of capability and complexity on the agent implementation, though it is not an uncommon choice
943 in the agent community at large. An alternative, and very pragmatic, view is to pre-specify the protocols, so that a
944 simpler agent implementation can nevertheless engage in meaningful conversation with other agents, simply by
945 carefully following the known protocol.

946 This section of the specification details a number of such protocols, in order to facilitate the effective inter-operation of
947 simple and complex agents. No claim is made that this is an exhaustive list of useful protocols, nor that they are
948 necessary for any given application. The protocols are given pre-defined names: the requirement for adhering to the
949 specification is:

> **Requirement 8:**
> An ACL compliant agent need not implement any of the standard protocols, nor is it restricted from
> using other protocol names. However, if one of the standard protocol names is used, the agent must
> behave consistently with the protocol specification given here.

950 Note that, by their nature, agents can engage in multiple dialogues, perhaps with different agents, simultaneously. The
951 term *conversation* is used to denote any particular instance of such a dialogue. Thus, the agent may be concurrently
952 engaged in multiple conversations, with different agents, within different protocols. The remarks in this section which
953 refer to the receipt of messages under the control of a given protocol refer only to a particular conversation.

954 **7.1 Specifying when a protocol is in operation**

955 Notionally, two agents intending to use a protocol should first negotiate whether to use a protocol, and, if so, which
956 one. However, providing the mechanism to do this would negate a key purpose of protocols, which is to simplify the
957 agent implementation. The following convention is therefore adopted: placing the name of the protocol that is being
958 used in the *:protocol* parameter of a message is equivalent to (and slightly more efficient than) prepending with an
959 *inform* that i intends that the protocol will be done (i.e., formally, I$_i$ Done(*protocol-name*)). Once the protocol is
960 finished, which may occur when one of the final states of the protocol is reached, or when the name of the protocol is
961 dropped from the `:protocol` parameter of the message, this implicit intention has been satisfied.

962 If the agent receiving a message in the context of a protocol which it cannot, or does not wish to, support, it should
963 send back a *refuse* message explaining this.

964 Example:

```
965      (request :sender i
966              :receiver j
967              :content some-act
968              :protocol fipa-request
969      )
```

970

971 **7.2 Protocol Description Notation**

972 The following notation is used to describe the standard interaction protocols in a convenient manner:

973   Boxes with double edges represent communicative actions.

974   White boxes represent actions performed by initiator.

975   Shaded boxes are performed by the other participant(s) in the protocol.

976   *Italicised text* with no box represents a comment.

977

978
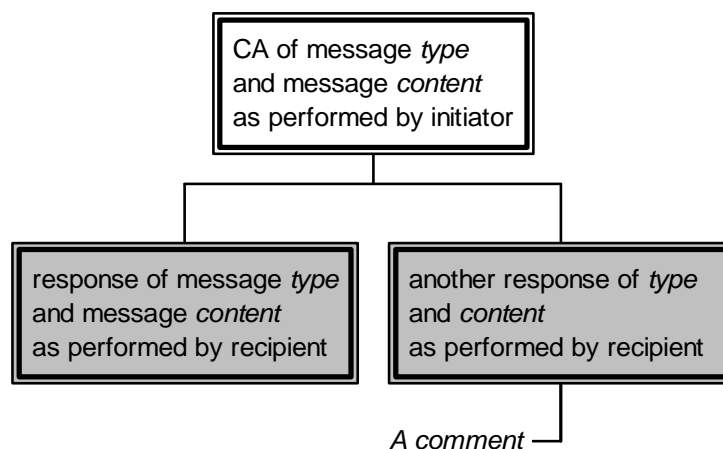
979   **Figure 2 — Example of graphical description of protocols**

980   The above notation is meant solely to represent the protocol as it might be seen by an outside observer. In particular,
981   only those actions should be depicted which are explicit objects of conversation. Actions which are internal to an agent
982   in order to execute the protocol are not represented as this may unduly restrict an agent implementation (e.g. it is of no
983   concern how an agent arrives at a proposal).

984

985   **7.3 Defined protocols**

986   **7.3.1   Failure to understand a response during a protocol**

987   Whilst not, strictly speaking, a protocol, by convention an agent which is expecting a certain set of responses in a
988   protocol, and which receives another message not in that set, should respond with a *not-understood* message.

989   To guard against the possibility of infinite message loops, it is not permissible to respond to a *not-understood* message
990   with another *not-understood* message!

991   **7.3.2   FIPA-request Protocol**

992   The FIPA-request protocol simply allows one agent to request another to perform some action, and the receiving agent
993   to perform the action or reply, in some way, that it cannot.

994
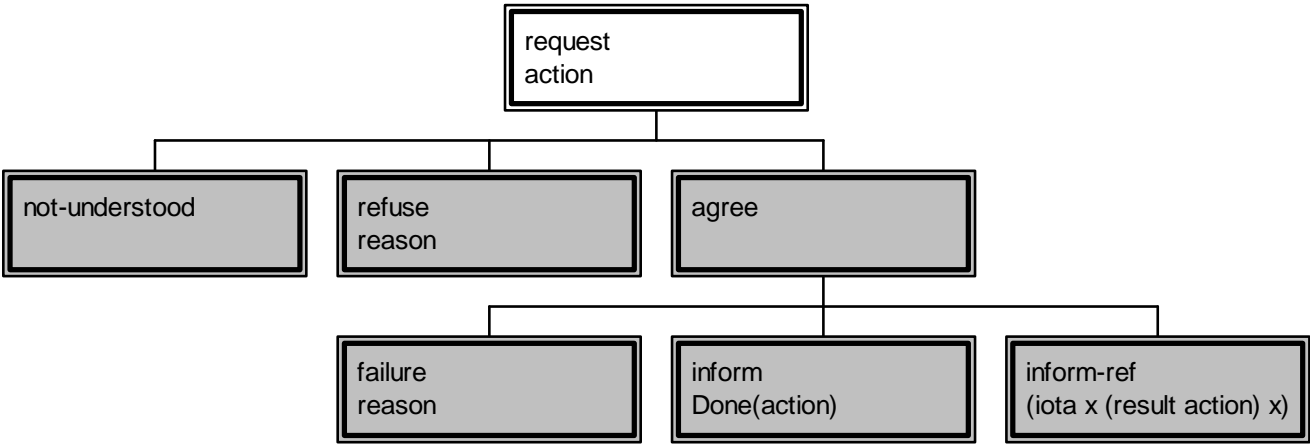
995   **Figure 3 — FIPA-Request Protocol**

996   **7.3.3   FIPA-query Protocol**

997   In the FIPA-query protocol, the receiving agent is requested to perform some kind of inform action. Requesting to
998   inform is a query, and there are two query-acts: query-if and query-ref. Either act may be used to initiate this protocol. If
999   the protocol is initiated by a query-if act, it the responder will plan to return the answer to the query with a normal inform
1000   act. If initiated by query-ref, it will instead be an inform-ref that is planned. Note that, since inform-ref is a macro act, it
1001   will in fact be an inform act that is in fact carried out by the responder.

1002

1003
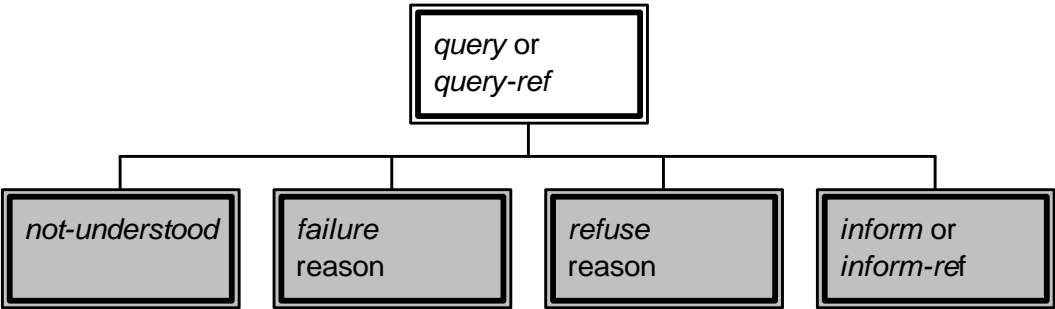
1004   **Figure 4 — FIPA-Query Protocol**

1005   **7.3.4   FIPA-request-when Protocol**

1006   The FIPA-request-when protocol is simply an expression of the full intended meaning of the request-when action. The
1007   requesting agent uses the *request-when* action to seek from the requested agent that it performs some action in the
1008   future once a given precondition becomes true. If the requested agent understands the request and does not refuse, it
1009   will wait until the precondition occurs then perform the action, after which it will notify the requester that the action has
1010   been performed. Note that this protocol is somewhat redundant in the case that the action requested involves notifying
1011   the requesting agent anyway. If it subsequently becomes impossible for the requested agent to perform the action, it
1012   will send a refuse request to the original requestor.

1013

1014
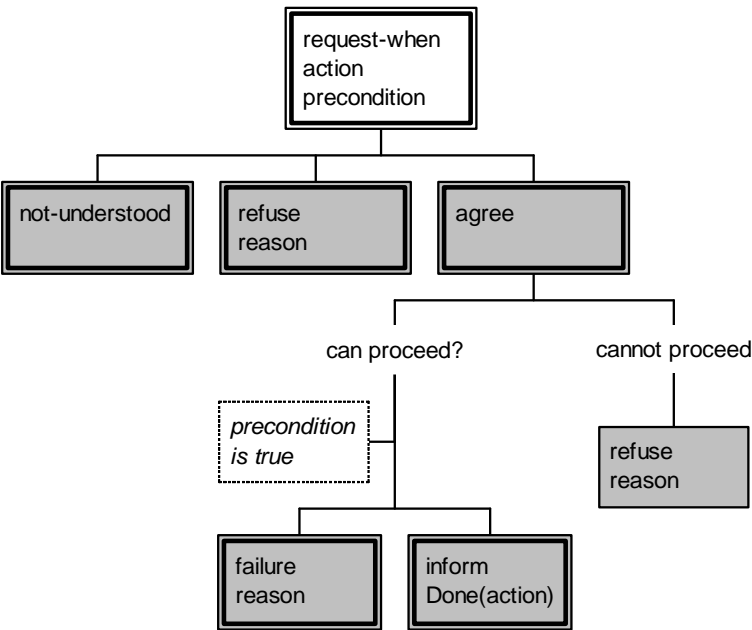
1015   **Figure 5 — FIPA-request-when protocol**

1016   **7.3.5   FIPA-contract-net Protocol**

1017   This section presents a version of the widely used *Contract Net Protocol*, originally developed by Smith and Davis
1018   [Smith & Davis 80]. FIPA-Contract-Net is a minor modification of the original contract net protocol in that it adds
1019   *rejection* and *confirmation* communicative acts. In the contract net protocol, one agent takes the role of *manager*. The
1020   manager wishes to have some task performed by one or more other agents, and further wishes to optimise a function
1021   that characterises the task. This characteristic is commonly expressed as the *price*, in some domain specific way, but
1022   could also be soonest time to completion, fair distribution of tasks, etc.

1023   The manager solicits *proposals* from other agents by issuing a *call for proposals*, which specifies the task and any
1024   conditions the manager is placing upon the execution of the task. Agents receiving the call for proposals are viewed as
1025   potential *contractors*, and are able to generate proposals to perform the task as *propose* acts. The contractor's
1026   proposal includes the preconditions that the contractor is setting out for the task, which may be the price, time when the
1027   task will be done, etc. Alternatively, the contractor may *refuse* to propose. Once the manager receives back replies
1028   from all of the contractors, it evaluates the proposals and makes its choice of which agents will perform the task. One,
1029   several, or no agents may be chosen. The agents of the selected proposal(s) will be sent an acceptance message, the
1030   others will receive a notice of rejection. The proposals are assumed to be binding on the contractor, so that once the
1031   manager accepts the proposal the contractor acquires a commitment to perform the task. Once the contractor has
1032   completed the task, it sends a completion message to the manager.

1033   Note that the protocol requires the manager to know when it has received all replies. In the case that a contractor fails
1034   to reply with either a *propose* or a *refuse*, the manager may potentially be left waiting indefinitely. To guard against this,
1035   the *cfp* includes a deadline by which replies should be received by the manager. Proposals received after the deadline
1036   are automatically rejected, with the given reason that the proposal was late.
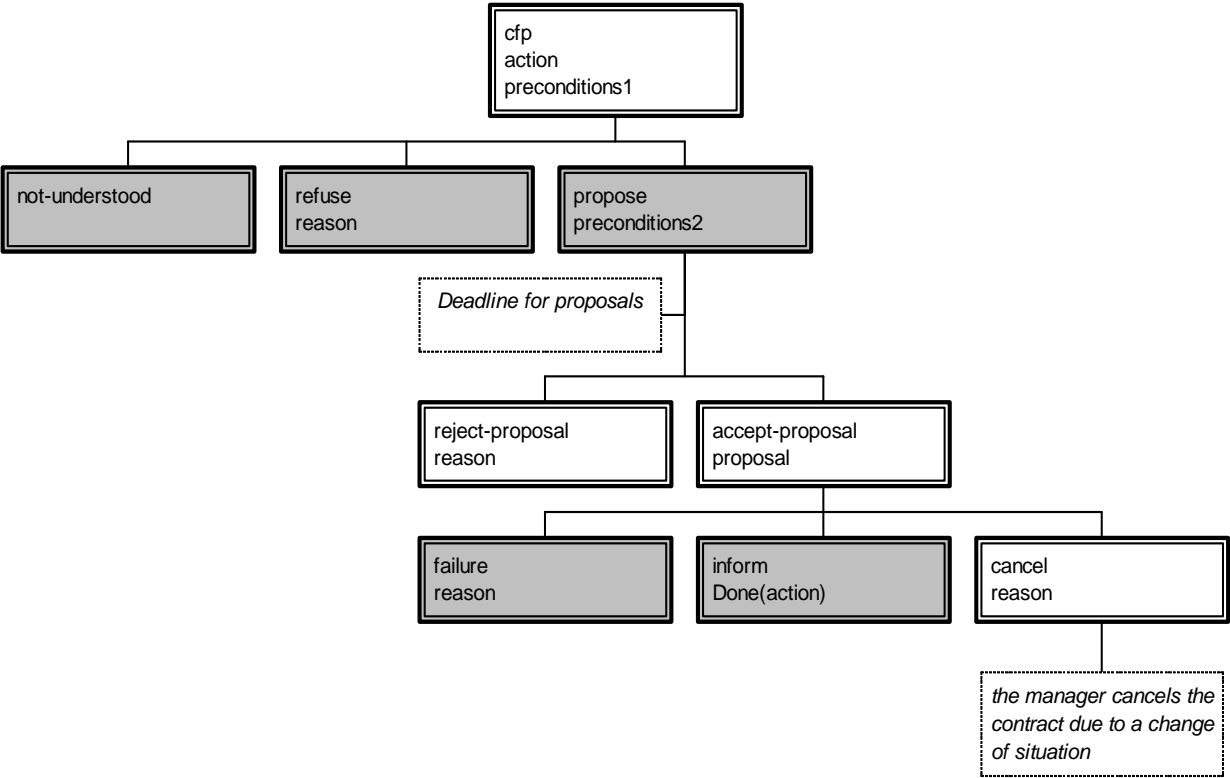
1038   **Figure 6 — FIPA-Contract-Net**

1039   **7.3.6   FIPA-Iterated-Contract-Net Protocol**

1040   The *iterated contract net* protocol is an extension of the basic contract net as described above. It differs from the basic
1041   version of the contract net by allowing multi-round iterative bidding. As above, the manager issues the initial call for
1042   proposals with the *cfp* act. The contractors then answer with their bids as *propose* acts. The manager may then accept
1043   one or more of the bids, rejecting the others, or may iterate the process by issuing a revised cfp. The intent is that the
1044   manager seeks to get better bids from the contractors by modifying the call and requesting new (equivalently, revised)
1045   bids. The process terminates when the manager refuses all proposals and does not issue a new call, accepts one or
1046   more of the bids, or the contractors all refuse to bid.

```
                            ┌──────────────────┐
                            │ cfp              │
                            │ action           │
                            │ preconditions1   │
                            └──────────────────┘
      ┌──────────────────┬──────────────────┬──────────────────┐
┌──────────────┐  ┌──────────────┐  ┌──────────────────┐
│ not-understood│  │ refuse       │  │ propose          │
│               │  │ reason       │  │ preconditions2   │
└──────────────┘  └──────────────┘  └──────────────────┘
         ┌──────────────────┬──────────────────┬──────────────────┐
   ┌──────────────────┐  ┌──────────────────┐  ┌──────────────────┐
   │ reject-proposal  │  │ accept-proposal  │  │ reject-proposal  │
   │ reason           │  │ preconditions3   │  │ reason           │
   └──────────────────┘  └──────────────────┘  └──────────────────┘
                     ┌──────────────┬──────────────┐
               ┌──────────────┐  ┌──────────────┐
               │ failure      │  │ inform       │
               │ reason       │  │ Done(action) │
               └──────────────┘  └──────────────┘
```

1047

1048   **Figure 7 — FIPA-iterated-contract-net protocol**

1049   **7.3.7   FIPA-Auction-English Protocol**

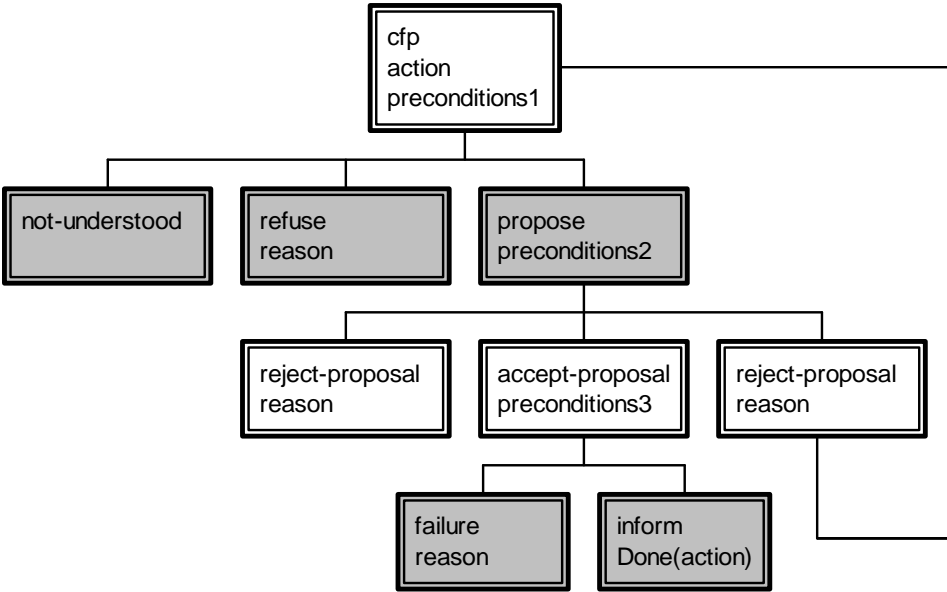1050   In the English Auction, the auctioneer seeks to find the market price of a good by initially proposing a price below that
1051   of the supposed market value, and then gradually raising the price. Each time the price is announced, the auctioneer
1052   waits to see if any buyers will signal their willingness to pay the proposed price. As soon as one buyer indicates that it
1053   will accept the price, the auctioneer issues a new call for bids with an incremented price. The auction continues until no
1054   buyers are prepared to pay the proposed price, at which point the auction ends. If the last price that was accepted by a
1055   buyer exceeds the auctioneer's (privately known) reservation price, the good is sold to that buyer for the agreed price.
1056   If the last accepted price is less than the reservation price, the good is not sold.

1057   In the following protocol diagram, the auctioneer's calls, expressed as the general *cfp* act, are multicast to all
1058   participants in the auction. For simplicity, only one instance of the message is portrayed. Note also that in a physical
1059   auction, the presence of the auction participants in one room effectively means that each acceptance of a bid is
1060   simultaneously broadcast to all participants, not just the auctioneer. This may not be true in an agent marketplace, in
1061   which case it is possible for more than one agent to attempt to bid for the suggested price. Even though the auction will
1062   continue for as long as there is at least one bidder, the agents will need to know whether their bid (represented by the
1063   *propose* act) has been accepted. Hence the appearance in the protocol of *accept-proposal* and *reject-proposal*
1064   messages, despite this being implicit in the English Auction process that is being modelled.
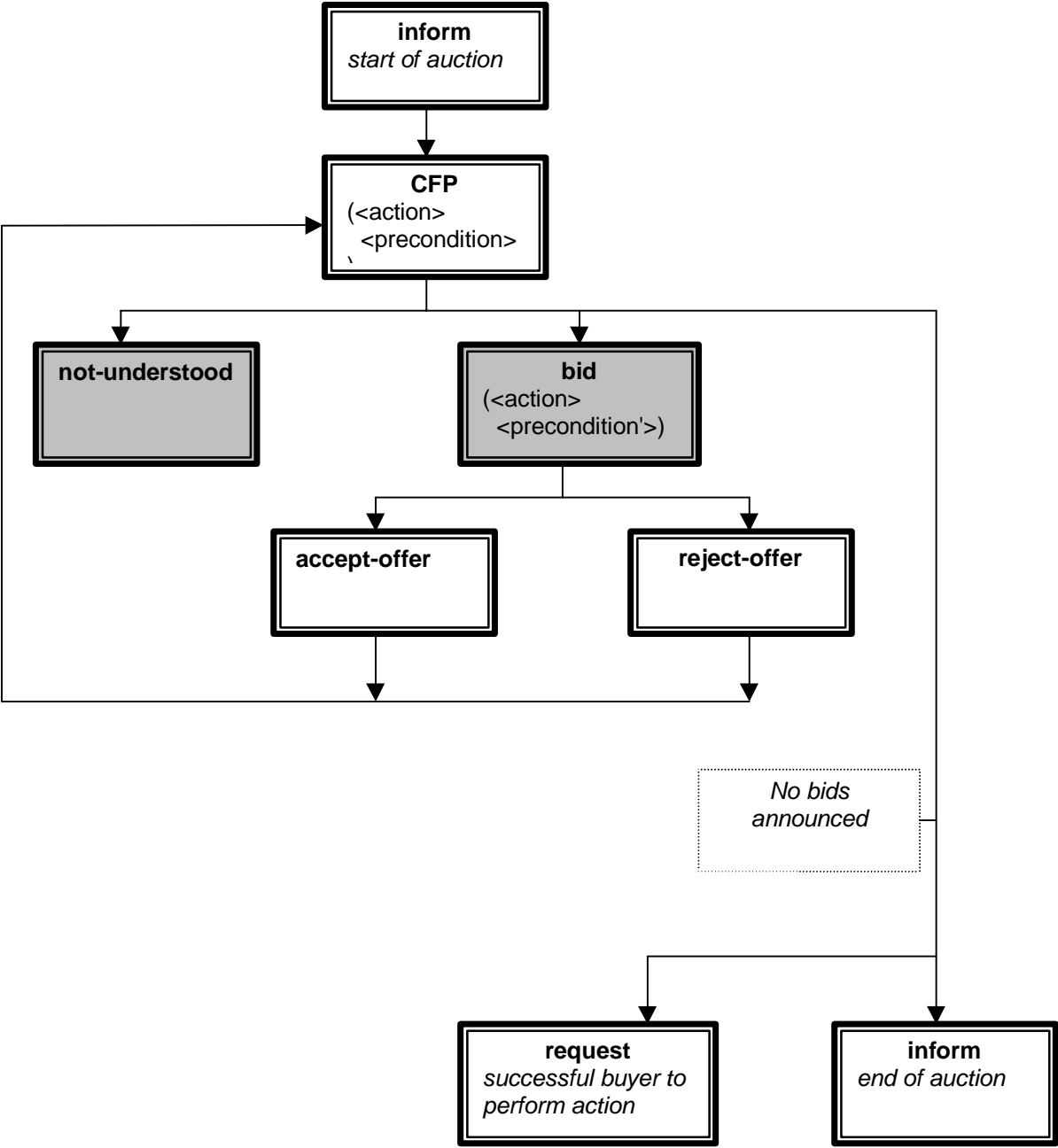
1065   **Figure 8 — FIPA-auction-english protocol**

1066   **7.3.8   FIPA-Auction-Dutch Protocol**

1067   In what is commonly called the *Dutch Auction*, the auctioneer attempts to find the market price for a good by starting
1068   bidding at a price much higher than the expected market value, then progressively reducing the price until one of the
1069   buyers accepts the price. The rate of reduction of the price is up to the auctioneer, and the auctioneer usually has a
1070   *reserve price* below which it will not go. If the auction reduces the price to the reserve price with no buyers, the auction
1071   terminates.

1072   The term "Dutch Auction" derives from the flower markets in Holland, where this is the dominant means of determining
1073   the market value of quantities of (typically) cut flowers. In modelling the actual Dutch flower auction (and indeed in
1074   some other markets), some additional complexities occur. First, the good may be split: for example the auctioneer may
1075   be selling five boxes of tulips at price *x*, and a buyer may step in and purchase only three of the boxes. The auction

1076　then continues, with a price at the next increment below *x*, until the rest of the good is sold or the reserve price met.
1077　Such partial sales of goods are only present in some markets; in others the purchaser must bid to buy the entire good.
1078　Secondly, the flower market mechanism is set up to ensure that there is no contention amongst buyers, by preventing
1079　any other bids once a single bid has been made for a good. Offers and bids are binding, so there is no protocol for
1080　accepting or rejecting a bid. In the agent case, it is not possible to assume, and too restrictive to require, that such
1081　conditions apply. Thus it is quite possible that two or more bids are received by the auctioneer for the same good. The
1082　protocol below thus allows for a bid to be rejected. This is intended only to be used in the case of multiple, competing,
1083　simultaneous bids. It is outside the scope of this specification to pre-specify any particular mechanism for resolving this
1084　conflict. In the general case, the agents should make no assumptions beyond "first come, first served". In any given
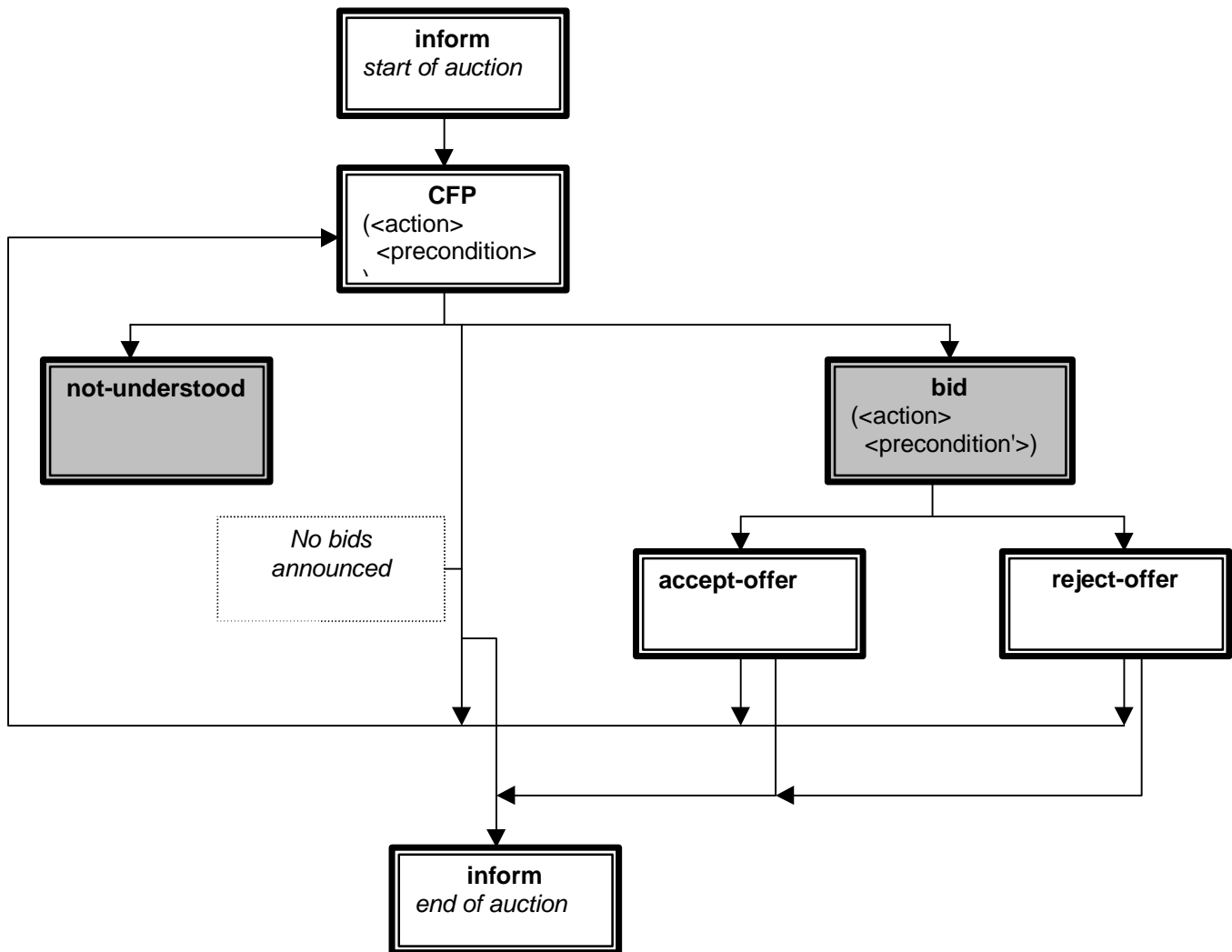1085　domain, other rules may apply.

1086　**Figure 9 — FIPA-auction-dutch protocol**

1087

1088

1088   ## 8   Formal basis of ACL semantics

1089   This section provides a formal definition of the communication language and its semantics. The intention here is to
1090   provide a clear, unambiguous reference point for the standardised meaning of the inter-agent communicative acts
1091   expressed through messages and protocols. This section of the specification is *normative*, in that agents which claim to
1092   conform to the FIPA specification ACL must behave in accordance with the definitions herein. However, this section
1093   may be treated as informative in the sense that no new information is introduced here that is not already expressed
1094   elsewhere in this document. The non mathematically-inclined reader may safely omit this section without sacrificing a
1095   full understanding of the specification.

1096   Note also that *conformance testing*, that is, demonstrating in an unambiguous way that a given agent implementation is
1097   correct with respect to this formal model, is not a problem which has been solved in this FIPA specification.
1098   Conformance testing will be the subject of further work by FIPA.

1099   ### 8.1 Introduction to formal model

1100   This section presents, in an informal way, the model of communicative acts that underlies the semantics of the
1101   message language. This model is presented *only in order to ground the stated meanings* of communicative acts and
1102   protocols. It **is not a proposed architecture** or a structural model of the agent design.

1103   Other than the special case of agents that operate singly and interact only with human users or other software
1104   interfaces, agents must communicate with each other to perform the tasks for which they are responsible.
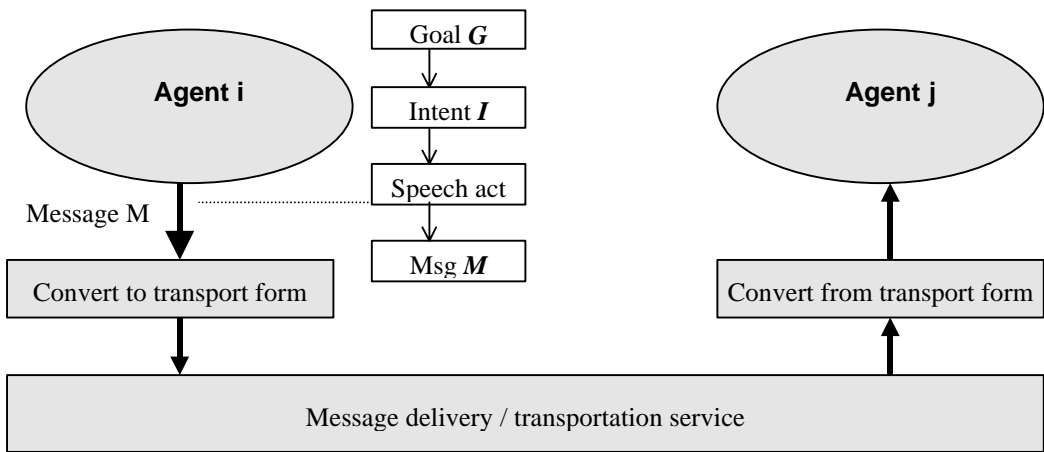
1105   Consider the basic case shown below:

1106

1107   **Figure 10 — Message passing between two agents**

1108   Suppose that, *in abstract terms*, Agent i has amongst its *mental attitudes* the following*:* some goal or objective G, and
1109   some intention I. Deciding to satisfy G, the agent adopts a specific intention, I. Note that neither of these statements
1110   entail a commitment on the design of i: G and I could equivalently be encoded as explicit terms in the mental
1111   structures of a BDI agent, or implicitly in the call stack and programming assumptions of a simple Java or database
1112   agent.

1113   Assuming that i cannot carry out the intention by itself, the question then becomes which message or set of messages
1114   should be sent to another agent (j in the figure) to assist or cause intention I to be satisfied? If agent i is behaving in
1115   some reasonable sense rationally, it will not send out a message whose effect will not satisfy the intention and hence
1116   achieve the goal. For example, if Harry wishes to have a barbecue (G = "have a barbecue"), and thus derives a
1117   goal to find out if the weather will be suitable (G' = "know if it is raining today"), and thus intends to find

1118 out the weather (I = "find out if it is raining"), he will be ill-advised to ask Sally "have you bought Acme
1119 stock today?". From Harry's perspective, whatever Sally says, it will not help him to determine whether it is raining
1120 today.

1121 Continuing the example, if Harry, acting more rationally, asks Sally "can you tell me if it is raining today?", he has acted
1122 in a way he hopes will satisfy his intention and meet his goal (assuming that Harry thinks that Sally will know the
1123 answer). Harry can reason that the effect of asking Sally is that Sally would tell him, hence making the request fulfils
1124 his intention. Now, having asked the question, can Harry actually assume that, sooner or later, he will know whether it
1125 is raining? Harry *can* assume that Sally knows that he does not know, *and* that she knows that he is asking her to tell
1126 him. But, simply on the basis of having asked, Harry cannot assume that Sally will act to tell him the weather: she is
1127 independent, and may, for example, be busy elsewhere.

1128 In summary: an agent plans, explicitly or implicitly (through the construction of its software) to meet its goals ultimately
1129 by communicating with other agents, i.e. sending messages to them and receiving messages from them. The agent will
1130 select acts based on the relevance of the act's expected outcome or *rational effect* to its goals. However, *it cannot*
1131 *assume* that the rational effect will necessarily result from sending the messages.

1132 **8.2 The SL Language**

1133 SL, standing for *Semantic Language*, is the formal language used to define the semantics of the FIPA ACL. As such,
1134 SL itself has to be precisely defined. In this section, we present the SL language definition and the semantics of the
1135 primitive communicative acts.

1136 **8.2.1   Basis of the SL formalism**

1137 In SL, logical propositions are expressed in a logic of mental attitudes and actions, formalised in a first order modal
1138 language with identity[6] (see [Sadek 91a] for details of this logic). The components of the formalism used in the
1139 following are as follows:

1140 p, p_1, ... are taken to be closed formulas denoting propositions,

1141   and   are formula schemas, which stand for any closed proposition

1142 i and j are schematic variables which denote agents

1143 | means that   is valid.

1144 The mental model of an agent is based on the representation of three primitive attitudes: *belief*, *uncertainty* and *choice*
1145 (or, to some extent, *goal*). They are respectively formalised by the modal operators *B*, *U*, and *C*. Formulas using these
1146 operators can be read as:

1147 $B_i p$ "*i* (implicitly) believes (that) *p*"

1148 $U_i p$ "*i* is uncertain about *p* but thinks that *p* is more likely than   *p*"

1149 $C_i p$ "*i* desires that *p* currently holds"

1150 The logical model for the operator *B* is a *KD45* possible-worlds-semantics Kripke structure (see, e.g., [Halpern &
1151 Moses 85]) with the fixed domain principle (see, *e.g.*, [Garson 84]).

---

[6] This logical framework is similar in many aspects to that of Cohen and Levesque (1990).

1152   To enable reasoning about action, the universe of discourse involves, in addition to individual objects and agents,
1153   sequences of events. A sequence may be formed with a single event. This event may be also the *void* event. The
1154   language involves terms (in particular a variable *e*) ranging over the set of event sequences.

1155   To talk about complex *plans*, events (or actions) can be combined to form *action expressions*:

1156 a_1 ; a_2 is a *sequence* in which a_2 follows a_1

1157 a_1 a_2 is a *nondeterministic choice*, in which either a_1 happens or a_2, but not both.

1158   Action expressions will be noted *a*.

1159   The operators *Feasible*, *Done* and *Agent* are introduced to enable reasoning about actions, as follows:

1160   *Feasible*(*a*, *p*) means that *a* can take place and if it does *p* will be true just after that

1161   *Done*(*a*, *p*) means that *a* has just taken place and *p* was true just before that

1162   *Agent*(*i*, *a*) means that *i* denotes the only agent performing, or that will be performing, the actions which
1163   appear in action expression *a*.

1164   *Single(a)* means that *a* denotes an action expression that is not a sequence. Any individual action is *Single*.
1165   The composite act a ; b is not *Single*. The composite act a | b is *Single* iff both a and b are *Single*.

1166   From belief, choice and events, the concept of *persistent goal* is defined. An agent *i* has *p* as a persistent goal, if *i* has
1167   *p* as a goal and is self-committed toward this goal until *i* comes to believe that the goal is achieved or to believe that it
1168   is unachievable. *Intention* is defined as a persistent goal imposing the agent to act. Formulas as $PG_i p$ and $I_i p$ are
1169   intended to mean that "*i* has *p* as a persistent goal" and "*i* has the intention to bring about *p*", respectively. The
1170   definition of *I* entails that *intention generates a planning process*. See [Sadek 92] for the details of a formal definition of
1171   intention.

1172   Note that there is no restriction on the possibility of embedding mental attitude or action operators. For example,
1173   formula $U_i B_j I_j Done(a, B_i p)$ informally means that agent *i* believes that, probably, agent *j* thinks that *i* has the intention
1174   that action *a* be done before which *i* has to believe *p*.

1175   A fundamental property of the proposed logic is that the modelled agents are perfectly in agreement with their own
1176   mental attitudes. Formally, the following schema is valid:

1177 B_i

1178   where   is governed by a modal operator formalising a mental attitude of agent *i*.

1179   **8.2.2   Abbreviations**

1180   In the text below, the following abbreviations are used:

1181   i)   Feasible(a)   Feasible(a, True)

1182   ii)   Done(a)   Done(a, True)

1183   iii) Possible(   )   (a)Feasible(a,   )

1184   iv) Bif_i B_i B_i
1185 Bif_i means that either agent *i* believes   or that it believes   .

1186   v) $Bref_i$ (x)   (y)B_i (x) (x) = y

1187   where is the operator for definite description and (x) (x) is read "the (x which is) ". $Bref_i$ (x) means that

1188   agent *i* believes that it knows the (x which is) .

1189   vi) Uif_i U_i U_i

1190 Uif_i means that either agent *i* is uncertain (in the sense defined above) about or that it is uncertain

1191   about .

1192   vii) $Uref_i$ (x)   (y)U_i (x) (x) = y

1193 $Uref_i$ (x) has the same meaning as $Bref_i$ (x), except that agent *i* has an uncertainty attitude with respect

1194   to (x) instead of a belief attitude

1195   viii) $AB_{n,i,j}$ $B_i B_j B_i \dots$

1196   introduces the concept of *alternate beliefs*, *n* is a positive integer representing the number of *B* operators

1197   alternating between *i* and *j*.

1198 In the text, the term "knowledge" is used as an abbreviation for "believes or is uncertain of".

1199 ## 8.3 Underlying Semantic Model

1200 The components of a communicative act (CA) model that are involved in a planning process characterise both the

1201 reasons for which the act is selected and the conditions that have to be satisfied for the act to be planned. For a given

1202 act, the former is referred to as the *rational effect* or RE[7], and the latter as the *feasibility preconditions* or *FP*'s, which

1203 are the qualifications of the act.

1204 ### 8.3.1   Property 1

1205 To give an agent the capability of planning an act whenever the agent intends to achieve its RE, the agent should

1206 adhere to the following property:

1207 Let a_k be an act such that:

1208   i)   (x) $B_i a_k$ = x,

1209   ii)   *p* is the RE of a_k and

1210   iii) C_i Possible(Done(a_k));

1211 then the following formula is valid:

1212 $I_i p$ I_i Done(a_1 ... a_n)

1213 where $a_1, ..., a_n$ are *all* the acts of type a_k.

1214 This property says that an agent's intention to achieve a given goal generates an intention that one of the acts known

1215 to the agent be done. Further, the act is such that its rational effect corresponds to the agent's goal, and that the agent

1216 has no reason for not doing it.

---

[7] Rational effect is also referred to as the *perlocutionary effect* in some of the work prior to this specification, e.g. [Sadek 90].

1217 The set of feasibility preconditions for a CA can be split into two subsets: the *ability preconditions* and the *context-*
1218 *relevance preconditions*. The ability preconditions characterise the intrinsic ability of an agent to perform a given CA.
1219 For instance, to *sincerely assert* some proposition *p*, an agent has to believe that *p*. The context-relevance
1220 preconditions characterise the relevance of the act to the context in which it is performed. For instance, an agent can
1221 be intrinsically able to make a promise while believing that the promised action is not needed by the addressee. The
1222 context-relevance preconditions correspond to the Gricean quantity and relation maxims.

1223 **8.3.2 Property 2**

1224 This property imposes on an agent an intention to seek the satisfiability of its FP's, whenever the agent elects to
1225 perform an act by virtue of property 1 [8]:

1226 $\quad I_i\ Done(a) \quad B_i\ Feasible(a) \quad I_iB_i\ Feasible(a)$

1227 **8.3.3 Property 3**

1228 If an agent has the intention that (the illocutionary component of) a communicative act be performed, it necessarily has
1229 the intention to bring about the rational effect of the act. The following property formalises this idea:

1230 $\quad I_i\ Done(a) \quad I_i\ RE(a)$

1231 where *RE*(*a*) denotes the rational effect of act *a*.

1232 **8.3.4 Property 4**

1233 Consider now the complementary aspect of CA planning: the consuming of CA's. When an agent observes a CA, it
1234 should believe that the agent performing the act has the intention (to make public its intention) to achieve the rational
1235 effect of the act. This is called the *intentional effect*. The following property captures this intuition:

1236 $\quad B_i(\ Done(a) \quad Agent(\ j, a\ ) \quad I_j\ RE(a)\ )$

1237 Note, for completeness only, that a strictly precise version of this property is as follows:

1238 $\quad B_i(\ Done(a) \quad Agent(\ j, a\ ) \quad I_j\ B_i\ I_j\ RE(a)\ )$

1239 **8.3.5 Property 5**

1240 Some FP's persist after the corresponding act has been performed. For the particular case of CA's, the next property is
1241 valid for all the FP's which do not refer to time. In such cases, when an agent observes a given CA, it is entitled to
1242 believe that the persistent feasibility preconditions hold:

1243 $\quad B_i(\ Done(a) \quad FP(a)\ )$

1244 **8.4 Notation**

1245 A communicative act model will be presented as follows:

---

[8] See [Sadek 91b] for a generalised version of this property.

1246 $<i, Act(j, C)>$
1247   FP: $_1$
1248   RE: $_2$

1249   where *i* is the agent of the act, *j* the recipient, *Act* the name of the act, *C* stands for the semantic content or
1250   propositional content[9], and $_1$ and $_2$ are propositions. This notational form is used for brevity, only within this section
1251   on the formal basis of ACL. The correspondence to the standard transport syntax adopted above is illustrated by a
1252   simple translation of the above example:

```
1253        ( Act
1254             :sender i
1255             :receiver j
1256             :content C )
```

1257   Note that this also illustrates that some aspects of the operational use of the FIPA-ACL fall outside the scope of this
1258   formal semantics but are still part of the specification. For example, the above example is actually incomplete without
1259   `:language` and `:ontology` parameters to given meaning to *C*, or some means of arranging for these to be known.

1260   **8.5 Primitive Communicative Acts**

1261   **8.5.1   The assertive Inform**

1262   One of the most interesting assertives regarding the core of mental attitudes it encapsulates is the act of *informing*. An
1263   agent *i* is able to *inform* an agent *j* that some proposition *p* is true *only* if *i* believes *p* (*i.e.*, only if $B_i p$). This act is
1264   considered to be context-relevant only if *i* does not think that *j* already believes *p* or its negation, or that *j* is uncertain
1265   about *p* (recall that belief and uncertainty are mutually exclusive). If *i* is already aware that *j* does already believe *p*,
1266   there is no need for further action by *i*. If *i* believes that *j* believes *not p*, *i* should *disconfirm p*. If *j* is uncertain about *p*, *i*
1267   should *confirm p*.

1268 $<i, INFORM (j,)>$
1269   FP: B_i $B_i(Bif_j$ Uif_j)
1270   RE: B_j

1271   The FP's for *inform* have been constructed to ensure mutual exclusiveness between CA's, when more that one CA
1272   might deliver the same rational effect.

1273   Note, for completeness only, that the above version of the *Inform* model is the operationalised version. The complete
1274   theoretical version (regarding the FP's) is the following:

1275 $<i, INFORM (j,)>$
1276   FP: B_i $\bigwedge_{n>1}$ $AB_{n,i,j}$ B_i B_iB_j $\bigwedge_{n>2}$ $AB_{n,i,j}$ B_j
1277   RE: B_j

1278   **8.5.2   The directive Request**

1279   The following model defines the directive *Request*:

---

[9] See [Searle 69] for the notions of *propositional content* (and *illocutionary force*) of an *illocutionary act*.

1280   *<i, REQUEST (j, a)>*

1281   FP: *FP(a) [\j] Bᵢ Agent(j, a) Bᵢ PGⱼ Done(a)*

1282   RE: *Done(a)*

1283 where:

1284   *a* is a schematic variable for which any action expression can be substituted;

1285   *FP*(*a*) denotes the feasibility preconditions of *a;*

1286   *FP*(*a*) [\j] denotes the part of the FP's of *a* which are mental attitudes of *i.*

8.5.3   Confirming an uncertain proposition: Confirm

1288 The rational effect of the act *Confirm* is identical to that of most of the assertives, *i.*e., the addressee comes to believe
1289 the semantic content of the act. An agent *i* is able to *confirm* a property *p* to an agent *j only* if *i* believes *p* (*i.e.*, *Bᵢp*).
1290 This is the sincerity condition an assertive act imposes on the agent performing the act. The act *Confirm* is context-
1291 relevant *only* if *i* believes that *j* is uncertain about *p* (*i.e.*, *Bᵢ Uⱼ p*). In addition, the analysis to determine the qualifications
1292 required for an agent to be entitled to perform an *Inform* act remains valid for the case of the act *Confirm.* These
1293 qualifications are identical to those of an *Inform* act for the part concerning the ability preconditions, but they are
1294 different for the part concerning the context relevance preconditions. Indeed, an act *Confirm* is irrelevant if the agent
1295 performing it believes that the addressee is not uncertain of the proposition intended to be *confirmed.*

1296 In view of this analysis, the following is the model for the act *Confirm:*

1297   <i, *CONFIRM*(j,)>

1298   FP: *Bᵢ   BᵢUⱼ*

1299   RE: *Bⱼ*

8.5.4   Contradicting knowledge: Disconfirm

1301 The *Confirm* act has a negative counterpart: the *Disconfirm* act. The characterisation of this act is similar to that of the
1302 *Confirm* act and leads to the following model:

1303   <i, *DISCONFIRM*(j,)>
1304   FP: *Bᵢ   Bᵢ₍Uⱼ   Bⱼ)*
1305   RE: *Bⱼ*

8.6 Composite Communicative Acts

1307 An important distinction is made between acts that can be carried out directly, and those macro acts which can be
1308 planned (which includes requesting another agent to perform the act), but cannot be directly carried out. The distinction
1309 centres on whether it is possible to say that an act has been done, formally *Done(Action, p)* (see §8). An act which is
1310 composed of primitive communicative actions (inform, request, confirm), or which is composed from primitive
1311 messages by substitution or sequencing (via the ";" operator), can be performed directly and can be said afterwards to
1312 be done. For example, agent *i* can inform *j* that *p*; *Done(<i, inform(j, p) >)* is then true, and the meaning (i.e. the
1313 *rational effect*) of this action can be precisely stated.

1314 However, a large class of other useful acts is defined by composition using the disjunction operator (written "|"). By the
1315 meaning of the operator, only one of the disjunctive components of the act will be performed when the act is carried
1316 out. A good example of these macro-acts is the *inform-ref* act. *Inform-ref* is a macro act defined formally by:

1317   *<i, INFORM-REF(j, x (x))>   <i, INFORM(j, x (x) = r₁)> | … | <i, INFORM(j, x (x) = rₙ)>*

1318   where n may be infinite. This act may be requested (for example, j may request i to perform it), or i may plan to perform
1319   the act in order to achieve the (rational) effect of j knowing the referent of (x). However, when the act is actually
1320   performed, what is sent, and what can be said to be *Done*, is an *inform* act.

1321   Finally an inter-agent plan is a sequence of such communicative acts, using either composition operator, involving two
1322   or more agents. Communications protocols (q.v.) are primary examples of pre-enumerated inter-agent plans.

1323   8.6.1   The closed-question case

1324   In terms of illocutionary acts, exactly what an agent *i* is *requesting* when uttering a sentence such as "Is *p*?" toward a
1325   recipient *j*, is that *j* performs the act of "*informing i that p*" or that *j* performs the act "*informing i that   p*". We know the
1326   model for both of these acts: $<j, INFORM (i,)>$. In addition, we know the relation "*or*" set between these two acts: it is
1327   the relation that allows for the building of action expressions which represent a *non-deterministic choice* between
1328   several (sequences of) events or actions.

1329   In fact, as mentioned above, the semantic content of a directive refers to an *action expression*; so, this can be a
1330   *disjunction* between two or more acts. Hence, by using the utterance "Is *p*?", what an agent *i requests* an agent *j* to do
1331   is the following action expression:

1332 $<j, INFORM (i, p)>$ $<j, INFORM (i, p)>$

1333   It seems clear that the semantic content of a directive realised by a yes/no-question can be viewed as an action
1334   expression characterising an indefinite choice between two CA's *Inform*. In fact, it can also be shown that the binary
1335   character of this relation is only a special case: in general, any number of CA's *Inform* can be handled. In this case, the
1336   addressee of a directive is allowed to choose one among several acts. This is not only a theoretical generalisation: it
1337   accounts for classical linguistic behaviour traditionally called *Alternatives question*. An example of an utterance
1338   realising an alternative question is "Would you like to travel in first class, in business class, or in economy class?". In
1339   this case, the semantic content of the *request* realised by this utterance is the following action expression:

1340 $<j, INFORM (i, p_1)>$ $<j, INFORM (i, p_2)>$ $<j, INFORM (i, p_3)>$

1341   where p_1, p_2 and p_3 are intended to mean respectively that *j* wants to travel in first class, in business class, or in
1342   economy class.

1343   As it stands, the agent designer has to provide the plan-oriented model for this type of action expression. In fact, it
1344   would be interesting to have a model which is not specific to the action expressions characterising the non-
1345   deterministic choice between CA's of type *Inform*, but a more general model where the actions referred to in the
1346   disjunctive relation remain unspecified. In other words, to describe the preconditions and effects of the expression
1347 a_1 a_2 … a_n where a_1, a_2, …, a_n are any action expressions. It is worth mentioning that the goal is to characterise this
1348   action expression as a *disjunctive macro-act* which is planned as such; we are not attempting to characterise the non-
1349   deterministic choice between acts which are planned separately. In both cases, the result is a branching plan but in the
1350   first case, the plan is branching in an *a priori* way while in the second case it is branching in an *a posteriori* way.

1351   An agent will plan a macro-act of non-deterministic choice when it intends to achieve the rational effect of one of the
1352   acts composing the choice, *no matter which one it is*. To do that, one of the feasibility preconditions of the acts must be
1353   satisfied, *no matter which one it is*. This produces the following model for a disjunctive macro-act:

1354 a_1 a_2 … a_n
1355   FP: $FP(a_1)$ $FP(a_2)$ ... $FP(a_n)$
1356   RE: $RE(a_1)$ $RE(a_2)$ ... $RE(a_n)$

1357   where $FP(a_k)$ and $RE(a_k)$ represent the FP's and the RE of the action expression a_k, respectively.

1358   Because the yes/no-question, as shown, is a particular case of alternatives question, the above model can be
1359   specialised to the case of two acts *Inform* having opposite semantic contents. Thus, we get the following model:

1360 $<i, INFORM(j,)> <i, INFORM(j,)>$

1361   FP: Bif_i $B_i(Bif_j$ $Uif_j)$

1362   RE: Bif_j

1363   In the same way, we can derive the disjunctive macro-act model which gathers the acts *Confirm* and *Disconfirm*. We
1364   will use the abbreviation $<i, CONFDISCONF(j,)>$ to refer to the following model:

1365 $<i, CONFIRM(j,)> <i, DISCONFIRM(j, >$

1366   FP: Bif_i B_iU_j

1367   RE: Bif_j

1368   8.6.2   The query-if act:

1369   Starting from the act models $<j, INFORM-IF(i,)>$ and $<i, REQUEST(j, a)>$, it is possible to derive the query-if act
1370   model (and not plan, as shown below). Unlike a confirm/disconfirm-question, which will be addressed below, an query-
1371   if act requires the agent performing it not to have any knowledge about the proposition whose truth value is asked for.
1372   To get this model, a transformation[10] has to be applied to the FP's of the act <j, INFORM-IF(i,)> and leads to the
1373   following model for a query-if act:

1374 $<i, QUERY-IF(j,   <i, REQUEST(j, <j, INFORM-IF(i,)>)>$
1375   FP: Bif_i Uif_i B_i $PG_j Done(<j, INFORM-IF (i,)>)$
1376   RE: $Done(<j, INFORM(i,)> <j, INFORM(i,)>)$

1377   8.6.3   The confirm/disconfirm-question act:

1378   In the same way, it is possible to derive the following *Confirm/Disconfirm-question* act model:

1379 $<i, REQUEST(j, <j, CONFDISCONF(i,)>)>$
1380   FP: U_i B_i $PG_j Done(<j, CONFDISCONF(i,)>)$
1381   RE: $Done(<j, CONFIRM(i,)> <j, DISCONFIRM(i,))$

1382   8.6.4   The open-question case:

1383   *Open question* is a question which does not suggest a choice and, in particular, which does not require a yes/no
1384   answer. A particular case of open questions are the questions which require referring expressions as an answer. They
1385   are generally called *wh-questions*. The "wh" refers to interrogative pronouns such as "what", "who", "where", or "when".
1386   Nevertheless, this must not be taken literally since the utterance "How did you travel?" can be considered as a wh-
1387   question.

1388   A formal plan-oriented model for the wh-questions is required. In the model below, *from the addressee's viewpoint*, this
1389   type of question can be viewed as a closed question where the suggested choice is not made explicit because it is *too
1390   wide*. Indeed, a question such as "What is your destination?" can be restated as "What is your destination: Paris,
1391   Rome,... ?".

1392   The problem is that, in general, the set of definite descriptions among which the addressee can (and must) choose is
1393   potentially an infinite set, not because, referring to the example above, there may be an infinite number of destinations,
1394   but because, theoretically, each destination can be referred to in potentially an infinite number of ways. For instance,
1395   Paris can be referred to as "the capital of France", "the city where the Eiffel Tower is located", "the capital of the country

---

[10] For more details about this transformation, called the *double-mirror transformation*, see [Sadek 91a, 91b].

1396 where the Man-Rights Chart was founded", *etc*. However, it must be noted that in the context of man-machine
1397 communication, the language used is finite and hence the number of descriptions acceptable as an answer to a *wh-*
1398 *question* is also finite.

1399 When asking a *wh-question*, an agent *j* intends to acquire from the addressee *i* an identifying referring expression (IRE)
1400 [Sadek 90] for a definite description, in the general case. Therefore, agent *j* intends to make his interlocutor *i* perform a
1401 CA which is of the following form:

1402 $<i, INFORM(j, \iota x (x) = r)>$

1403 where *r* is an IRE (*e.g.*, a standard name or a definite description) and $\iota x (x)$ is a definite description. Thus, the
1404 semantic content of the directive performed by a wh-question is a disjunctive macro-act composed with acts of the form
1405 of the act above. Here is the model of such a macro-act:

1406 $<i, INFORM(j, \iota x (x) = r_1)> \ ... \ <i, INFORM(j, \iota x (x) = r_k)>$

1407 where r_k are IREs. To deal with the case of closed questions, the generic plan-oriented model proposed for a
1408 disjunctive macro-act can be instantiated for the account of the macro-act above. Note that the following equivalence is
1409 valid:

1410 $(B_i \ \iota x (x) = r_1 \quad B_i \ \iota x (x) = r_2 \quad ...) \quad (y) B_i \ \iota x (x) = y$

1411 This produces the following model, which is referred to as $<i, INFORM\text{-}REF(j, \iota x (x))>$:

1412 $<j, INFORM\text{-}REF(i, \iota x (x))>$
1413 FP: $Bref_i (x) \quad B_i \ ref_j (x)$
1414 RE: $Bref_j (x)$

1415 where $Bref_j (x)$ and $Uref_j (x)$ are abbreviations introduced above, and $ref_j (x)$ is an abbreviation defined as:

1416 $ref_j (x) \quad Bref_j (x) \quad Uref_j (x)$

1417 Provided the act models $<j, INFORM\text{-}REF (i, \iota x (x))>$ and $<i, REQUEST (j, a)>$, *the wh-question* act model can be built
1418 up in the same way as for the yn-question act model. Applying the same transformation to the FP's of the act schema
1419 $<j, INFORM\text{-}REF (i, \iota x (x))>$, and by virtue of property 3, the following model is derived:

1420 $<i, REQUEST(j, <j, INFORM\text{-}REF(i, \iota x (x)>)>$
1421 FP: $ref_i (x) \quad B_i \quad PG_j Done(<j, INFORM\text{-}REF(i, \iota x (x))>)$
1422 RE: $Done(<i, INFORM (j, \iota x (x) = r_1)> \ ... \ <i, INFORM(j, \iota x (x) = r_k)>)$

1423 **8.6.5   Summary definitions for all standard communicative acts**

1424 **8.6.5.1   Supporting definitions**

1425 *Enables(e, p)*
1426 $Done(e, \ p) \quad p$

1427 *After(e_1, e_2)*
1428 $Done(e_1)$
1429 $(e') Feasible(e', (f) (f = e_1 ; e_2 ; e') \quad (f = e' ; e_2))$

1430   *Before(e₁, e₂)*
1431   *After(e₂, e₁)*

1432   *Will-occur-when(x, p(e, x))*
1433   *(e') Done(e' ; x, Feasible(e', p(e', x)))*

1434   *Enabled(e, p)*
1435   *Done(e, p) p*

1436   **8.6.5.2   Agree**

1437   To be completed.

1438   **8.6.5.3   Accept-proposal**

1439   <*i*, accept-proposal(*j*, <*j*, a>, p(e, <*j*, a>))>
1440   <*i*, inform(*j*, I*ᵢ* Will-occur-when(<*j*, a>, p(e, <*j*, a>)))>

1441   i informs j that i has the intention that j will perform action a just as soon as the precondition parameterised by the
1442   action and some event in the future becomes true.

1443   **8.6.5.4   Propose**

1444   <*i*, propose(*j*, <*i*, a>, p(e, <*i*, a>)>
1445   <i, inform(j,   e Feasible(e, Done(<*j*, inform(*i*, I*ⱼ* Done(<*i*, a>)
1446   p(e, <*i*, a>)   I*ᵢ* Done(<*i*, a>)
1447   Feasible(<*i*, a>))) >

1448   i informs j that, once j informs i that j has adopted the intention for i to perform action a, and the preconditions for i
1449   performing a have been established, it will be feasible for i to perform a and i will adopt the intention to perform a.

1450   **8.6.5.5   Cancel**

1451   <*i*, cancel(*j*, a)>
1452   <*i*, disconfirm(*j*, I*ᵢ* Done(a)>

1453   *Cancel* is the action of cancelling any form of *request*ed action. In other words, an agent *i* has requested an agent *j* to
1454   perform some action, possibly if some condition holds. This has the effect of *i* informing *j* that *i* has an intention. When *i*
1455   comes to drop its intention, it has to inform *j* that it no longer has this intention, i.e. a *disconfirm*.

1456

1457   **8.6.5.6   cfp**

1458   <*i*, cfp(*j*, <*j*, a>, p(e, <*j*, a>))>
1459   <*i*, query-ref(*j*, x
1460   (I*ⱼ*   e Feasible(e, Done(e ; <*i*, inform(*j*, I*ᵢ* <*j*, a>) >)
1461   ((x = p'(e, <*j*, a>))   p(e, <*j*, a>)
1462
1463   I*ⱼ* Done(<*j*, a>)   Feasible(<*j*, a>))))) >

1464   i requests j to inform i of the additional preconditions (i.e. predicate p') j would require before performing the action a
1465   with i's preconditions (i.e. predicate p).

1466 **8.6.5.7 confirm**

1467 $<i$, confirm(j,)>
1468 FP: B_i B_iU_j
1469 RE: B_j

1470 Confirm is a primitive communicative act.

1471 **8.6.5.8 Disconfirm**

1472 $<i$, disconfirm(j,)>
1473 FP: B_i $B_i(U_j$ B_j)
1474 RE: B_j

1475 Disconfirm is a primitive communicative act.

1476 **8.6.5.9 Failure**

1477 $<i$, failure(j, a, p)>
1478 $<i$, inform(j, (e) Single(e) Done(e, I_i Done(a)) p
1479 (p (Done(a) I_i Done(a))))>

1480 i informs j that, in the past, i had the intention to do action a, but because p was true, a was not done and i no longer
1481 has the intention to do a.

1482 **8.6.5.10 inform**

1483 $<i$, inform(j,)>
1484 FP: B_i $B_i(Bif_j$ Uif_j)
1485 RE: B_j

1486 Inform is a primitive communicative act.

1487 **8.6.5.11 inform-if**

1488 $<i$, inform-if(j, p)>
1489 $<i$, inform(j, p)> | $<i$, inform(j, p)>

1490 Inform-if represents two possible courses of action: i informs j that p, or i informs j that not p.

1491 **8.6.5.12 inform-ref**

1492 $<i$, inform-ref(j, x (x))>
1493 $<i$, inform(j, x (x) = r_1)> ... $<i$, inform(j, x (x) = r_k)>

1494 Inform-ref represents an unbounded, possibly infinite set of possible courses of action, in which i informs j of the
1495 referent of x.

1496 **8.6.5.13 query-if**

1497 $<i$, query-if(j,)
1498 $<i$, request(j, $<j$, inform-if(i,)>)>

1499 i requests j that j informs i whether or not is true.

1500 **8.6.5.14 query-ref**

1501 *<i*, query-ref(*j*, x (x))
1502   *<i*, request(*j*, <*j*, inform-ref(*i*, x (x))>)>

1503 i requests j that j informs i of the referent of x

1504 **8.6.5.15 refuse**

1505 *<i*, refuse(*j*, a,)>
1506   *<i*, disconfirm(*j*, Feasible(a))> ;
1507   *<i*, inform(*j*, ((Done(a) I$_i$ Done(a))))>

1508 i informs j that action a is not feasible, and further that, because of proposition , a has not been done and i has no
1509 intention to do a.

1510 **8.6.5.16 reject-proposal**

1511 *<i*, reject(*j*, <*j*, a>,)>
1512   *<i*, inform(*j*, I$_i$ Done(< *j*, a>))>

1513 i informs j that, because of proposition , i does not have the intention for j to perform action a.

1514 **8.6.5.17 request**

1515 *<i*, request(*j*, a)>
1516   FP: FP(a) [i\j] B$_i$ Agent(*j*, a) B$_i$ PG$_j$ Done(a)
1517   RE: Done(a)

1518 Request is a primitive communicative act.

1519 **8.6.5.18 request-when**

1520 *<i*, request-when(*j*, <*j*, a>, p)>
1521   *<i*, inform(*j*, I$_i$ (e) Enables(e, B$_j$p) After(e, <*j*, a>))>

1522 i informs j that i intends that, when some event happens that enables j to believe p, that event will be followed by j
1523 performing action a.

1524 **8.6.5.19 Request-whenever**

1525 *<i*, request-whenever(*j*, <*j*, a>, p)>
1526   *<i*, inform(*j*, I$_i$ Done(a, (e) Enables(e, B$_j$ p)))>

1527 *i* informs *j* that *i* intends that *j* will not believe *p* until *j* comes to believe *p* and also performs *a*.

1528 **8.6.5.20 subscribe**

1529 *<i*, subscribe(*j*, x (x))>
1530   *<i*, inform(*j*, I$_i$ (e) (e') (y)
1531   Feasible(e; e',
1532   Done(e', Bj (• x (x))=y)
1533   Bj (•x (x))=y)
1534   Feasible(e ; e', (e1) Feasible(e1)

1535   (e2) (e3)
1536   (e1 = (e2 ; <j, inform(i, (•x   (x)) = y)> ; e3))))>

1537   **8.6.5.21   not-understood**

1538   <*i*, not-understood(*j*, a)>
1539   FP: to be completed
1540   RE: to be completed

1541   not-understood is a primitive communicative act.

1542   8.7 Inter-agent Communication Plans

1543   The properties of rational behaviour stated above in the definitions of the concepts of rational effect and of feasibility
1544   preconditions for CA'S suggest an algorithm for CA planning. A plan is built up by this algorithm builds up through the
1545   inference of causal chain of intentions, resulting from the application of properties 1 and 2.

1546   With this method, it can be shown that what are usually called "*dialogue acts*" and for which models are postulated,
1547   are, in fact, complex plans of interaction. These plans can be derived from primitive acts, by using the principles of
1548   rational behaviour. The following is an example of how such plans are derived.

1549   The interaction plan "hidden" behind a question act can be more or less complex depending on the agent mental state
1550   when the plan is generated.

1551   Let a *direct question* be a question underlain by a plan which is limited to the reaction strictly legitimised by the
1552   question. Suppose that the main content of *i*'s mental state is:

1553 $B_I Bif_j$,

1554 $I_i Bif_i$

1555   By virtue of property 1, the intention is generated that the act <*j*, *INFORM-IF*(*i*,   )> be performed. Then, according to
1556   property 2, there follows the intention to bring about the feasibility of this act. Then, the problem is to know whether the
1557   following belief can be derived at that time from *i*'s mental state:

1558 $B_i(Bif_j$ ($B_j Bif_i$ Uif_i

1559   This is the case with *i*'s mental state. By virtue of properties 1 and 2, the intention that the act
1560   <*i*, *REQUEST* (*j*, <*j*, *INFORM-IF* (*i*,   )>)> be done and then the intention to achieve its feasibility, are inferred. The
1561   following belief is derivable:

1562 $B_i($ Bif_i Uif_i

1563   Now, no intention can be inferred. This terminates the planning process. The performance of a direct strict-yn-question
1564   plan can be started by uttering a sentence such as "Has the flight from Paris arrived?", for example.

1565   Given the FP's and the RE of the plan above, the following model for a *direct strict-yn-question plan* can be
1566   established:

1567   <*i*, *YNQUESTION*(*j*,   )>
1568   FP: $B_i Bif_j$ Bif_i Uif_i B_i $B_j(Bif_i$ Uif_i)
1569   RE: Bif_i

1570

9   References

To be completed.

[Austin 62] Austin J. L. How to Do Things with Words. *Clarendon Press* 1962

[Cohen & Levesque 90] Cohen P.R. & Levesque H.J. Intention is choice with commitment. *Artificial Intelligence,* 42(2-3):213--262, 1990.

[Cohen & Levesque 95] Cohen P.R. & Levesque H.J. Communicative actions for artificial agents; *Proceedings of the First International Conference on Multi-agent Systems (ICMAS'95)*, San Francisco, CA, 1995.

[Finin et al 97] Finin T., Labrou Y. & Mayfield J., KQML as an agent communication language, Bradshaw J. ed., *Sofware agents*, MIT Press, Cambridge, 1997.

[Freed & Borenstein 1996] Freed N & Borenstein N. Multipurpose Internet Mail Extensions (MIME) Part One: Format of the Internet Message Bodies. Internic RFC2045. `ftp://ds.internic.net/rfc/rfc2045.txt`

[Genesereth & Fikes 92] Genesereth M.R. & Fikes R.E. Knowledge interchange format. *Technical report Logic-92-1*, CS Department, Stanford University, 1992.

[Garson 84] Garson, G.W. Quantification in modal logic. In Gabbay, D., & Guentner, F. eds. *Handbook of philosophical logic, Volume II: Extensions of classical Logic*. D. Reidel Publishing Company: 249-307. 1984.

[Guinchiglia & Sebastiani 97] Guinchiglia F. & Sebastiani R., Building decision procedures for modal logics from propositional decision procedures: a case study of Modal K. *Proceedings of CADE 13*, published in Lecture Notes in Artificial Intelligence. 1997.

[Halpern & Moses 85] Halpern, *J.*Y., & Moses Y. A guide to the modal logics of knowledge and belief: a preliminary draft. *Proceedings of the* IJCAI-85, Los Angeles, CA. 1985.

[KQML93] External Interfaces Working Group, Specification of the KQML agent-communication language, 1993.

[Labrou & Finin 94]   Labrou Y. & Finin T., A semantic approach for KQML - A general purpose communication language for software agents, *Proceedings of the 3rd International Conference on Information Knowledge Management*, November 1994.

[Labrou 96] Labrou Y. Semantics for an agent communication language. *PhD thesis dissertation submission,* University of Maryland Graduate School, Baltimore, September, 1996.

[Sadek 90] Sadek M.D., Logical task modelling for Man-machine dialogue. *Proceedings of AAAI'90*: 970-975, Boston, MA, 1990.

[Sadek 91a] Sadek M.D. Attitudes mentales et interaction rationnelle: vers une théorie formelle de la communication. *Thèse de Doctorat Informatique, Université de Rennes I,* France, 1991.

[Sadek 91b] Sadek M.D. Dialogue acts are rational plans. *Proceedings of the ESCA/ETRW Workshop on the structure of multimodal dialogue,* pages 1-29, Maratea, Italy, 1991.

[Sadek 92] Sadek M.D. A study in the logic of intention. *Proceedings of the 3rd Conference on Principles of Knowledge Representation and Reasoning (KR'92),* pages 462-473, Cambridge, MA, 1992.

1604 **[Sadek et al 95]** Sadek M.D., Bretier P., Cadoret V., Cozannet A., Dupont P., Ferrieux A., & Panaget F. A co-operative
1605 spoken dialogue system based on a rational agent model: A first implementation on the AGS application. *Proceedings*
1606 *of the ESCA/ETR Workshop on Spoken Dialogue Systems : Theories and Applications*, Vigso, Denmark, 1995.

1607 **[Searle 69]** Searle J.R. *Speech Acts*, Cambridge University Press, 1969.

1608

Additional suggested reading

1610 **[Bretier & Sadek 96]** Bretier P. & Sadek D. A rational agent as the kernel of a cooperative spoken dialogue system:
1611 Implementing a logical theory of interaction. In Muller J.P., Wooldridge M.J., and Jennings N.R. (eds) *Intelligent agents*
1612 *III - Proceedings of the third ATAL*, LNAI, 1996.

1613 **[Sadek 94]** Sadek M. D. Belief reconstruction in communication*. Speech Communication Journal'94, special issue on*
1614 *Spoken Dialogue*, 15(3-4), 1994.

1615 **[Sadek et al 97]** Sadek M. D., P. Bretier, & F. Panaget. ARTIMIS: Natural language meets rational agency.
1616 *Proceedings of IJCAI '97*, Nagoya, Japan, 1997.

1617

1618

<div align="center">

Annex A

(informative)

</div>

<div align="center">

ACL Conventions and Examples

</div>

1622 This annex describes certain conventions that, while not a mandatory part of the specification, are commonly adopted
1623 practices that aid effective inter-agent communications. This annex will also serve to provide examples of ACL usage
1624 for illustrative purposes.

A.1 Conventions

A.1.1 Conversations amongst multiple parties in agent communities

1627 There is commonly a need in inter-agent dialogues to involve more than two parties in the conversation. A typical
1628 example would be of agent *i* posing a question to agent *j* by sending a *query-if* message. Agent *i* believes that *j* is able
1629 to answer the query, but in fact *j* finds it necessary to delegate some or all of the task of answering the question to
1630 another agent *k*.

1631 The formal definition of the query-if communicative act reads that *i* is requesting *j* that *j informs i* of the truth of
1632 proposition *p*. Therefore, even if *j* does delegate all of the query to *k*, the semantics of ACL requires that *j* will be the
1633 one to perform the act of informing *i*. *K* cannot inform *i* directly. By extension, any chain of such delegation acts will
1634 have to be unwound in reverse order to conform to the current specification.

1635 The restriction that a delegating agent in such a scenario must, in effect, remain "in the loop" clearly does not alter the
1636 meaning of the act (except, perhaps, that it exposes *i* to the existence of *k*), but it can be critiqued on the grounds of
1637 overall efficiency. A future version of this specification may generalise the semantic definition to allow delegation which
1638 includes passing responsibility for answering the originator of the request directly.

1639 See also §A.1.4 *Negotiating by exchange of goals*.

A.1.2 Maintaining threads of conversation

1641 Agents are frequently implemented with the ability to participate in more than one conversation at the same time.
1642 These conversations may all be with different agents, or may be with the same agent but in the context of different
1643 tasks or subjects. The internal representation and maintenance of structures to manage the separate conversations is
1644 a matter for the agent designer. However, there must be some support in the ACL for the concept of separate
1645 conversations, else an agent will have no standardised way of disambiguating the conversational context in which to
1646 interpret a given message. ACL supports conversation threading through the use of standard message parameters
1647 which agents are free (but not required) to use. These are: *:reply-with*, *:in-reply-to* and *:conversation*. Additional
1648 contextual information to assist the agent to interpret the meaning of a message is provided through the protocol
1649 identifier, *:protocol*.

1650 The first case is one of annotating a message which is expected to generate a response with an expression which
1651 serves to abbreviate the context of the enquiry. This abbreviation is then cross-referenced in the reply. For example,
1652 agent i asks agent j if the summer in England was wet. Without any ability to refer back to the question, j cannot simply
1653 say "yes" because that would be potentially ambiguous. J can disambiguate its reply by saying "yes, the summer in
1654 England was wet", or it could say "in response to your question, the answer is yes". Different styles and
1655 implementations of agents might adopt either of these tactics. The latter case is performed through the use of *:reply-*
1656 *with* and *:in-reply-to*. The *:reply-with* parameter is used to introduce an abbreviation for the query, *:in-reply-to* is used to
1657 refer back to it. For example:

```
1658        (ask-if
1659            :sender I
1660            :receiver j
1661            :content (= (weather England (summer 1997)) wet)
1662            :ontology meteorology
1663            :reply-with query-17)
1664
1665        (inform
1666            :sender j
1667            :receiver I
1668            :content true
1669            :in-reply-to query-17)
```

1670   In addition to maintaining context over instances of exchanges of communicative acts, the agents may also wish to
1671   maintain a longer lived conversational structure. They may be exchanging information about the weather in the UK,
1672   and at the same time be discussing that of Peru. The conversation can provide additional interpretative context: for
1673   example the question "what was the weather like in the summer?" is meaningful in the context of a conversation about
1674   UK meteorology, and rather less so if no such context is known. In addition, the conversation may simply be used by
1675   the agent to manage its communication activities, particularly if conversations are strongly link to current tasks. The
1676   parameter *:conversation-id* is used to denote a word which identifies the conversation.

A.1.3   Initiating sub-conversations within protocols

1678   The use of protocols (c.f. §7) in agent interactions is introduced in order to provide a tool that facilitates the
1679   simplification of the design of some agents, since the agent can expect to know which messages are likely to be
1680   received or need to be generated at each stage of the conversation. However, this simplicity can also be restrictive:
1681   there may legitimately be cause to step outside the prescribed bounds of the protocol. For example, in a contract net
1682   protocol, the manager sends out a *cfp* message, which should normally be followed by a *propose* or a refusal. Suppose
1683   that the contractor, however, wishes some additional information (perhaps a clarification). Replying to the *cfp* with, for
1684   example, a *query-if* action would break the protocol. While agents with powerful, complete reasoning capabilities can
1685   be expected to deal appropriately with such an occurrence, simpler agents, adhering closely to the protocol, may not.
1686   Nor is it a solution to anticipate all such likely responses in the protocol: such anticipation is unlikely to cover every
1687   possibility, and anyway the resulting complexity would defeat the primary purpose of the protocol.

1688   Instead, the convention is suggested that adopting a new conversation-id (see above) for a reply is sufficient to indicate
1689   to the receiver that the reply should not be considered the next step in the protocol. It should not cause a not-
1690   understood message to be generated (the normal occurrence if a protocol is broken unexpectedly). A problem remains
1691   that adopting a new conversation-id does not make available to the agents involved the convenience of knowing that a
1692   rich context is shared. This release of the specification does not address the issue of structured conversation-id's, in
1693   which the idea of a context-sharing sub-conversation is supported, though a future version may do so. In the interim, it
1694   is suggested that, where a given domain finds that this capability is a necessity, a domain specific solution to the
1695   problem of defining conversation-id's is adopted.

A.1.4   Negotiating by exchange of goals

1697   A common practice amongst agent communities is to interact and negotiate at the level of goals and commitments,
1698   rather than explicit commands. Indeed, some researchers will say that such *indirect manipulation* is one of the most
1699   compelling arguments for the effectiveness of the agent technology paradigm.

1700   While the ACL semantics does include a concept of goal and intention, the core communicative act for influencing
1701   another agent's behaviour is the *request* action. The main argument to request is an action, not a goal, which requires
1702   the requesting agent to be aware of the actions that another agent can perform, and to plan accordingly. In many
1703   instances, the agent may wish to communicate its objectives, and leave the reasoning and planning towards the
1704   achievement of those objectives to the recipient agent.

1705   Since no *achieve-goal* action is currently built-in to the ACL, it is common to embed the goal in an expression in the
1706   chosen content language which expresses the *action of achieving the goal*. This action can then be requested by the
1707   sending agent. Precise details of such a goal encoding depend on the chosen content language. An example might be:

```
1708            (request
1709                :sender i
1710                :receiver j
1711                :content (achieve (at (location 12 84) box17))
1712                :ontology factory-management
1713                :reply-with query-17)
1714
```

1715   Note, for symmetry, that a converse domain action `achieved` can also be used to map actions to goals.

A.2   Additional examples

A.2.1   Actions and results

1718   In general, the semantic model underlying the ACL states that an action does not have a value. Clearly all actions have
1719   effects, which are causally related to the performance of the action. However, it may be difficult or impossible to
1720   determine the causal effects of an action. Even *a posterori* observation may not be able to determine all of the effects
1721   of an action. Thus, in general, actions do not have a result. SL allows the capture of some intuitive notions about the
1722   effects of actions by associating the occurrence of the action with statements about the state of the world through the
1723   *Done* and *Feasible* operators.

1724   However, there is a class of actions which are defined as computational activities, in which it is useful to say that the
1725   action has a result. For example, the action of adding two and two in a computational device. These actions are related
1726   to the result they produce through the result predicate, which is the remit of a content language and given domain
1727   theory. In defining the result predicate, it should be noted that it takes as an argument a term, not an action which is a
1728   separate category.

1729   Consider the following three example actions:

```
1730        A:  (request     :sender i  :receiver j
1731                :content (action j action))
1732
1733        B:  (query-ref  :sender i :receiver j
1734                :content (iota ?x (result (action-term j action) ?x))
1735
1736        C:  (request     :sender i :receiver j
1737                :content (action j action))      ;
1738            (inform-ref :sender j :receiver i
1739                :content (iota ?x (result (action-term j action) ?x)))
```

1740   The question then arises as to the differences between these actions. In summary, the meaning of the actions, are,
1741   respectively:

1742   *A:* Agent i says to j "do *action*", but does not say anything about the result

1743   *B:* Agent i says to j "tell me the result of doing *action*"

1744   *C:* Agent i says to j "do *action*, and then inform me of the result of doing *action*".

1745   In action B, the question can legitimately be asked whether the action is actually performed or not. It should be noted
1746   that *result* is a function in the domain language, SL in this case. Thus this question must really be devolved to the

1747   domain representation language. Some languages may be able to compute the meaning of an action without
1748   performing that action: this would be very useful for planning agents who may not wish to perform an action before
1749   considering its likely effects[11]. Other agents, such as expression simplifiers, do not want to be overburdened with the
1750   complexity of performing the simplification, then separately having to inform the questioner of the result of the
1751   simplification. Of course, if the meaning of the result predicate in a given context is that the action does, in fact, get
1752   done, then example C will likely result in the action being done twice.

1753

---

[11] Consider the bomb disposal agent being asked "what is [i.e. would be] the effect of cutting the red wire?". Agents which are able to reason about the future consequences of their actions are likely to differentiate between the operation of observing the effects of an action (*result* predicate) and predicting the effects (an *effect-of* prediate perhaps).

Annex B

(normative/informative)

SL as a Content Language

1753
1754
1755
1756

This annex introduces a concrete syntax for the SL language that is compatible with the description in §8. This syntax, and its associated semantics, are suggested as a candidate *content language* for use in conjunction with FIPA ACL. In particular, the syntax is defined to be a sub-grammar of the very general s-expression syntax specified for message content in §6.4.

This content language is included in the specification on an *informative* basis. *It is not mandatory for any FIPA specification agent to implement the computational mechanisms necessary to process all of the constructs in this language.* However, SL is a general purpose representation formalism that may be suitable for use in a number of different agent domains.

Statement of conformance
The following definitions of SL, and subsets SL0, SL1 and SL2 are *normative defininitions* of these languages. That is, if a given agent chooses to implement a parser/interpreter for these languages, the following definitions must be adhered to. However, these languages are *informative suggestions* for the *use* of a content language: no agent is required as part of part 2 of this FIPA 97 specification to use the following content languages. However it should be noted that certain other parts of the FIPA 97 specification do make normative use of (some of) the following languages.

B.1　Grammar for SL concrete syntax

```
SLContentExpression         = SLWff
                            | SLIdentifyingExpression
                            | SLActionExpression.

SLWff                       = SLAtomicFormula
                            | "(" "not"          SLWff ")"
                            | "(" "and"          SLWff SLWff ")"
                            | "(" "or"           SLWff SLWff ")"
                            | "(" "implies"      SLWff SLWff ")"
                            | "(" "equiv"        SLWff SLWff ")"
                            | "(" SLQuantifier   SLVariable SLWff ")"
                            | "(" SLModalOp  SLAgent SLWff ")"
                            | "(" SLActionOp SLActionExpression ")".
                            | "(" SLActionOp
                                    SLActionExpression SLWff ")".

SLAtomicFormula             = SLPropositionSymbol
                            | "(" "=" SLTerm SLTerm ")"
                            | "(" "result" SLTerm SLTerm ")"
                            | "(" SLPredicateSymbol SLTerm* ")"
                            | true
                            | false.

SLQuantifier                = "forall"
                            | "exists".
```

```
1794   SLModalOp                  = "B"
1795                              | "U"
1796                              | "PG"
1797                              | "I".

1798   SLActionOp                 = "feasible"
1799                              | "done".

1800   SLTerm                     = SLVariable
1801                              | SLConstant
1802                              | SLFunctionalTerm
1803                              | SLActionExpression
1804                              | SLIdentifyingExpression.

1805   SLIdentifyingExpression    = "(" "iota" SLVariable SLWff ")"

1806   SLFunctionalTerm           = "(" SLFunctionSymbol SLTerm* ")".

1807   SLConstant                 = NumericalConstant
1808                              | Word
1809                              | StringLiteral.

1810   NumericalConstant          = IntegerLiteral
1811                              | FloatingPointLiteral.

1812   SLVariable                 = VariableIdentifier.

1813   SLActionExpression         = "(" "action" SLAgent SLFunctionalTerm ")"
1814                              | ACLCommunicativeAct
1815                              | "(" "|" SLActionExpression SLActionExpression ")"
1816                              | "(" ";" SLActionExpression SLActionExpression ")".

1817   SLPropositionSymbol        = Word.

1818   SLPredicateSymbol          = Word.

1819   SLFunctionSymbol           = Word.

1820   SLAgent                    = AgentName.
```

1821   ## B.1.1 Lexical definitions

```
1822   Word                       = [~ "\0x00" - "\0x1f",
1823                                   "(", ")", "#", "0"-"9", "-", "?"]
1824                                [~ "\0x00" - "\0x1f",
1825                                   "(", ")"] *.

1826   VariableIdentifier         = "?"
1827                                [~ "\0x00" - "\0x1f",
1828                                   "(", ")"] *.

1829   IntegerLiteral             = ( "-" )? DecimalLiteral
1830                              | ( "-" )? HexLiteral.

1831   FloatingPointLiteral       = (["0"-"9"])+ "." (["0"-"9"])+ (Exponent)?
1832                              | (["0"-"9"])+ Exponent.
```

```
1833   DecimalLiteral              = ["0"-"9"]+.

1834   HexLiteral                  = "0" ["x", "X"] (["0"-"9","a"-"f","A"-"F"])+.

1835   Exponent                    = ["e", "E"] (["+","-"])? (["0"-"9"])+.

1836   StringLiteral               = "\""
1837                                  ( [~ "\"" ] | "\\\"" )*
1838                                  "\"".
```

1839   B.2   Notes on SL content language semantics

1840   This section contains explanatory notes on the intended semantics of the constructs introduced in §B.1 above.

1841   B.2.1   Grammar entry point: SL content expression

1842   An SL content expression may be used as the content of an ACL message. There are three cases:

1843   A proposition, which may be assigned a truth value in a given context. Precisely, it is a well-formed formula
1844   using the rules described in SLWff. A proposition is used in the *inform* act, and other acts derived from it.

1845   An action, which can be performed. An action may be a single action, or a composite action built using the
1846   sequencing and alternative operators. An action is used as a content expression when the act is the *request*
1847   act, and other CA's derived from it.

1848   An identifying reference expression (IRE), which identifies an object in the domain. This is the iota operator,
1849   and is used in the *inform-ref* macro act and other acts derived from it.

1850   B.2.2   SL Well-formed formula (SLWff)

1851   A well-formed formula is constructed from an atomic formula, whose meaning will be determined by the semantics of
1852   the underlying domain representation, or recursively by applying one of the construction operators or logical
1853   connectives described in the grammar rule. These are:

1854   `(not <SLWff>)`
1855   Negation. The truth value of this expresion is false if *SLWff* is true. Otherwise it is true.

1856   `(and <SLWff0> <SLWff1>)`
1857   Conjunction. This expression is true iff well-formed formulae *SLWff0* and *SLWff1* are both true, otherwise it is
1858   false.

1859   `(or <SLWff0> <SLWff1>)`
1860   Disjunction. This expression is false iff well-formed formulae *SLWff0* and *SLWff1* are both false, otherwise it is
1861   true.

1862   `(implies <SLWff0> <SLWff1>)`
1863   Implication. This expression is true if either *SLWff0* is false, or alternatively if *SLWff0* is true and *SLWff1* is
1864   true. Otherwise it is false. The expression corresponds to the standard material implication connective:
1865   SLWff0   SLWff1.

1866   `(equiv <SLWff0> <SLWff1>)`
1867   Equivalence. This expression is true if either *SLWff0* is true and *SLWff1* is true, or alternatively if *SLWff0* is
1868   false and *SLWff1* is false. Otherwise is is false.

1869   `(forall <variable> <SLWff>)`

1870   Universal quantification. The quantifed expression is true if *SLWff* is true for every value of value of the

1871   quantified variable.

1872   `(exists <variable> <SLWff>)`

1873   Existential quantification. The quantifed expression is true if there is at least one value for the variable for

1874   which *SLWff* is true.

1875   `(B <agent> <expression>)`

1876   It is true that *agent* believes that *expression* is true.

1877   `(U <agent> <expression>)`

1878   It is true that *agent* is uncertain of the truth of *expression*. *Agent* neither believes *expression* nor its negation,

1879   but believes that *expression* is more likely to be true than its negation.

1880   `(I <agent> <expression>)`

1881   It is true that *agent* intends that *expression* becomes true, and will plan to bring it about.

1882   `(PG <agent> <expression>`)

1883   It is true that *agent* holds a persistent goal that *expression* becomes true, but will not necessarily plan to bring

1884   it about.

1885   (feasible <SLActionExpression> <SLWff>)

1886   It is true that action *SLActionExpression* (or, equivalently, some event) can take place, and just afterwards

1887   *SLWff* will be true.

1888   (feasible <SLActionExpression>)

1889   Same as (feasible <SLActionExpression> true).

1890   (done <SLActionExpression> <SLWff>)

1891   It is true that action *SLActionExpression* (or, equivalently, some event) has just taken place, and just before

1892   that *SLWff* was true.

1893   (done <SLActionExpression>)

1894   Same as (done <SLActionExpression>, true)

1895   **B.2.3   SL Atomic Formula**

1896 The atomic formula represents an expression which has a truth value in the language of the domain of discourse.

1897 Three forms are defined: a given propositional symbol may be defined in the domain language, which is either true or

1898 false; two terms may or may not be equal under the semantics of the domain language; or some predicate is defined

1899 over a set of zero or more arguments, each of which is a term.

1900 The SL representation does not define a meaning for the symbols in atomic formulae: this is the responsibility of the

1901 domain language representation and ontology.

1902   **B.2.4   SL Term**

1903 Terms are the arguments to predicates, and are either themselves atomic (constants and variables), or recursively

1904 constructed as a functional term in which a functor is applied to zero or more arguments. Again, SL only mandates a

1905 syntactic form for these terms. With small number of exceptions (see below), the meanings of the symbols used to

1906 define the terms are determined by the underlying domain representation.

1907   Note, as mentioned above, that no legal well-formed expression contains a free variable, that is, a variable not
1908   declared in any scope within the expression. Scope introducing formulae are the quantifiers (*forall*, *exists*) and the
1909   reference operator *iota*. Variables may only denote terms, not well-formed formulae.

1910   The following special term is defined:

1911   (iota <variable> <term>)
1912   The iota operator introduces a scope for the given *expression* (which denotes a term), in which the given
1913   *identifier*, which would otherwise be free, is defined. An expression containing a free variable is not a well-
1914   formed SL expression. The expression "(iota x (P x)" may be read as "the x such that P [is true] of x. The *iota*
1915   operator is a constructor for terms which denote objects in the domain of discourse.

B.2.5   Result predicate

1917   A common need is to determine the result of performing an action or evaluating a term. To facilitate this operation, a
1918   standard predicate *result*, of arity two, is introduced to the language. Result/2 has the declarative meaning that the
1919   result of evaluating the term, or equivalently of performing the action, encoded by the first argument term, is the second
1920   argument term. However, it is expected that this declarative semantics will be implemented in a more efficient,
1921   operational way in any given SL interpreter.

1922   A typical use of the *result* predicate is with a variable scoped by *iota*, giving an expression whose meaning is, for
1923   example, "the *x* which is the result of agent *i* performing *act*":

1924   (iota x (result (action i *act*) x)))

B.2.6   Actions and action expressions

1926   Action expressions are a special subset of terms. In particular, three functional term functors are reserved: "*action*", "*|*"
1927   and "*;*". An action itself is introduced by the keyword "*action*", and comprises the agent of the action (i.e. an identifier
1928   representing the agent performing the action) and a term denoting the action which is [to be] performed. An alternative
1929   form of action is precisely the ACL communicative act. For syntactic rules, see §6.4.

1930   Two operators are used to build terms denoting composite acts:

1931   the sequencing operator ";" denotes a composite act in which the first action (the represented by the first
1932   operand) is followed by the second action;

1933   the alternative operator "|" denotes a composite act in which either the first action occurs, or the second, but
1934   not both.

B.2.7   Agent identifier

1936   An agent is represented by referring to its name. The name is defined using the standard format from part one of this
1937   specification, which is repeated in §17

B.2.8   Numerical Constants

1939   Due to the necessarily unpredictable nature of cross-platform dependencies, agents should not make strong
1940   assumptions about the precision with which another agent is able to represent a given numerical value. SL assumes
1941   only 32 bit representations of both integers and floating point numbers. Agents should not exchange message contents
1942   containing numerical values requiring more than 32 bits to encode precisely, unless some prior arrangement is made to
1943   ensure that this is valid.

1944   **B.3   Reduced expressivity subsets of SL**

1945   The SL definition given above is a very expressive language, but for some agent communication tasks it is
1946   unnecessarily powerful. This expressive power has an implementation cost to the agent, and introduces problems of
1947   the decidability of modal logic. To allow simpler agents, or agents performing simple tasks to do so with minimal
1948   computational burden, this section introduces semantic and syntactic subsets of the full SL language for use by the
1949   agent when it is appropriate or desirable to do so. These subsets are defined by the use of *profiles*, that is, statements
1950   of restriction over the full expressiveness of SL. These profiles are defined in increasing order of expressiveness as
1951 SL_0, SL_1 and SL_2.

1952   Note that these subsets of SL, with additional ontological commitments (i.e. the definition of domain predicates and
1953   constants) are used in other parts of the FIPA 97 specification.

1954   **B.3.1   SL0: minimal subset of SL**

1955   Profile 0 is denoted by the normative constant SL0 in the `:language` parameter of an ACL message.

1956   Profile 0 of SL is the minimal subset of the SL content language. It allows the representation of actions, the
1957   determination of the result a term representing a computation, the completion of an action and simple binary
1958   propositions.

1959   The following defines the SL0 grammar:

```
1960    SL0ContentExpression        = SL0Wff
1961                                | SL0ActionExpression.
1962
1963    SL0Wff                      = SL0AtomicFormula
1964                                | "(" SL0ActionOp SL0ActionExpression ")".
1965
1966    SL0AtomicFormula            = SLPropositionSymbol
1967                                | "(" "result" SL0Term SL0Term ")"
1968                                | "(" SLPredicateSymbol SL0Term* ")"
1969                                | "true"
1970                                | "false".
1971
1972    SL0ActionOp                 = "done".
1973
1974    SL0Term                     = SLVariable
1975                                | SLConstant
1976                                | SL0FunctionalTerm
1977                                | SL0ActionExpression.
1978
1979    SL0ActionExpression         = "(" "action" SLAgent SL0FunctionalTerm ")"
1980                                | ACLCommunicativeAct.
1981
1981    SL0FunctionalTerm           = "(" SLFunctionSymbol SL0Term* ")"
1982
```

1983   **B.3.2   SL1: propositional form**

1984   Profile 1 is denoted by the normative constant SL1 in the `:language` parameter of an ACL message.

1985   Profile 1 of SL extends the minimal representational form of SL0 by adding Boolean connectives to represent
1986   propositional expressions.

1987   The following defines the SL1 grammar:

```
1988    SL1ContentExpression      = SL1Wff
1989                              | SL1ActionExpression.
1990
1991    SL1Wff                    = SL1AtomicFormula
1992                              | "(" "not"           SL1Wff ")"
1993                              | "(" "and"           SL1Wff SL1Wff ")"
1994                              | "(" "or"            SL1Wff SL1Wff ")"
1995                              | "(" SL1ActionOp SL1ActionExpression ")".
1996
1997    SL1AtomicFormula          = SLPropositionSymbol
1998                              | "(" "result" SL1Term SL1Term ")"
1999                              | "(" SLPredicateSymbol SL1Term* ")"
2000                              | "true"
2001                              | "false".
2002
2003    SL1ActionOp               = "done".
2004
2005    SL1Term                   = SLVariable
2006                              | SLConstant
2007                              | SL1FunctionalTerm
2008                              | SL1ActionExpression.
2009
2010    SL1ActionExpression       = "(" "action" SLAgent SL1FunctionalTerm ")"
2011                              | ACLCommunicativeAct.
2012
2013
2014    SL1FunctionalTerm         = "(" SLFunctionSymbol SL1Term* ")".
2015
```

2016   **B.3.3   SL2: restrictions for decidability**

2017   Profile 2 is denoted by the normative constant SL2 in the `:language` parameter.

2018   Profile 2 of SL is a subset of the SL content language which still allows first order predicate and modal logic, but is
2019   restricted to ensure that it is decidable. Well-known effective algorithms exist (for instance KSAT and Monadic
2020   **[references? –ed]**) that can derive whether or not an SL2 wff is a logical consequence of a set of wffs.

2021   The following defines the SL2 grammar:

```
2022    SL2ContentExpression      = SL2Wff
2023                              | SL2QuantifiedExpression
2024                              | SL2IdentifyingExpression
2025                              | SL2ActionExpression.
2026
2027    SL2Wff                    = SL2AtomicFormula
2028                              | "(" "not"           SL2Wff ")"
2029                              | "(" "and"           SL2Wff SL2Wff ")"
2030                              | "(" "or"            SL2Wff SL2Wff ")"
2031                              | "(" "implies"       SL2Wff SL2Wff ")"
2032                              | "(" "equiv"         SL2Wff SL2Wff ")"
2033                              | "(" SLModalOp  SLAgent SL2QuantifiedExpression ")"
2034                              | "(" SLActionOp SL2ActionExpression ")"
2035                              | "(" SLActionOp SL2ActionExpression
2036                                       SL2UnivExistQuantWff ")".
2037
```

```
2038   SL2AtomicFormula              = SLPropositionSymbol
2039                                 | "(" "=" SL2Term SL2Term ")"
2040                                 | "(" "result" SL2Term SL2Term ")"
2041                                 | "(" SLPredicateSymbol SL2Term* ")"
2042                                 | "true"
2043                                 | "false".
2044
2045   SL2QuantifiedExpression       = SL2UnivQuantExpression
2046                                 | SL2ExistQuantExpression
2047                                 | SL2Wff.
2048
2049   SL2UnivQuantExpression        = "(" "forall" SL2variable SL2Wff ")"
2050                                 | "(" "forall" SL2variable SL2UnivQuantExpression ")".
2051                                 | "(" "forall" SL2variable SL2ExistQuantExpression ")".
2052
2053   SL2ExistQuantExpression       = "(" "exists" SL2variable SL2Wff ")"
2054                                 | "(" "exists" SL2variable S2ExistQuantExpression ")"
2055
2056   SL2Term                       = SLVariable
2057                                 | SLConstant
2058                                 | SL2FunctionalTerm
2059                                 | SL2ActionExpression
2060                                 | SL2IdentifyingExpression.
2061
2062   SL2IdentifyingExpression      = "(" "iota" SLVariable SL2Wff ")"
2063
2064   SL2FunctionalTerm             = "(" SLFunctionSymbol SL2Term* ")".
2065
2066   SL2ActionExpression           = "(" "action" SLAgent SL2FunctionalTerm ")"
2067                                 | ACLCommunicativeAct
2068                                 | "(" "|" SL2ActionExpression SL2ActionExpression ")"
2069                                 | "(" ";" SL2ActionExpression SL2ActionExpression ")".
2070
```

2071 That is the SL2Wff production no longer directly contains the logical quantifiers, but these are treated separately to
2072 ensure only prefixed quantified formulas, such as:

```
2073          (forall ?x1 (forall ?x2
2074            (exists ?y1 (exists ?y2
2075                (Phi ?x1 ?x2 ?y1 ?y2) )) ))
```

2076 where (Phi ?x1 ?x2 ?y1 ?y2) does not contain any quantifier.

2077 The grammar of SL2 still allows for *quantifying-in* inside modal operators. E.g. the following formula is still admissible
2078 under the grammar:

```
2079          (forall ?x1
2080                  (or
2081                    (B i (p ?x1))
2082                    (B j (q ?x1)) ))
```

2083 It is not clear that formulae of this kind are decidable. However, changing the grammar to express this context
2084 sensitivity would make the EBNF form above essentially unreadable. Thus the following additional mandatory
2085 constraint is placed on well-formed content expressions using SL2:

2086 **Within the scope of an SLModalOperator only closed formulas are allowed, i.e. formulas without free variables.**

2087

Annex C

<p style="text-align:center">2087</p>

(informative)

<p style="text-align:center">2088</p>

2089

Relationship of ACL to KQML

2090

2091 This annex outlines some of the primary similarities and differences between FIPA ACL and the *de facto* standard
2092 agent communication language KQML (Knowledge Querying and Communication Language) [Finin et al 97]. The
2093 intention of this appendix is not to deliver a complete characterisation of KQML (which is an evolving language in itself
2094 anyway) and the differences between it and ACL, but simply to outline some key areas of difference as an aide to
2095 readers already familiar with KQML.

2096 **C.1 Primary similarities and differences**

2097 Both KQML and ACL are interlingua languages, intended to provide a common linguistic basis for independent agents
2098 to communicate with each other. Both languages are based on speech act theory, which states that individual
2099 communications can be reduced to one of a small number of primitive *speech*, or more generally, *communicative* acts,
2100 which shape the basic meaning of that communication. The full meaning is conveyed by the meaning that the speech
2101 act itself imparts to the content of the communication. In KQML, the speech act is called the *performative*, though it
2102 should be noted that some researchers prefer other terms.

2103 Syntactically, KQML sets out to be simple to parse and generate, yet easily human readable. To this end, KQML's
2104 syntax is Lisp based (Lisp sharing similar syntactic goals, as well as being an early implementation vehicle for KQML):
2105 each message is an s-expression and uses a core of Lisp-like tokenising rules. Some extensions are added to allow for
2106 the encoding of content in arbitrary other notations. FIPA ACL adopts a very similar syntax, including the form of
2107 messages and message parameters. Some differences exist in the names of both the message type keywords and the
2108 parameter keywords. Both languages can be challenged in the compactness of their encoding; ACL explicitly notes
2109 that future revisions may include one or more alternative transport syntaxes optimised for message compactness.

2110 KQML was designed originally to fulfil a very pragmatic purpose as part of the Knowledge Sharing Effort (KSE)
2111 consortium. Initially, the semantics of the performatives were described informally by natural language descriptions.
2112 Subsequent research has addressed the need for a more precise semantics [Labrou 96], though it is not clear that the
2113 proposed semantics has been universally adopted. Indeed, several flavours of KQML are extant. ACL is derived from
2114 the research work of Sadek et al [Sadek et al '95], and was designed from its inception to be grounded in a formally
2115 defined semantics.

2116 KQML aims to serve several needs in inter-agent communication. These can be summarised as:

2117   querying and information passing (e.g. evaluate, ask-if, tell, achieve, etc)

2118   managing multiple responses to queries (e.g. ask-all, stream-all, standby, ready, next, etc)

2119   managing capability definition and dissemenation (advertise, recommend, etc)

2120   managing communications (e.g. register, forward, broadcast, etc)

2121 That these are all needs that must be addressed in inter-agent communication (in the general case, at least) is clear.
2122 KQML attempts to define a core set of performatives that together meet all of these needs, while balancing a desire for
2123 parsimony in the language. ACL does not attempt to cover all of these needs within the language. Instead, some
2124 categories are explicitly devolved to the agent management system (see part 1 of the FIPA 97 specification) or are the
2125 responsibility of the content language (notably managing multiple responses to queries).

2126   **C.2 Correspondence between KQML message performatives and FIPA CA's**

2127   This section outlines some specific categories of KQML messages and the (approximately) equivalent constructs in the
2128   ACL and other sections of the FIPA specification.

2129   **C.2.1 Agent management primitives**

2130   Some of the message types included in KQML can not be considered speech acts in the traditional sense, but do have
2131   a useful role to play in mediating conversations between agents and providing capabilities to manage an agent society.
2132   This specification adopts the position that, despite the arguable increase in complexity, it is better to clearly separate
2133   such concerns from the core communication primitives. Thus, equivalents to the following KQML messages are not
2134   directly included in the ACL specification:

2135   register

2136   unregister

2137   recommend (-one, -all)

2138   recruit (-one, -all)

2139   broker (-one, -all)

2140   advertise

2141   Instead, effects similar or equivalent to these messages can be obtained by embedding the agent management
2142   primitives defined in part one of the FIPA 97 specification, embedded in an ACL *request* act addressed to the
2143   appropriate facilitator agent.

2144   **C.2.2 Communications management**

2145   Similarly, the following KQML performatives find their equivalents in the FIPA specification as agent management
2146   actions, communicated via a *request* act:

2147   broadcast

2148   transport-address

2149   forward

2150   In the last case, *forward* is one solution to the problem of sending a message to an agent whose agent identifier or
2151   network transport address are not known at the time of sending the message. In the semantics of KQML, each
2152   intermediary does not interpret the message embedded within the *forward* performative, and thus does not perform any
2153   action implied by it. This capability does exist in the FIPA specification using the agent management capabilities
2154   defined in part one of this specification.

2155   **C.2.3 Managing multiple solutions**

2156   There is frequently a need to convey more than one answer to an enquiry. This may be because the query was under-
2157   constrained, or may be due to the nature of the application, e.g. selecting records from a database. KQML provides a
2158   number of mechanisms for handling multiple queries at the message level:

2159   sender asks replier to send any solution (ask-one)

2160   sender asks replier to send all solutions (ask-all)

2161   sender asks replier to send all solutions, each one in its own message (stream-all) and then to demark the
2162   end of the solution stream (eos)

2163   sender asks replier to set up a solution generator; a protocol then exists to test, acces and destroy the
2164   generator (standby, ready, next, rest, discard).

2165 Although enquiring is a general and very useful category of speech acts, these performatives suffer from being
2166 complicated by assumptions about the representational form of the content of the reply. ACL takes the position that the
2167 requirement for managing multiple solutions is properly the remit of the content language. For example, if an
2168 application requires a solution generator, of the kind implied by KQML standby, etc, such a construct should be a part
2169 of domain content language. Operations on the generator object would then be the subject of generic *request* acts.

2170 ## C.2.4 Other discourse performatives

2171 The following discusses the remaining performatives in the core KQML specification. Note that statements of
2172 equivalence in the following list are advisory only, since there is no universally accepted KQML formal semantics to
2173 check against ACL semantics for equivalence.

2174   *ask-if*: nearest equivalent in ACL is *query-if*

2175   *tell*: equivalent to ACL's $<i, inform(\ j, B_i\ p)>$

2176   *untell*: equivalent to $<i, inform(\ j,\ \ B_i\ p)>$

2177   *deny*: equivalent to $<i, inform(\ j, B_i\ \ p)>$ or $<i, disconfirm(\ j, p\ )>$

2178   *insert, uninsert*: these performatives are not supported in ACL, since an agent is not given the power to
2179   directly manipulate the beliefs of another agent. Use *inform* and *disconfirm* instead.

2180   *delete-(one, all), undelete*: these performatives are not supported in ACL, since an agent is not given the
2181   power to directly manipulate the beliefs of another agent.

2182   *achieve*: goals can be communicated among agents through the use of an achieve domain-language
2183   primitive, if that is appropriate to the domain (see §A.1.4)

2184   *unachieve*: KQML's unachieve is a kind of undo action: the recipient is asked to return the world (or at least,
2185   that part it has control over) to the state it was in before the corresponding achieve. There is no equivalent to
2186   this action in ACL. If a given domain is able to support such an action (e.g. the domain of text editing), specific
2187   actions may be defined in the domain ontology to support undo actions.

2188   *subscribe*: equivalent to the *subscribe* in ACL

2189   *error*: use *not-understood*

2190   *sorry*: use *refuse* or *failure*.

2191

2191 **Annex D**
2192 (informative)
2193
2194 **MIME-encoding to extend content descriptions**

2195 This Annex provides a means for agents to extend the representational capability of a given message content by using
2196 MIME style content description and encoding.

2197 **D.1 Extension of FIPA ACL to include MIME headers**

2198 The MIME enhancements extend the grammar shown in §6.4 as follows:

```
2199 MIMEEnhancedExpression      = Word
2200                             | String
2201                             | Number
2202                             | MIMEEncapsulatedExpression
2203                             | "(" MIMEEnhancedExpression * ")".
2204
2205 MIMEEncapsulatedExpression = "(" MIMEVersionField
2206                                 MIMEOptionalHeader *
2207                                 MIMEEnhancedExpression
2208                             ")".
2209
2210 MIMEVersionField            = "(" "MIME-Version 1.0 (FIPA ACL Message)" ")".
2211
2212 MIMEOptionalHeader          = "(" "Content-type:" MIME_CT_Expression ")"
2213                             | "(" "Content-Transfer-Encoding:" MIME_CTE_Expression ")"
2214                             | "(" "Content-ID:" MIME_CID_Expression ")"
2215                             | "(" "Content-Description:" MIME_CD_Expression ")"
2216                             | "(" MIME_Additional_CF ")".
2217
2218 MIME_CT_Expression      = see RFC2045.
2219 MIME_CTE_Expression     = see RFC2045.
2220 MIME_CID_Expression     = see RFC2045.
2221 MIME_CD_Expression      = see RFC2045.
2222 MIME_Additional_CF      = see RFC2045.
```

2223 As shown here, the grammar is not complete. However, rather than duplicate the full syntax from RFC2045, and risk
2224 introducing errors or failing to keep track of changes in that specification, this document refers the reader to [Freed &
2225 Borenstein 96].

2226 Note that the MIME headers have been introduced in such a way that they do not alter the basic s-expression form of
2227 the ACL content expression. The MIME grammar presented here is a sub-grammar of the ACL s-expression grammar.

2228 **D.2 Example**

2229 The following example illustrates the use of MIME-style encoding of message content:

```
2230          (inform
2231              :sender translator
2232              :receiver agent01
2233              :content (translation
2234                          (English "File system full")
2235                          (Japanese ((MIME-Version: 1.0 (FIPA ACL Message))
2236                                     (Content-Type: Text/Plain; Charset=ISO-2022-JP)
2237                                     (Content-Transfer-Encoding: 7BIT)
2238                                     "<7 bit ISO 2022 Japanese text>"
2239                                    )
2240                          ))
2241              :ontology translation-service
2242              :in-reply-to request07)

2243
```