UNIVERSITY OF CALGARY

Establishing Conversations as the First-Order Agent Communication Mechanism

by

Daniel Bidulock

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2012

© Daniel Bidulock 2012

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-83561-6

Our file Notre référence
ISBN: 978-0-494-83561-6

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distrbute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protege cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Multi-agent systems (MAS) research has not produced consensus on how to best structure dialogues conducted between artificial agents. Even simple interactions require robust protocols to accommodate agent autonomy, provide fault tolerance, and signal the termination of dialogues. This research analyzes and demonstrates the feasibility of establishing conversations as the first-order mechanism of communication between artificial agents. It identifies recurring patterns of performative messages exchanged in existing protocols and builds a system in which these patterns are used alongside social commitments and policies to construct complete, coherent agent dialogues. The first-order conversational paradigm allows the agent designer to easily extend agent and dialogue functionality, as demonstrated by the simple Transaction Agents and various Auction Agents implemented here. As well, these conversation protocols naturally signal their own termination when all social commitments formed over their course have been fulfilled - a recognized problem within belief-desire-intention (BDI) and ad hoc agent development paradigms.

Acknowledgements

Thank you, Ray Aldred, Dr. Rick Love, and Dr. Charles Cook from Ambrose Seminary. Y'all have probably forgotten about me, but I haven't forgotten about y'all. Blessings upon you for prying open the door to this unorthodox ministerial sidestep.

Thank you, Dr. Robert Kremer for taking a chance on a slow learner and a spotty student.

And, thank you, Lyndsay, my love. All I've ever done I did to impress a pretty girl.

The LORD said, “If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other.”

Genesis 11:6-7 (NASB)

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The complexities involved in facilitating communication between artificial agents in multi-agent systems (MASs) are being addressed as research in the area progresses, but difficult challenges persist. One such challenge involves uncertainty in how to best establish connections between the individual performative messages agents exchange in order to form complete, cohesive dialogues (Amgoud and de Saint-Cyr, 2008). This research attempts to address this problem by providing context to these message exchanges. Toward this end, the feasibility of providing agents a first-order communicative mechanism with which to establish conversational bounds will be explored.

The agent conversations described hereafter consist of policies and states. Typically, when a message is exchanged between agents (whether sent or received) it passes through a conversation and the relevant policies contained therein are applied. Here, conversations are attributed a first-order status because they are the fundamental method by which individual performative messages are semantically organized within the context of coherent dialogues.

In establishing conversations as the first-order communicative mechanism through which artificial agents interact, observations made of existing communication protocols and standardized agent communication languages (ACLs) will be exploited. Specifically, recurring patterns of message exchanges have been identified in the implementations of several performatives defined by The Foundation for Intelligent Physical Agents (FIPA) ACL standard (FIPA, 1997). These patterns will be isolated and modularized as reusable components, which in turn will be used in the construction of first-order conversation protocols.

This chapter describes the research problem in Section 1.1, states the precise aim of this research in Section 1.2, outlines the steps taken to reach that aim in Section 1.3, and gives an overview of the remaining chapters in Section 1.4. A glossary of key terminology is provided at the end of this thesis.

1.1 Problem Statement

The performative messages exchanged between agents are seldom issued in isolation. Even simple interactions, as when one agent requests another agent to perform some action on its behalf, require several messages to be exchange in a predetermined sequence. The problem that arises is that there is little agreement as to how to establish the relationship between one message in this exchange and all the messages that preceded it (Amgoud and de Saint-Cyr, 2008).

When conducted in isolation, communicative interactions like the simple request just described pose few difficulties for the agent designer. In a MAS, however, it is necessary that agents be equipped to engage in a multitude of such interactions simultaneously. The importance of this ability becomes apparent when the successful completion of a task depends upon the communicative collaboration of more than two agents. Such group interactions may, for example, involve coordinating team actions, tendering competitive bids, or conducting an auction. It is in situations like these that it becomes especially important to establish a firm connection between the individual messages issued during these interactions.

Currently, one common method by which agents track and sort multiple interactions involve message filters and finite state machines. This method is employed by The Java Agent DEvelopment framework (JADE), which allows agents to filter incoming messages through a `MessageTemplate` and direct the appropriate course of action via instructions encompassed in a `Behaviour` class (Bellifemine et al., 2007). When a JADE agent receives a performative message, it examines it for certain characteristics[1]. If the incoming message meets the criteria set out by the `MessageTemplate`, the appropriate `Behaviour` will be called upon to process it. Typically, the `Behaviour` instructions are organized as a finite state machine (Bellifemine et al., 2007).

The message handling method employed by JADE gives rise to several problems. First, the `MessageTemplate` directs messages by evaluating them against a number of parameters. The number of parameters in need of consideration rises alongside the complexity of a conversation. As such, complex conversations quickly become unwieldy. Second, JADE agent behaviour, as directed by instructions contained in the `Behaviour` class, is not easily extensible. The protocols and agent behaviours that result from this arrangement are typically ad hoc. Finally, there is no built-in mechanism to determine if an agent interaction concluded successfully. A communication failure or mangled data may have compromised agents' interactions. In such situations the agent designer, minimally, is trusted to have employed a timeout mechanism to reset a `Behaviour`'s states.

A solution to the problems concerning agent message handling, code re-usability, and fault tolerance identified so far requires a discussion of some of the broader problems known in the field of MAS research. One related goal that has yet to be fully achieved is communication between open multi-agent systems (open MASs). Part of the reason this has not yet fully been achieved is because a satisfactory semantics has not been established (Guerin and Vasconcelos, 2007). ACLs define standard communicative performatives for use by agents in MASs, but are implemented in diverse ways across distinct agent societies (Dignum et al., 2007). The diversity in application often creates *regional dialects*, which cannot be understood between these societies (Singh, 1998). This problem, though not directly addressed by this research, does impact decisions made in

[1]Message ID, FIPA performative, ontology, etc. (Bellifemine et al., 2007).

addressing the problem identified here.

There are three main categories of semantics employed by ACLs: *mentalistic*, *protocol-based*, and *social* (Amgoud and de Saint-Cyr, 2008). Of these, social semantics is perceived to be the most viable, though it has not escaped criticism (Flores et al., 2007). Social semantics is concerned with agent interactions and the formation and fulfillment of *social commitments*, which obligate a debtor agent to perform some action for a creditor agent. These commitments usually form as the result of a communicative act, and may be fulfilled with subsequent acts, whether communicative or physical. A society's policies dictate when an act adds a commitment to or removes a commitment from an agent's collection of debts and credits. Although social semantics has proven to be the most popular option available, social commitments have been criticised for being impractical (Amgoud and de Saint-Cyr, 2008).

Determining the role social commitments play in connecting isolated messages containing ACL-defined performatives into a single coherent conversation is the practical matter of concern here (Amgoud and de Saint-Cyr, 2008). In a MAS, performatives are intended to have the same illocutionary impact as Austin and Searle's concept of speech acts (Austin, 1975; Searle, 1969). Here a message impacts agents by forming or fulfilling their social commitments. Established conversational protocols[2] contain performative messages that need to be exchanged in a predetermined sequence in order to meet the expectations of their design. Policy-directed social commitments have proven useful as the motivating factor for agents exchanging sequenced messages with one another (Flores et al., 2007).

Given the inherent autonomy of artificial agents and the unpredictable nature of open MASs (Huhns and Singh, 1998; Wooldridge, 1999), social commitment-driven conversational protocols need to account for a number of potential pitfalls, any one of which may

[2]E.g., the *protocol for proposals* (Flores et al., 2007)

require the premature termination of the conversation. This is where the established message sequences mentioned previously become valuable. Under ideal circumstances, of course, a conversation will only conclude once the purpose for which it commenced has been satisfied. In the event of a communication failure[3], however, a conversation should still terminate gracefully without leaving any unfulfilled social commitments. As such, potential message sequences need to be created in order to account for each of the events that may prevent a conversation from concluding successfully.

In order to construct a robust, failure-accommodating conversational protocol, the initiating message sequence will need to be connected to any subsequent message sequences to account for potential conversational turns, be it a successful conclusion or a failure of some sort. Social commitments will play a role in establishing this connection. As well, once message sequences are identified, it will be desirable to modularize them so that they may be reused in the construction of new conversational protocols. If this mode of conversation protocol design proves itself expedient, it may be desirable to conduct all agent communicative interaction within the context of modular conversations.

1.2 Project Aim

The aim of this research is to analyze and demonstrate the feasibility and value of conversations as the first-order mechanism of agent communication. It will identify recurring patterns of performatives in existing protocols and build a system in which these patterns are used in conjunction with social commitments and policies to easily design and build complete, coherent agent dialogues.

The Collaborative Agent System Architecture (CASA) (Kremer et al., 2004) will be employed to achieve the stated aim. CASA is a multi-agent system in which the com-

[3]Due to: network service interruption, mangled message contents, limited agent functionality, etc.

mon *request* and *propose* performatives have been implemented. Recurring patterns of performative messages have been recognized in the interactions between agents engaged in the protocols necessary for these implementations. The existence of such patterns suggests that elements of these dialogues may be used to construct new conversations for different purposes. For example, it may be shown that the steps taken to construct the *request* and *propose* conversations may be reapplied in the construction of the conversations necessary to implement other performatives, like *subscribe*. In order to meet the aim of this thesis, these patterns will be isolated and redeployed within the CASA framework as *conversationlets*.

As the *let* suffix is meant to suggest, a conversation*let* is not a conversation in and of itself. Conversationlets are used in the construction of conversations. Like a conversation, a conversationlet is comprised of policies that dictate the formation and fulfillment of social commitments, which, in turn, motivate the issuance of performative messages. Unlike a conversation, however, a conversationlet will often leave social commitments unfulfilled, or *hanging*. These hanging commitments connect one conversationlet to the next. A hanging commitment necessitates the issuance of further performative messages if the conversation as a whole is ever to terminate, which happens when no social commitments remain. Once all of the possible communicative acts in one conversationlet have been issued, the commitment it leaves hanging will direct the flow of a dialogue to the conversationlet with the performative messages necessary to fulfil the hanging commitment and motivate the dialogue further, if necessary. In this way, conversations, as the first-order communicative mechanisms, may be composed of reusable conversationlets.

An agent dialogue has taken place when one agent sends a meaningful message to another and the receiving agent, in turn, issues a meaningful response to the dialogue's initiator (Flores and Kremer, 2002). *Meaningful* messages are those with semantics known by the interacting agents. A message's meaning is determined primarily by the performative it contains. Performatives are locutionary actions defined by an agent society's adopted ACL. As stated earlier, CASA has established protocols to carry out the common *request* and *propose* performatives. Without such protocols, for example, it would be difficult for an agent to issue a *request* message to another and be assured of the successful completion of the requested action without back-and-forth communication. Performative messages issued in isolation can accomplish little more than simply conveying information, and then only if the intended audience is receptive. If any meaningful interaction has taken place between agents, it is likely that it was accomplished through a dialogue. It is for this reason that conversations, as the motivators of performative message exchanges, are to be attributed first-order status in agent communicative interaction.

The notion that social commitments may be used to motivate and attribute meaning to a conversation is already established (Flores et al., 2007). In a conversational context, social commitments are formed and fulfilled when agents exchange messages. When one agent sends or receives a message, the performative contained therein is examined in light of the society's policies and the agents' conversational context, which determine if a social commitment needs to be formed or fulfilled. If a commitment is formed, a debtor agent is obliged to perform an action for a creditor agent. This action may be communicative and will remain until the debtor agent fulfills its commitment by issuing a message containing an appropriate performative. The creation and fulfillment of social commitments resulting from message exchanges motivates agents to perform the communicative actions that comprise conversations.

An agent conversation concludes when all of the social commitments formed over the course of the conversation are fulfilled. This supposition "gives us a precious (this is a well known problem in the field) stopping condition" (Pasquier and Chaib-draa, 2005). The problem of conversational *stopping conditions*[4] is often decided by an agent society's policies (Kinny, 2001; McBurney and Parsons, 2003) or other creative means[5]. Here the role of policies is to determine whether a performative creates or fulfils a social commitment. Social commitments, meanwhile, motivate agent conversations by requiring fulfillment with the issuance of appropriate policy-determined performative messages. When the conversation's participants have fulfilled all of their social commitments, the conversation ends.

The idea that conversations terminate when all social commitments are fulfilled gives rise to a potential point of contention. It could be argued that the conversations necessary to, for example, enter into a business contract, subscribe to a magazine, or enter into marriage leave unfulfilled commitments hanging once their associated conversations conclude. This concern, however, is mitigated if it is understood that the respective commitments formed are done so within the context of an ongoing conversation. Consider marriage: although a commitment has been formed as the result of a conversation, it is understood that the marriage conversation continues until the persistent commitment ends in death or divorce, whereupon that commitment is fulfilled or violated, as the case may be. Alternatively, in the interests of mitigating potential points of contention, a special status could potentially be attributed to *persistent* social commitments. The existence of these commitments would not be factored into the accounting of conversation termination. This would forgive any agent implementations that might rather delete persistent-commitment-forming conversations than to store them indefinitely. It is understood, for the purposes of this research, that the former qualification is preferred to the latter; that persistent commitments exist within the context of an ongoing conversation, which exists only so long as the interested parties are willing and able to live up to

[4]Or how to know when a conversation has ended.

[5]As with Pasquier, et al's application of economic utility in motivating agent dialogues. An agent decides to engage, or continue, in a conversation if it believes it will be useful in reducing incoherence about some aspect of its environment (Pasquier and Chaib-draa, 2005).

their respective obligations.

1.3 Objectives

In order to successfully achieve the aim stated in the previous section, the following objectives must be met:

O1: Survey the Literature. This project draws inspiration and direction from existing research and literature. A survey of works concerning speech acts, conversations, and social commitments will be undertaken. This will serve to ground this research and to identify the key issues that it will address. Investigations into the application of social commitments in multi-agent systems are of special interest here.

O2: Analyze the Requirements. Several criteria must first be met before the efficacy of the communication methods resulting from this research can be demonstrated. These requirements include an analysis of the existing software upon which the system will be built, which must provide base agent functionality and be extensible so as to allow the changes necessary to accommodate the proposed first-order conversational mechanism. As well, it will be determined how the modified system is be to evaluated in order to determine if it meets the expectations set forth by the stated aim.

O3: Design the System. The successful completion of this objective depends on being able to correctly deconstruct the established conversations named previously: i.e., patterns in agent interaction common to CASA's *propose* and *request* conversations will be identified. These patterns outline the messages issued, the creation and fulfillment of social commitments, and internal considerations made by agents over the course of their respective dialogues. Having identified the recurring conversational patterns, it will be determined how these repackaged conversationlets will be introduced into CASA and how they might be used to create new conversations, like *subscribe*.

O4: Implement the System. Introduce the first-order conversational functionality into CASA. Conversationlets will take their place alongside policy-formed social commitments as the constituent components of the *propose* and *request* conversation. As well, a *subscribe* conversation will be implemented using these conversationlets wherever possible. A detailed description of their brick-and-mortar relationship and introduction into CASA will be provided.

O5: Analyze the System. In addition to the conversations necessary to form a MAS, the conversations necessary to carry out four different types of auctions using conversationlets will be constructed in order to demonstrate the efficacy of the system implemented in pursuit of achieving the stated aim of this research. The auction systems to be constructed include: English, Dutch, Sealed Bid, and Vickrey. Each auction-type consists of a number of distinct conversations, which work in conjunction to achieve a single goal. Differences and similarities between these conversations will be identified and conversationlets will be reused in their construction wherever appropriate.

1.4 Thesis Outline

This thesis is organized around the fulfillment of the objectives that lead to its stated aim. Chapter 2 provides a survey of related literature and research. Chapter 3 determines what expectations need to be met in order to implement and evaluate the system to be constructed. Chapter 4 describes the design and implementation of the new conversational framework. Chapter 5 provides an analysis of the new system and measures the results produced against the stated aim of this research. Chapter 6 concludes this thesis and identifies potential avenues of future investigation.

1.5 Summary

This research explores the viability of establishing conversations as the first-order mechanism through which agents communicate. It has been recognized that relationships must be defined between the individual performative messages exchanged during agents' communicative interaction. Current agent message-handling practices are insufficient, as they require message filtering mechanisms that limit agents' communicative accessibility, employ inextensible ad hoc state machines to direct agent behaviour, and typically have no built-in fault tolerance. It is believed that the first-order conversational mechanism will help resolve these issues.

In identifying the problems addressed by this research, related problems concerning ACL semantics were necessarily incorporated into the discussion. Specifically, the mentalistic semantics expected in the implementation of the FIPA ACL is not well-suited for communication in heterogeneous MASs (Dignum et al., 2007). Social semantics has been selected as the preferred alternative to the mentalistic variety. Here, the role played by social semantics' fundamental component, the social commitment, will be refined and applied in such a way as to produce modular, reusable conversationlets, which will in turn be established as that which comprises agent conversations. Conversationlets will be composed of performative patterns previously identified in CASA's established *propose* and *request* protocols. As well, a *subscribe* conversation will be implemented using the newly identified conversationlets wherever possible. Using these conversationlets, the conversations necessary to conduct several auction types will be constructed. Social commitments will serve as both the binding and driving force behind the conversational mode of agent interaction. Applied in this way, the problems associated with communication in MASs and current methods of agent method-handling, including problems associated with identifying conversation stopping conditions, will be mitigated.

Chapter 2

Literature Survey

A great deal of admirable effort has been poured into solving the existing problems in modern agent communication. ACLs have been standardized and semantics have been devised, but flexible societal interoperability[1] remains an elusive goal. Much of the associated difficulty revolves around the uncertainty of how to establish connections between isolated messages containing performatives so that they may form cohesive conversations (Amgoud and de Saint-Cyr, 2008). This research awards primacy to agent conversations, which consist of policies and states, and provide the first point of interpretation for most messages, whether sent or received. It demonstrates that performative messages are seldom issued in isolation, and inter-agent communication is made more manageable when message exchanges are semantically organized within the context of a conversation.

Recurring sequences of performative messages have been observed across some agent communication protocols. These sequences are to be modularized and reused in the construction of new protocols. This promise of reusability raises the question of how to connect these so-called *conversationlets* in such a way as to accommodate potential turns in agent dialogues. For example, a conversation between two agents may conclude successfully, in accordance with the purpose for which it was designed, or it may fail due to communication errors or some other factor. Conversationlets may be devised to account for each of these factors and then reused across distinct communication protocols.

Social semantics is perceived to be the most viable solution to many of the problems known in the field of multi-agent communication (Flores et al., 2007). Though the social semantic variety is a popular area of research, the precise role of its central component,

[1]I.e., the ability to communicate between distinct, heterogeneous MASs.

the social commitment, has yet to be established (Amgoud and de Saint-Cyr, 2008). Here, the formation and fulfillment of social commitments is to be governed by a MAS's policies. A message exchanged between agents is compared against these policies and a social commitment may form or be fulfilled as a result. Further, this thesis assigns social commitments the task of connecting converationlets in such a way as to accommodate possible conversational turns. In this way, policy-driven social commitments not only bind modular converationlets to form a complete conversation, they also motivate agent message exchanges until all have been fulfilled, whereupon the conversation terminates.

This chapter surveys the literature that motivates this research. Several broad topics related to agent communication will be examined in order to establish a foundational understanding of the relevant issues. These include concepts concerning *agency* in Section 2.1, agent communication language semantics in Section 2.2, and agent conversations in Section 2.3. As well, a brief overview of auction theory will be provided in Section 2.4, as several auctions simulations will be implemented to demonstrate the value of the first-order conversational mode under investigation.

2.1 Agency

2.1.1 Artificial Agents

In the context of computer science, the concept of the *agent* remains somewhat nebulous. As researchers in the field are fond of pointing out, a single satisfactory definition of agency has yet to emerge (Huhns and Singh, 1998; Wooldridge and Jennings, 1994). The difficulty in defining what it means to be an agent stems from the diversity of agent applications. Simply put: different areas of application place different emphasis on the qualities their agents must possess.

There is little agreement to be found in even the most tempered definitions of agency.

For example, Huhns and Singh say that "agents are active, persistent (software) components that perceive, reason, act and communicate" (Huhns and Singh, 1998). Wooldridge, on the other hand, describes an agent as "a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives" (Wooldridge, 1999). It is true that much of what is made explicit in the former definition is embodied in the latter. It may even be argued that notions of agent autonomy, though not specifically mentioned, can be inferred in the former. There are, however, a couple notable differences: Wooldridge's definition makes no mention of *persistence*, nor is an agent's ability to *reason* made explicit. These are non-superficial disagreements, as Wooldridge could rightly question the need for every agent to recall previous executions, or to respond to its environment in anything but a reactionary manner. For the purposes of this thesis, these differences are identified for no other reason than to illustrate the lack of consensus surrounding the definition of *agency*.

Despite the ongoing disagreement on what precise characteristics comprise an agent, some general expectations must be outlined. In this thesis, the *social* aspects of agency will be emphasized. That is, agents must be able to communicate with other agents. It is from this characteristic that the other, perhaps subordinate characteristics, find their purpose. These characteristics include autonomy and the ability to effect changes appropriate to the agent's design objectives and the consequences of agent interactions. As such, an agent will be defined hereafter as "a software component capable of communicating with similarly equipped components and acting within the bounds set by its design objectives and society's rules of *proper* conversation".

If autonomy cannot be inferred from the definition provided, it is to be assumed. As well, *propriety* is to be understood as the belief that agents act in good faith. That is, agents do not purposefully or maliciously break the social commitments formed over the course of their conversations (Stergiou and Arys, 2003). This expectation is in line with what Huhns and Singh call *social autonomy* (Huhns and Singh, 1998). How societies deal with antisocial agents is beyond the scope of this thesis.

2.1.2 Multi-Agent Systems

A discussion of the *social* aspects of agency and communication cannot take place if agents operate in isolation. Like the general assumption of autonomy as a key characteristic of agency (Wooldridge, 2001), it is hardly worth mentioning that communicative agents are not particularly useful unless they can establish contact with other communicative agents. How this contact is established and conversations are carried out, however, is determined in part[2] by the multi-agent system.

The definition of a *multi-agent system* seems a far less contentious issue than the definition of *agency.* The one provided by Wooldridge is sufficient. He says "a multi-agent system is one that consists of a number of agents, which *interact* with one another, typically by exchanging messages through some computer network infrastructure" (Wooldridge, 2001). This definition is narrow enough to emphasize the required social aspects of agency, but broad enough that potential systems will not be excluded because of the environment they provide to their agents or their receptivity to external systems (i.e., *open* versus *closed* systems). Further, this definition does not restrict the behaviour of the agents that occupy it, be they cooperative, competitive, or even antagonistic.

2.1.3 Environment

It can be appropriate to say that the MAS determines the *environment* in which its agents operate (Ferber, 1999), but this is not always true. Both agents and MASs operate within a subset of the physical environment, be it an artificial environment implemented on a computer or as with a robotic agent exploring the surface of Mars. It is, perhaps,

[2]The *environment* in which agents and MASs are situated also determines the mode of communication

because of the near-infinite number of physical environmental subsets existing in this universe that Norvig and Russell find it practical to define *environment* from an agent's perspective. That is, an environment is to be understood in accordance with how the occupying agents perceive it (Russell and Norvig, 2002).

From the work conducted by Norvig and Russell, Huhns and Singh distill six environmental properties to be considered from the agents' perspective (Huhns and Singh, 1998). These properties are shown in Table 2.1.

Property	Definition
Knowable	To what extent is the environment known to the agent?
Predictable	To what extent can it be predicted by the agent?
Controllable	To what extent can the agent modify the environment?
Historical	Do future states depend on the entire history, or only the current state?
Teleological	Are parts of it purposeful (i.e., are there other agents?)
Real time	Can the environment change while the agent is deliberating?

Table 2.1: Environment-Agent Characteristics (Huhns and Singh, 1998)

The properties listed in Table 2.1 are helpful in terms of both *understanding* and *modelling* various environments. *Understanding* an agent's environment is particularly important when the environment cannot be constrained, as with the example of the Mars explorer robot. *Modelling* an environment, on the other hand, affords the agent designer the luxury of creating an environment suitable to his agents and vice versa. Either way, the environment is instrumental in determining how an individual agent or an MAS should behave and communicate, both internally and externally.[3]

2.1.4 Actions

At a primitive level, an *action* may be understood in terms of *temporal projections*. "In a temporal projection problem, we are given a description of the initial state of the world,

[3]The distinction between the modelled and unconstrained environments should not be understood as being the same as the distinction between artificial and the natural physical environment, as even artificial environments can be beyond control (e.g., the Internet) and natural environments can be constrained (e.g., the zoo).

and we use properties of actions to determine what the world will look like after a series of actions is performed" (Gelfond and Lifschitz, 1993). With *temporal projections* in mind, it is tempting to simply define agent *actions* in terms of their impact on an MAS or environment and leave it at that. In fact, for the purposes of this research, understanding *actions* in this way would almost suffice, but to truly appreciate the complexities inherent to an MAS, it is worthwhile to elaborate more on the concept of agent *actions*, which Ferber describes as *deceptively elementary*.

For Ferber an action is both a *modification* and a *gesture*. Conceptual difficulties arise in part because an action is often treated as something that *modifies* the world from one stable state to the next, as with the notion of *temporal projections* (Gelfond and Lifschitz, 1993). This clearly cannot be the case, because the universe is in constant motion whether or not agents are acting upon it. Indeed, modifications do occur, but an action is also an *attempt* to exert influence on the world; "to modify it in accordance with our desires" (Ferber, 1999). Again, difficulty lies in the fact that the consequences of a gesture might not achieve the desired modification. Moreover, the same gesture may produce different modifications in spite of an agent's intent. So, in order to fully appreciate the concept of an agent action, "we must make a distinction between the gesture and the consequences of the gesture, between the act and the universe's response to the act" (Ferber, 1999).

The CASA multi-agent framework employed for the purposes of this research organizes the actions an agent may perform in a lattice structure, which defines the agent's ontology (see Figure 2.1). This has proven especially useful in terms of narrowing the semantic range of the performatives defined by the popular FIPA ACL (Kremer et al., 2004). The ontology defines all the actions (including performatives) available to an agent and how each of those actions relates to the others declared in the ontology (Kremer and Flores, 2005).

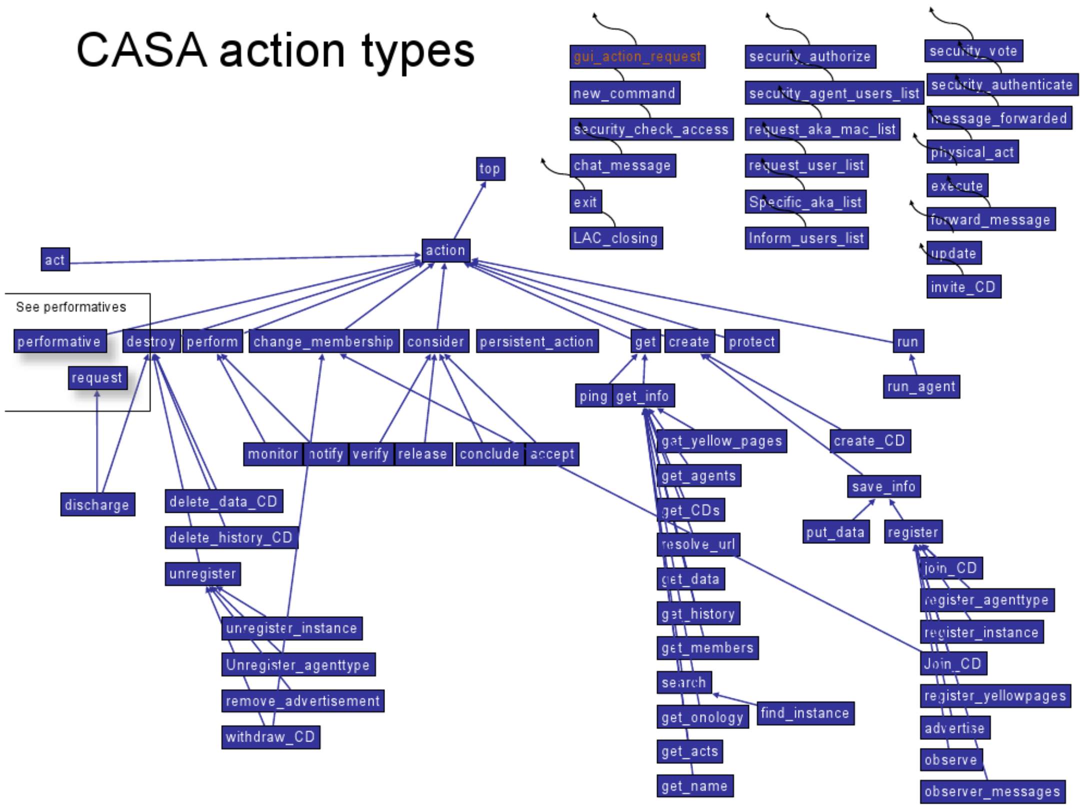


Figure 2.1: The CASA action hierarchy.

Ontology

As with most of the topics discussed so far, the concept of the *ontology* is nuanced according to the needs of the particular agent designer or researcher. It can be broadly defined as a "formal definition of a body of knowledge... Essentially a taxonomy of class and subclass relations coupled with definitions of the relationships between these things" (Wooldridge, 2001). Though this definition may be appropriate for some MAS applications, it allows more leeway than what is necessary - or desired - for the purposes of this research.

Fornara, et al, apply the concept of the ontology toward the goal of achieving interoperability between heterogeneous MASs, which is directly related to this research (Fornara et al., 2008). Her ontology's purpose is in defining the actions and artifacts appropriate to an artificial *institution*. Here the *institution* is analogous to human institutions like *property ownership* and *marriage*. They are distinct, though may potentially merge as in the case where a husband and wife cannot agree on whether to purchase a new car. Fornara's ontological framework outlines the entities that exist, the actions and events that may take place, and the roles agents may play within the institution.

Fornara's institutional paradigm is useful in emphasizing the point that certain actions are only appropriate[4] within the social norms established by an institution (Fornara et al., 2008)[5]. However, this institutional ontology offers a layer of abstraction that this research may render redundant or unnecessary. One of the objectives of this research is to implement agents that can participate in an auction, which is an interaction that could potentially fit into Fornara's formalized institutional framework. In this research, however, the actions that take place are dictated by first-order conversations. These actions, be they physical or communicative, are defined in the CASA ontology, thus allowing any agent with access to the ontology the potential ability to operate within the institution or participate in a conversation, as the case may be. As such, the CASA ontology, as used in conjunction with appropriate conversations, may supersede the need for a formalized institutional framework.

Performatives

A CASA *performative* is a type of *action*, as described in Section 2.1.4. Performatives are communicative in nature and take their place alongside *physical* actions in an agent's ontology. And, like physical actions, performatives are *gestures* made in an attempt to *modify* an agent's environment. This treatment of communicative acts as physical actions has been established in the works of Austin (Austin, 1975) and Searle (Searle, 1969).

[4]Or even sensible

[5]After all, what is an *institution*, if not a special place to conduct specialized conversations. See Section 2.3.2

As per Austin and Searle's work, a communicative act is much more than a simple exchange of information. These *speech acts* can be understood at *locutionary*, *illocutionary*, and *perlocutionary* levels. For Austin, a *performative* is a special kind of speech act in which the speaker, by uttering the locution, performs the action described therein (Austin, 1975). For example, by shouting "watch out for that car!" or by saying "will you help me move this weekend?" the speaker performs a *warning* and *request*, respectively. There is an action inherent in such sentences, which is performed as soon as the sentence is spoken out loud.

John Searle enhanced Austin's work in many ways, but here his most important contribution may be in his refinement of the distinctions between different kinds of speech acts at the illocutionary level. For him, Austin's five categories were insufficient, so he identified five of his own: assertives, directives, commissives, expressives, and declarations (Searle, 1975) (Kibble, 2006). Elements of this taxonomy are evident in a CASA agent's performative hierarchy, which is established as a branch of its standard *ontology* (Kremer and Flores, 2005) (see Figure 2.2).

CASA employs 34 performatives in total. Of those, 22 are part of the FIPA standard. Taken on the whole, each of the 34 performatives fall into one of three of Searle's broad categories: assertive, directive, and commissive (Searle, 1975). To put it another way, each of the available performatives may be considered to have accomplished at least one of three things when its container message is delivered from one agent to another:

1. *Asserts* the sending agent's *attitude* toward the truthfulness of the information being exchanged, as with the *inform*, *confirm*, and *disconfirm* performatives.

2. *Directs* the receiving agent to perform some action, as with a *petition*.

3. *Commits* an agent to some course of action, as with the obligation to *ack*nowledge or *reply* to another agent's attempt at communication.

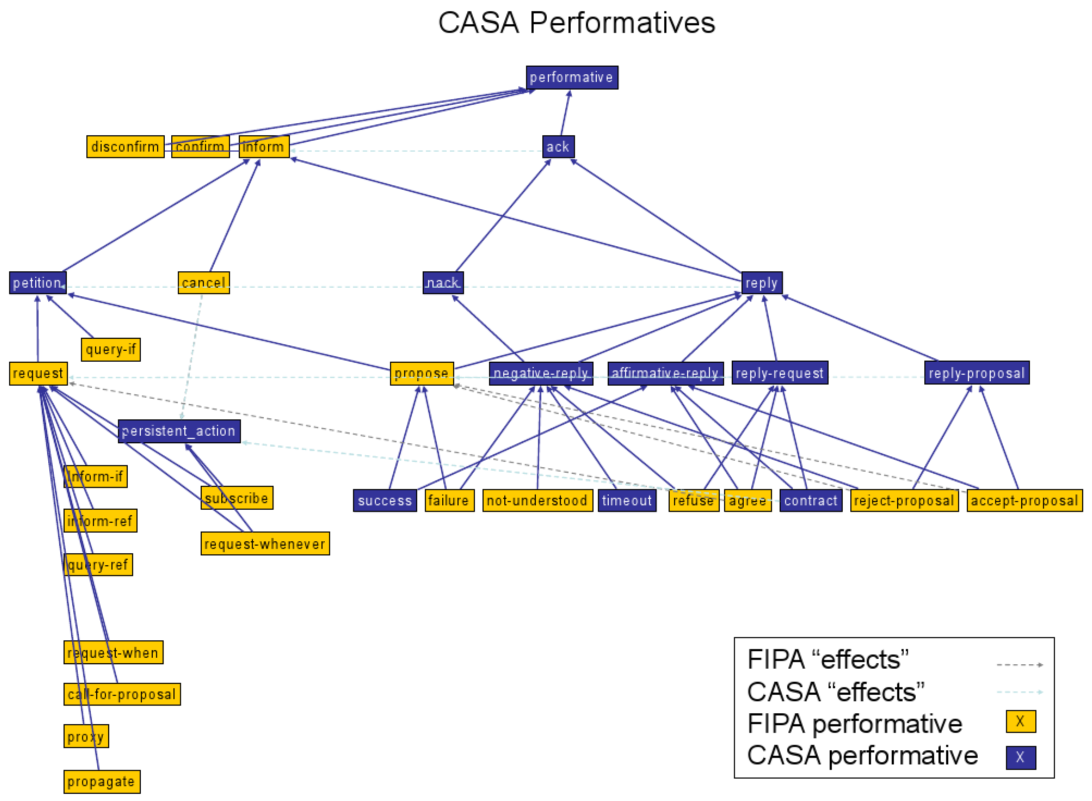


Figure 2.2: The CASA performative hierarchy.

These categories are not mutually exclusive. In fact, it is true that every act of communication in CASA *commits* the receiving agent to fulfill the social commitments that inevitably form. Likewise, as per Figure 2.2, every request made by an agent *asserts* that agent's attitude toward the consequences of that request[6]. That is, the requesting agent truly wants the receiving agent to perform the requested action. Further: should the receiving agent determine that it is able to perform the requested action, its *reply* to the requesting agent will *commit* it to the course of action that will see the request fulfilled. As such, the three relevant categories in Searle's taxonomy of illocutionary types necessarily overlap.

[6]As per, Searle's assertive, directive, and commissive illocutionary categories (Searle, 1975)

FIPA's ACL (FIPA, 1997), which is employed by CASA's communicating agents, depends conceptually upon Austin's notion of performative speech acts (Pitt and Mamdani, 1999). The illocutionary impact of all well-formed FIPA messages is determined by the performative specified therein. This, in turn, determines the perlocutionary framework with which the receiving agent's internal considerations are made. For example, if one agent *proposes* that it perform a service, the receiving agent will recognize the message as a proposal and will then consider whether it wants the proposed service to be performed.

2.2 Agent Communication Language Semantics

The syntax of the available ACLs is well-established, but the meaning of the locutions specified therein is not, or, if defined, is not sufficiently flexible. This situation has given rise to the problem of regional dialects, which are incomprehensible between agent societies (Singh, 1998). There are two noteworthy standardized agent communication languages: KQML (Finin et al., 1992) and FIPA (FIPA, 1997). The formal semantics of both these ACLs are *mentalistic* (Alagar and Zheng, 2005), as opposed to *social*. The characteristics of each semantic variety will be outlined in this section, though the bulk of the discussion will focus on social semantics, which plays an integral role in this research.

2.2.1 Mentalistic Semantics

Though the *mentalistic* semantic variety can be considered a constituent component of the FIPA ACL specification, the FIPA contributors themselves immediately recognized the difficulties inherent in such a scheme (FIPA, 1997). Mentalistic semantics require that communication be directed by an agent's internal state; its so-called *beliefs*, *desires*, and *intentions* (BDI). The problem with BDI is that an agent's internal workings cannot be verified by simply making external observations. As such, mentalistic semantics "cannot be considered normative until the issue of compliance testing is resolved" (FIPA, 1997).

The mentalistic BDI semantics is most effective in cooperative, homogeneous societies. Should it be desirable for heterogeneous agents to communicate between societies, however, cooperation and homogeneity cannot be assumed. Moreover, homogeneity gives rise to practices peculiar to a given agent society, which in turn, gives rise to *regional dialects* (Singh, 1998). Two societies may speak the same ACL, but each society's dialect is incomprehensible to the other. Assuming that communication between heterogeneous agents is desirable, mentalistic semantics alone is untenable.

2.2.2 Social Semantics

It was Singh who first proposed that agent societies adopt the *public* character of human communication in order to mitigate the problems associated with mentalistic semantics (Singh, 1998). With this proposal came the idea of the *social commitment.* An agent, as a member or guest of a particular society, forms and fulfills *social commitments* while conversing with agents in other roles. As such, Singh's socially conscious agents treat protocols as sets of commitments rather than as finite state machines (Singh, 2000). Thus, assumptions about agent conversations can be made through external observations and some of the communicative flexibility naturally enjoyed by humans is extended to artificial agents.

It is important to emphasize a social commitment's *external* and *observable* nature. Consider Amgoud and de Saint-Cyr's definition: "a *commitment* is a directed obligation from one agent, called the *debtor*, to another, called *creditor*, about the truth of a given fact or to perform certain actions in the future" (Amgoud and de Saint-Cyr, 2008). This is unlike mentalistic semantics, which are concerned with an agent's internal state and are not verifiable through external observation. A *social* commitment, in contrast, focuses exclusively on an agent's externally observable actions during conversation.

Since Singh first proposed social semantics as a solution to the problems plaguing its mentalistic cousin, the expectations of its *social commitment* constituent have evolved and been refined. With this evolution came the realization that social commitments need to be tracked and stored, and that an agent's role in a conversation impacts the policies that dictate their creation and fulfillment.

Commitment Stores

CASA treats social commitments as described above, and employs another feature described by Amgoud and de Saint-Cyr as a "commitment store, which holds the commitments of the players during the dialogue" (Amgoud and de Saint-Cyr, 2008). That is, each agent *tracks* and *stores* the social commitments that form over the course of all of its conversations. CASA, however, is only concerned with *observable* commitments. For a commitment to be observable, an agent eavesdropping on the conversation must be able to determine what social commitments were formed as the result of any given message exchanged over its course. Interestingly, Amgoud and de Saint-Cyr suggest that the commitment store itself be made externally observable, which potentially reintroduces the problems inherent in mentalistic semantics. CASA's commitment store, on the other hand, is not externally observable, because the observability of the social commitments formed therein make it unnecessary.

Agent Roles

An agent's *role* in any given context is determined by how it is expected to participate and interact within that context. For example, an agent may play a *role* in a society or institution (Guerin and Vasconcelos, 2007). In the auction setting, an agent may be an auctioneer or bidder. Similarly, an agent plays a role within a conversation. Again, the conversation participants may be an auctioneer or bidders. It is difficult to discuss an agent's role apart from a conversation or society.

With respect to social commitments, an agent will take on the role of *debtor* and *creditor*. These roles are not static, and an agent may take on both roles simultaneously at any given moment. Agents create and fulfil social commitments with almost every performative message they exchange. When an agent takes on the debtor role, it is obliged to perform some action on behalf of the corresponding creditor agent. Once that action has been performed and the social commitment fulfilled, both the creditor and the debtor agents release their respective roles. Often it is the case that an debtor agent will become the creditor by performing the action that it was obliged to perform. Then, the agent that previously held the creditor role, will be indebted to the new creditor agent as a result of it fulfilling its social commitment.

With respect to conversations, an agent may take on, for example, a *client* or *server* role. These roles remain static over the course of a conversation, wherein a server agent performs some action on behalf of a client agent. The roles do not, however, necessarily remain static between conversations. Once a conversation has ended and the server agent has performed some action on behalf of the client, a new conversation may take place wherein the previous roles are reversed. The conversation that follows may even serve the same purpose as the previous conversation. For example, there is no reason why an agent previously bidding (as a client) in an auction could not start a new auction and serve as the auctioneer (server).

In summary, a conversation typically takes place between a client agent and a server agent. It does not have to be the case that an agent's client or server role remain static from conversation to conversation, but it will remain static relative to the conversation partner's role over the course of a conversation. In contrast, the creditor and debtor roles do not exist outside of a conversation, and each agent may be indebted to the other any number of times over the course of the conversation.

Policies

Policies are not exclusive to the realm of social semantics, as they were first employed by researchers interested in traditional mentalistic approaches. Greaves, Holmback, and Bradshaw saw the adoption of conversation policies as a solution to what they have termed *The Basic Problem* (Greaves et al., 2000). That is, sufficiently expressive ACLs reveal nothing about the internal state of an agent issuing a locution. A receiving agent needs to have some notion of another's goals if it is to issue an appropriate response. Conversation policies constrain the potentially unbounded expressive powers of an ACL, thereby reducing the number of factors an agent needs to consider when formulating an intelligent response.

In the realm of social semantics, policies only constrain the expressive power of an ACL indirectly. Though Parsons, McBurney, and Wooldridge were interested in mentalistic semantics, their understanding of conversation policies is close to that of CASA and its implementation of *conversationlets* as described in this thesis. They say that conversation policies are "rules about short sequences of locutions which assemble sections of an overall conversation between agents" (Parsons et al., 2003). Conversationlets are, in fact, *short sequences of locutions*, which are called upon when needed to fulfill a social commitment. Here, policies may determine which conversationlet is needed to fulfil a commitment or when to add or remove one from an agent's commitment store.

2.3 Agent Conversations

In the field of multi-agent systems the conversational taxonomy established by Walton and Krabbe is a foundational point of reference. They identify *at least* six types of dialogue games: *persuasion*, *negotiation*, *inquiry*, *deliberation*, *information seeking*, and *eristic* (Walton and Krabbe, 1995). Walton and Krabbe make no claim that this is a comprehensive list. And, to clarify what might not be obvious in this brief summary, the difference between an inquiry and information-seeking conversation is the difference between general and personal ignorance, respectively. Multi-agent researchers typically take interest in all but the eristic conversational variety.

Walton and Krabbe's dialogue games model communicative interaction as a game in which the dialogue participants are players Walton and Krabbe (1995). When a player takes his turn in the game he makes a move, which consists of the issuance a legal locution. The next player takes her turn, whereupon she issues a legal locution in the light of the locution that was issued in the preceding turn, and so forth. As such, dialogue games are defined in terms of a collection of locutions and rules that govern their use (Mcburney and Parsons, 2001).

The agent conversations described in this research best fit into Walton and Krabbe's *information-seeking* category. This attempt at categorization is unsatisfactory, but still useful for illustrative purposes. Parsons, Wooldridge, and Amgoud have gained some interesting insights into the taxonomy as a result of their work concerning information seeking, inquiry, and persuasion conversations (Parsons et al., 2002). These researchers identify a common thread running through the three protocols they designed for these conversations. The information seeking protocol is foundational. As such, they describe the inquiry protocol as a series of implicit information seeking protocols. The persuasion protocol, in turn, shares features in common with the inquiry protocol. A similar observation was made of certain CASA protocols (i.e., *request* and *propose*). It was precisely these similarities that gave rise to the *conversationlet*, which could potentially become the building block of every conversation, irrespective of its category.

2.3.1 Conversation Purpose

Since this research is not concerned with any particular conversation *type*[7], it is better to turn attention to a conversation's *purpose*. The idea that the structure of a conversation is determined by the respective goals of the conversation partners is not a new one. The earliest computer science-specific literature surveyed here is credited to Grosz, who studied the interaction in a human master-apprentice conversation to identify which conversational elements are amenable to formalization and computation (Grosz, 1974). In doing so, she hoped to advance understanding of natural language processing and speech recognition. The interaction between the two human participants was purposefully constrained so as to best simulate the capabilities of the computer hardware available at the time. She discovered that the apprentice's feedback to the master, or expert, went well beyond simple reports of success and failure in following instructions. Explanations had to be offered, progress reported, and a common vocabulary established. Communication in this context was found to be highly interactive and the conversational structures produced were found to correspond to the structure of the task being carried out.

Grosz and Sidner attempted to formalize conversational structure by dividing it into three distinct, but related components (Grosz and Sidner, 1986). The first of these, the *linguistic* structure, consists of *segments*, which in turn consist of the conversation's actual utterances. The second component, the *intentional* structure, is grounded in two concepts: the overall *purpose* and the *segment purposes*. Each segment purpose, which may contain other segment purposes, is intended to contribute to the overall purpose of the conversation. The final component, the *attentional* state, represents the conversation participants' changing foci as the discourse unfolds. Taken together, these three components aid the conversation's participants in determining why an utterance was made and the meaning behind that utterance.

[7]As per Walton and Krabbe's typology (Walton and Krabbe, 1995).

The desire to establish a formalized conversational structure has persisted to this day. Reed was among the first to formally abstract five of Walton and Krabbe's six classes of human dialogue (Reed, 1998). The *eristic* dialogue is omitted. The others are modeled as three-tuples, which include the *dialogue type*, *topic*, and the *sequence of utterances*. This type of modeling is typical of research conducted in the field, though the language used to describe the various conversational mechanisms varies.

Building upon the work of Reed and others, McBurney and Parsons produced an agent dialogue formalism built upon components common to what they call the *Agent Dialogue Framework* (McBurney and Parsons, 2002). This consists of three layers: dialogue *topics* occupy the lowest layer, rule-governed *dialogues* occupy the second, and top-layer *controls* dictate what dialogue is appropriate at any given time. The rules governing dialogues include: commencement rules, locutions, combination rules, commitments, and termination rules. This conversational mechanism is fairly typical of those produced by other researchers, of which there is no shortage.

It would likely not be difficult to shoehorn CASA's conversationlet-commitment strategy into McBurney and Parsons' conversational formalism[8], but its greatest contribution is in how it mitigates problems associated with *termination rules*. Many such formalisms name these rules as an important point of consideration, including: Maudet and Chaib-Draa (2002); Karacapilidis and Moraitis (2004); Chaib-Draa et al. (2006); Parsons et al. (2002); McBurney and Parsons (2003), amongst many others. Whereas those named have decided that a conversation's termination occurs when inbuilt criteria are met, a CASA conversation terminates when there are no more social commitments left to be fulfilled.

[8]Or any of the others currently undergoing development

2.3.2 Conversation Context

How agent *institutions* impact social semantics is a topic worth mentioning because interest in this area of research seems to be gaining momentum (Guerin and Vasconcelos, 2007; Fornara et al., 2008; Verdicchio and Colombetti, 2009). According to Searle, "an institution is any collectively accepted system of rules (procedures, practices) that enable us to create institutional facts" (Searle, 2005). An *institutional fact* is one that cannot be true beyond the existence of a human institution. Using Searle's example, it could not be true that he owns stock in AT&T if it were not for its existence as a corporate entity[9]. It is, perhaps, Searle's preliminary observations about the symbiotic partnership between languages and institutions that are most relevant to this research: simply put, it may be the case that one cannot exist without the other. That is, an institution may not exist without the language to describe it and the existence of a language implies the existence of an institution.

With Searle's understanding of institutions, it comes as no surprise that Verdicchio and Colombetti would propose that ACLs be based on *institutional actions* (Verdicchio and Colombetti, 2009). That is, performative acts are defined and governed by the context in which they are uttered. These utterances form and manipulate *social commitments*, which are external and observable. Semantics are defined at the institutional level, so they propose that ACLs be designed to facilitate the manipulation of social commitments in any given context.

Verdicchio and Colombetti's emphasis on the role of the institution in determining meaning and governing discourse is noteworthy. Strangely, however, they criticize FIPA for failing to provide any mechanism with which to enforce commitments made by agents during the course of a conversation. It seems reasonable to expect the authors to assume that the responsibility of enforcement and sanctions would naturally fall to the

[9]Or for the existence of institutions protecting private property ownership, for that matter

institution, rather than being built into the ACL.

If Searle's conjectures about institutions come to be accepted as true, future discussions of CASA will be shaped by the implications. At its core, CASA provides the institutional framework with which the procedures and practices governing social commitments are enforced. And, social commitments comprise the *facts* at the heart of Searle's institution. If an institution were to be defined within the wider CASA institution, its only distinguishing feature would be the conversations that take place within its *walls*. That is, the CASA institution (or MAS) is defined by the conversations conducted therein.

2.4 Auction Theory

The overview of auction theory to follow is only concerned with auctions as a means to facilitate economic transactions. Any discussion of auctions as used in the testing of various economic theories is beyond the scope of this research. Here, simulated auctions are to be implemented in order to demonstrate the easy re-usability of conversationlets in the construction of conversation protocols, and the viability of the first-order mode of conversational interaction.

The basic premise, upon which all of the auction simulations implemented here are based, is that one agent, the auctioneer, calls for monetary bids from one or more bidder agents. The winning bidder is the one that offers the most money for the lot on the auction block. Once the auction is over, the auctioneer accepts simulated currency from the bidder in exchange for the lot. Each bidder ascribes a value to the lot up for sale, which is kept private. This bidder will not submit a bid in excess of its perceived value[10]. If over the course of the auction the bids submitted do not meet or exceed a threshold value, the *reserve* price, no transaction takes place and the lot's original holder maintains

[10]Assuming the bidder is rational.

ownership. The four standard types of auctions are to be implemented: i.e., English, Dutch, Sealed Bid, and Vickrey (Klemperer, 1999).

The English, or *ascending-bid auction*, is one in which the auctioneer agent sets a low opening value for the lot, which may be less than the reserve price. When the auctioneer signals that the auction has begun by calling for bids, bidder agents may then offer to purchase the lot for a value greater than or equal to the opening price. The auctioneer acknowledges the bidder that made the bid it is currently considering and then calls for new bids higher than the amount just received. Again, bidder agents may submit a bid greater than or equal to the amount just *cried* and the auctioneer will acknowledge the new bid being considered, if any. Bidders do not have to wait for the auctioneer's cry to submit a bid, as the English auctioneer may accept bids at any time once the auction is underway. The cycle repeats until the auctioneer's call for new bids goes unheeded. At this point, the lot is sold to the bidder with the last bid considered if it exceeds the reserve price (Klemperer, 1999).

The Dutch, or *descending-bid auction*, is one in which the auctioneer agent sets a high opening value for the lot, far in excess of the reserve price. Bidding proceeds in the manner opposite to that of the English auction. The auctioneer progressively lowers the price of the lot until a bidder signals that it is willing to pay the current price or until it falls below that of the reserve. As soon as a bidder signals that it wants to buy the lot, the auction ends (Klemperer, 1999).

In the Sealed Bid, or *first-price sealed bid auction*, the auctioneer calls for bidders to submit a single secret bid. Unlike the English and Dutch auctions, a bidder is unaware of the lot's value as perceived by competing bidders. After a set amount of time, during which bids may be submitted, the auction ends and the auctioneer examines the sealed bids. The lot is sold to the bidder with the highest bid, as long as it exceeds the reserve (Klemperer, 1999).

In the Vickrey, or *second-price sealed bid auction*, the auctioneer calls for bidders to submit a single secret bid in the same manner prescribed by the Sealed Bid auction. These two auction types are, in fact, identical except for in the final price paid by the winning bidder. As with the Sealed Bid auction, the winning bidder is the one that is willing to pay the highest price for the lot. However, the price the winning bidder pays is the second highest amount bid (Vickrey, 1961).

2.5 Summary

The literature reviewed in this chapter covers topics important to the subject of agent communication. These topics include: the definition of *agency*, ACL semantics, and conversations, which provide the background needed to understand the current challenges known to the field. As well, an overview of auction theory was provided, as simulated auction systems will be implemented in order to demonstrate the reuse of modular conversationlets in the construction of first-order agent conversation protocols.

Chapter 3

Requirements Analysis

If this thesis is to successfully meet its aim (see Section 1.2), a requirements analysis must be undertaken as per Objective O2 (Analyze the Requirements). An overview of these requirements is provided in Section 3.1. From there, each section that follows discusses these broad requirements in greater detail, starting with Section 3.2, which identifies qualities necessarily found in the supporting multi-agent software. Section 3.3 describes characteristics agents must possess. Section 3.4 outlines the agent conversations' necessary structural requirements. Section 3.5 describes the factors that need to be considered in order to determine if this project successfully achieves its stated aim. Finally, Section 3.6 identifies the user-interface requirements that need to be met to allow human observers to intervene in the agent society and verify success.

3.1 Requirements

This thesis aims to analyze and demonstrate the feasibility and value of conversations as the first-order mechanism of agent communication. The issues associated with the construction of conversational protocols (as described in Section 1.1) are made manageable if conversations are constructed from modular, social commitment-connected message sequences (as illustrated in Figure 3.1). Toward this end, it will demonstrate the following:

1. How social commitments are used in conjunction with policies to connect conversationlets to make a whole conversation

2. How conversationlets can be easily reused across conversations

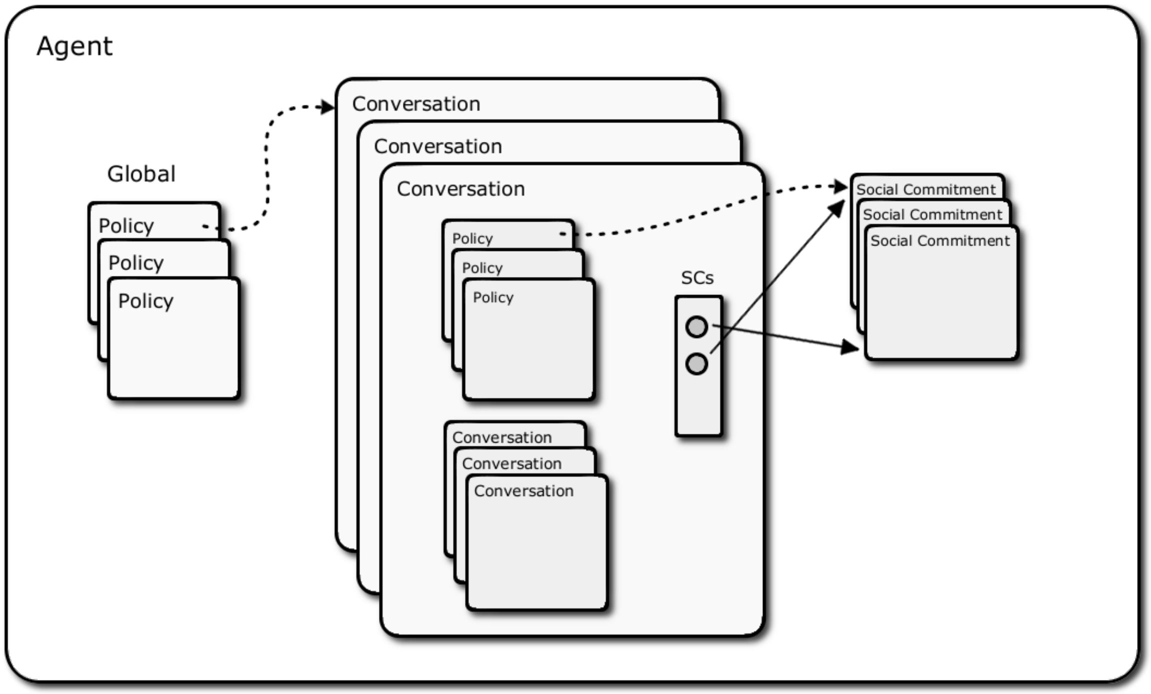


Figure 3.1: The conversation/conversationlet/social commitment relationship.

3. How the absence of unfulfilled social commitments signals a conversation's termination

To achieve this thesis's stated aim (see Section 1.2), several requirements must be fulfilled: there must be an existing, readily extensible MAS in which the role of policies and social commitments are commonly understood and integrated; the agents within this society must be equipped to treat conversations as their primary mode of interaction; the agents must have conversations with which to engage one another; conversations must *reuse* conversationlets where appropriate; investigative methods must be established; and there must be a user interface with which to monitor the outcome of the conversations in question. The following list provides an overview of each set of requirements:

S* - MAS Software Requirements The experimental agent society is to be built upon existing MAS software. As such, the software chosen must be able to accommodate extensions to its base functionality. The decision to employ extensible software is pragmatic, as there should be no need to re-implement well-established agent functionality, especially in terms of the underlying communication infrastructure. Likewise, the chosen MAS software must apply policies in the creation and fulfilment of social commitments (even though their precise roles are less established (Amgoud and de Saint-Cyr, 2008)) to avoid unnecessarily reimplementing the associated functionality in its entirety.

A* - Agent Requirements The agents designed for use in the chosen MAS are to treat conversations as their first-order mode of communicative interaction. As a consequence of employing extensible MAS software, communicative behaviour will be modified so that agents engage one another within the context of a conversation. That agents are capable of performing communicative acts is assumed from the fulfilment of S*. This alone is insufficient, however, as all communicative acts demanding a reply must be performed in the context of a policy-guided, social commitment-driven conversation.

C* - Conversation Structural Requirements It has been observed that different conversational patterns often contain identical sub-sequences of performative messages. As in object oriented programming, it can be convenient to catalog such patterns as second-order, reusable objects. Here, these patterns, or sequences of performatives, have been dubbed *conversationlets*. In order to properly use and reuse conversationlets, a `Conversation` container class must be implemented and it must accommodate the agent society's treatment of policies and social commitments.

I* - Investigation of First-Order Conversations Having attributed first-order status to the conversational mode of agent interaction (see A*), conversations tailored to agents' objectives and behaviour must be provided. These include primary conversations necessary to the formation of the society and secondary conversations that may take place once a society has been established. It must be shown that the various conversations achieve their intended objectives and terminate upon the fulfillment of all pertinent social commitments. As well, a secondary CASA conversation must be implemented in a non-CASA MAS so that the strengths and weaknesses of each implementation may be identified and the utility of the first-order conversational mechanism may be demonstrated.

U* - User Interface In addition to providing the means with which to intervene in the agents' society, a user interface is a necessary tool to provide feedback to observers and to verify correct agent behaviour for experimental purposes (e.g., to confirm that conversations terminate when all an agent's conversational social commitments have been fulfilled). Similarly, logs detailing agent interactions must be generated in order to further investigate and document agent behaviour.

3.2 MAS Software Requirements

The MAS software chosen for the purposes of this research must be readily extensible, as per the requirement stated in S*. Its code should be freely accessible to researchers, so as to allow for the necessary modifications and extensions. The software must already have implemented basic communication infrastructure and apply policies in the creation and fulfillment of social commitments. It is important that it would accommodate or provide some mechanism to allow agents to announce their presence or to establish contact with one another in the MAS.

S1: The chosen agent software must be extensible. The source code must be available and free from licensing restrictions that would otherwise disallow modifications to the base system. The ideal software will be open-source and free to use for research purposes.

S2: The chosen software must provide basic communication infrastructure. The chosen MAS must provide its agents the means to send and receive messages. The syntax of any existing messages is not critically important, so long as the software has implemented some ACL (e.g., FIPA).

S3: The chosen software must apply policies in the creation and fulfilment of social commitments. Though there is debate surrounding the appropriate application of social commitments in agent communication (see Section 1.1), the chosen MAS must, at a minimum, create and fulfil social commitments under the dictates of the society's policies.

S4: Agents must be able to seek out one another in order to establish contact. Agents need to announce their availability to communicate to other agents. The software should provide the means for this announcement to take place across a network.

3.3 Agent Requirements

Following from the requirements outlined in A*, the agents that occupy the MAS must be properly equipped to treat conversations as their first-order mode of interaction. This requirement also follows from the extensibility requirements outlined in S1. Following from Requirement S3 (The chosen software must apply policies in the creation and fulfilment of social commitments), agents must be policy-aware, *social*, and be able to interpret messages containing FIPA performatives (FIPA, 2002b).

A1: The agents must adopt conversations as their first-order communicative

mechanism. This requirement again follows, in part, from Requirement S1 (The chosen agent software must be extensible). It is not essential that all agents in a society treat conversations as their first-order communicative mechanism *exclusively*, but rather that they be prepared to act appropriately in response to agents that do.

A2: Agents must be policy-aware and social. Following from requirement S3 (The chosen software must apply policies in the creation and fulfilment of social commitments), the agents that would participate in the society must accommodate the MAS's application of policies and social commitments. That is, in order to participate in the society, agents must be able to apply and adhere to the policies that govern the creation and fulfilment of social commitments.

A3: Agents must interpret messages containing FIPA performatives. Part of the aim of this research is to contribute to a workable semantics for the FIPA ACL. As such, the chosen MAS's agents need to *speak* the language.

3.4 Conversation Structural Requirements

This section follows from the requirements outline in C*. Much of what motivates this research revolves around observations made of existing agent conversational protocols. Specifically, it was observed that the messages required to properly implement FIPA's *propose* and *request* performatives could comprise conversations that contain identical subsequences of messages[1]. The recurring performative message patterns in existing protocols, once isolated and identified, were dubbed *conversationlets*.

Cataloging recurring sequences of performative-containing messages and packaging

[1]The conversations required to implement *propose* and *request* serve different purposes and are dependent upon different agent roles. I.e., a *propose* conversation is initiated by a *server* agent, while a *request* is initiated by a *client*

them as reusable conversationlets is useful for improving software in a variety of ways, as is well-documented in software engineering literature (Booch, 1995). For example, by reusing the code that comprises a conversationlet we may improve efficiency both in terms of system resources and system design. As well, from the AI perspective, the reuse of conversational patterns, especially in diverse contexts, mimics human interaction[2]. For these reasons, it is supposed that conversationlets should be reused whenever appropriate.

The use and reuse of conversationlets depends upon the creation of a suitable `Conversation` container class. This class will provide the functionality required for conversations to take first-order status as the preferred mode of agent interaction. Social commitments and policies are integral to the internal organization of conversationlets in the `Conversation` class. As such, the class must accommodate the expectations of policies and social commitments established by the chosen MAS.

C1: Identify and package recurring message patterns as conversationlets.

The *propose* and *request* performatives, as specified by the FIPA ACL, require the exchange of multiple messages if they are to achieve their semantic intent[3]. Recurring patterns have been recognized in these sequences of messages. Once these message patterns are properly cataloged, they need to be demarcated and packaged in a manner consistent with the communication infrastructure of the chosen MAS.

C2: Implement a `Conversation` class to contain conversationlets. Having determined the precise nature and composition of conversationlets, a `Conversation` class

[2]Take for example the various conversations that take place in a fast food restaurant versus those in a sit-down restaurant. The contexts are different, but many of the conversations that take place are the same. In both contexts an order is placed, food is served, and the bill is paid. But, depending on the context, the customer may place an order with a waiter or a cashier and the order in which the bill is paid and the food is served will vary.

[3]I.e., so the *propose* and *request* performatives provoke agent actions appropriate to meaning commonly attributed to those words. E.g., one agent may *request* that another agent tell it the time of day. Or, one agent may *propose* it sell another agent a product or service.

needs to implemented so that they may be organized and contained. This class, in turn, must be incorporated into the structure of the society's agents in order to meet requirement A1 (The agents must adopt conversations as their first-order communicative mechanism).

C3: The `Conversation` class must accommodate the society's implementation of policies and social commitments. The chosen MAS must implement policies and social commitments in the prescribed capacity (See requirement S3 (The chosen software must apply policies in the creation and fulfilment of social commitments)). As such, the `Conversation` class must be designed in a manner fitting with the society's expectations of these components. This is also important because social commitments are an integral part of drawing conversationlets and the messages they contain into a complete, coherent conversation, as per the stated aim of this thesis (see Section 1.2).

3.5 Investigation Requirements

Following from the requirements outlined in I* and the expectations established by Objective O5 (Analyze the System), the agents must have conversations in which to engage if the efficacy of the first-order conversational mechanism is to be established. It must be shown that all of the back-and-forth communication that once took place in the established MAS can be redeployed under the auspices of the new first-order conversational mechanism.

Certain conversations need to take place in order to establish an agent society. Here, these are called *primary* conversations. They establish the means through which agents may seek out one another in order to interact. If denied the ability establish mutual awareness in a common operating environment, artificial agents cannot form a MAS, as defined in Section 2.1.2. The MAS software chosen for this research will require that certain formative communicative interactions take place in order to allow its agents to become aware of and interact with one another. These primary interactions must be take place within the context of a first-order conversation if the aim of this research is to be fulfilled.

Agents, having formed a society by engaging in primary conversations with one another, must be provided the means to interact in ways dependent on the existence of their society. *Secondary* conversations will be created for that purpose. Specifically, the first-order conversations necessary to simulate several different auction types will take place within the MAS. As with the primary conversations, secondary conversations are to be created in order to demonstrate the feasibility and value of conversations as the first-order mechanism of agent communication.

Taking what was previously written in this section into consideration, five investigative requirements have been identified. The first requirement involves identifying the conversations required to establish a MAS. The second involves deploying those conversations under the auspices of the first-order conversational mechanism. The third requires that new conversations be developed that are secondary, and dependent upon, the formation of the agent society. The fourth requires that correct agent behaviour and conversation termination be verified. And finally, the fifth requires the deployment of secondary conversations in a non-CASA MAS in order to provide contrast and demonstrate the utility of the CASA first-order conversational mechanism.

I1: Identify the conversations necessary to form a MAS. This requirement follows from Requirement S4 (Agents must be able to seek out one another in order to establish contact). It is assumed that a MAS cannot exist unless its agents are aware of the society's other members and are able to establish contact with them. This expectation may necessitate the use of a *registrar* agent, to which a society's would-be member agents announce their presence. Member agents may also turn to the registrar agent for address information when seeking to establish contact with other members.

I2: Deploy primary MAS conversations. Following from Requirement I1, the agent interactions necessary to establish a MAS must be conducted as first-order conversations.

I3: Deploy secondary conversations. These are conversations that cannot take place outside of an established MAS. Once an agent society has been established, its purpose is determined by the conversations that take place within it. The fulfillment of this requirement and I2 will mean having met Objective O4 (Implement the System).

I4: Verify agent behaviour and conversation termination. Having attributed first-order status to the conversational mode of agent interaction (see Requirement A* - Agent Requirements), and having provided the society's agents with conversations designed to achieve a variety of purposes (see Requirement C* - Conversation Structural Requirements), it must be shown that the conversational agents achieve their intended objectives and all instantiated conversations terminate upon the fulfillment of all pertinent social commitments. This is done in fulfilment of Objective O5 (Analyze the System).

I5: Deploy a secondary conversation in a non-CASA MAS. This research aims to, in part, build a system in which modular conversation patterns may be used in conjunction with social commitments and policies to easily build and design complete, coherent agent dialogues. In order to determine if this aim has been successfully achieved, at least one of the secondary conversations deployed under the CASA MAS must be deployed under a comparable MAS. This implementation must be compared against the CASA implementation so that the strengths and weaknesses of each may be identified.

3.6 User Interface Requirements

Following from requirement U*, a user interface must be provided so that human observers may confirm correct agent behaviour. The ability to confirm appropriate behaviour is necessary from an experimental standpoint and to accommodate the supposition that agents ultimately act upon human interests. As such, it is necessary to enable human observers to intervene in the agent society, if only to initialize the conversations with which to assess agent behaviour.

Following from Requirements I2 (Deploy primary MAS conversations) and I3 (Deploy secondary conversations), the interactions that take place in both the society's formative and secondary conversations must be made observable to humans. For the former requirement, a simple way to verify whether or not a formative conversation has successfully concluded would be to trace the formation and fulfillment of social commitments when an agent engages another in conversation. This data must be logged for later analysis. For the secondary conversations (e.g., those that comprise an auction), specialized interfaces will need to be provided. Human observers must be able to manage an agent's inventory, desires, and the values it attributes to items in both categories. Having done that, the human must be able to initiate the auction proceedings, observe the bidding, and then be able to verify that the auction was conducted fairly and that the final transaction between the auctioneer and winning bidder successfully concludes.

U1: Agent interactions must be logged. This is especially important for monitoring the formation and fulfillment of social commitments carried out during the course of a conversation. The logs will be consulted to verify correct agent behaviour.

U2: User interfaces need to reflect agent functionality. Conversations secondary to the formation of the MAS will need specialized interfaces to enable a human observer to adjust an agent's internal data, verify correct behaviour, and motivate the agents' conversations in some way.

3.7 Summary of Requirements

This aim of this thesis is to establish conversations as the first-order mechanism of agent communication. It will identify recurring patterns of performative messages in existing conversations and modularize them as *conversationlets*. It will demonstrate how social commitments are used to combine conversationlets into a single coherent conversation. In doing so, it will help disambiguate the role social commitments play in social semantics and demonstrate how the absence of unfulfilled social commitments signals the conversation's termination. Several requirements must be met before this goal can be achieved.

The requirements of this thesis, at their root, revolve around the extensibility of the software from which the conversational agents and their society will be constructed. It must be sufficiently flexible to allow modifications to agent functionality. The agents built upon the software must adopt conversations as their first-order mode of engagement. In turn, several requirements are made of the `Conversation` class implemented with this purpose in mind. Having incorporated the `Conversation` class into their base functionality, the agents will need conversations with which to engage on another. These conversations have been categorized as *primary* and *secondary*, where the former are essential to the formation of a MAS and the latter are dependent on the formation of a

MAS. Here, the secondary conversations to be provided are designed to simulate several different auction types, none of which could take place outside of an established agent society. Once the conversations that form the MAS are implemented and secondary auction conversations created, a user interface must be provided to the human observer, who must be allowed the opportunity to intervene in the society, if only to motivate the agents' secondary conversations. Having motivated the secondary auction conversations, and assuming they have been properly composed of conversationlets, it must be demonstrated that the agents are able to successfully simulate various auctions before all their social commitments have been fulfilled, which will signal the conclusion of their communicative interaction.

Chapter 4

Design and Implementation

This chapter describes the steps taken to meet Objectives O3 and O4 in Section 1.3: the design and implementation of the conversational framework. It also describes how the requirements outlined in Chapter 3 are to be met. Section 4.1 provides the rationale for the choice of CASA as the foundational MAS software upon which first-order conversational agents and their societies will be built. Section 4.2 describes the details of how CASA agents were modified to treat conversations as their first-order mode of communication. Section 4.3 describes the identification and modularization of conversationlets, the class designed to organize them, and the role played by CASA-style social commitments and policies in composing coherent first-order conversations. Section 4.4 provides an overview of the investigatory steps taken so that the effectiveness of the first-order conversational mode of communication can be demonstrated. Section 4.5 describes how the interface requirements are fulfilled for the investigation of, and intervention in, the CASA agent society. Finally, Section 4.6 summarizes what was covered in this chapter.

4.1 MAS Software

The Collaborative Agent System Architecture (CASA) was chosen as the foundational MAS software for this research. It meets all of the criteria encompassed in Requirement S*. A description of how CASA fulfills each specific requirement follows.

CASA is freely available, readily extensible, and is intended for research and experimentation (Kremer et al., 2004). This fulfills Requirement S1 (The chosen agent software must be extensible). It also provides the facilities for agents to communicate via TCP/IP ports. Messages are sent and received through the ports agents claim on start up. The performatives contained in these messages are compliant to a superset of the FIPA standard (FIPA, 2002a). Message headers may be either XML (Bray et al., 2008) or KQML (Finin et al., 1994) formatted. These CASA features fulfil requirement S2 (The chosen software must provide basic communication infrastructure).

As has been alluded to previously, CASA has implemented *request* and *propose* conversation protocols (Kremer and Flores, 2006). Agents are motivated to exchange the messages necessary to execute these protocols by fulfilling the social commitments that form over the course of their execution. The social commitments, in turn, are added and subtracted from the agents' lists of obligations as dictated by the society's policies (Kremer and Flores, 2006). In this way, CASA's treatment of social commitments and policies meets Requirement S3 (The chosen software must apply policies in the creation and fulfilment of social commitments).

In fulfilment of Requirement S4 (Agents must be able to seek out one another in order to establish contact), CASA instantiates a Local Area Coordinator (LAC) upon start up (Kremer et al., 2004). The LAC is the first agent initialized in a society. Its primary roles are to instantiate other agents and to accept their registrations. That is, upon instantiation, a new agent is required to register itself with the LAC so that the wider society may know what agents are currently running and the URL addresses through which those agents may be contacted. In further facilitating agent interaction, CASA provides a specialized agent call the Cooperation Domain (CD) (Kremer et al., 2004). The primary purpose of a CD is to act as a communication proxy. As a proxy, the CD may receive a message from a member agent and relay it to all other member agents. Agents (including other CDs) may request to join a CD. Membership in a CD affords agents the ability to send proxy messages to other members and to have access to the membership roster. This thesis uses CDs to determine which agents are participating in any of the various auctions that will be implemented. By providing the LAC and CD agents, CASA fulfils requirement S4.

4.2 Agents

As outlined in Requirement A*, the agents designed for the purposes of this research must interpret messages containing FIPA performatives, be policy-aware and social, and treat conversations as their first-order mode of communication. As most of the requirements pertaining to FIPA, social commitments, and policies have already been addressed in Section 4.1, there will be only limited discussion of how these agent requirements are fulfilled. The bulk of this section will be spent detailing the steps taken to fulfil Requirement A1 (The agents must adopt conversations as their first-order communicative mechanism).

As stated in Section 4.1, CASA performatives are compliant with a superset of the FIPA standard. The messages in which these performatives are sent and received may conform to either the XML or KQML specifications. In order to communicate within the CASA MAS, agents must be designed to send, receive, and interpret the contents of such messages. Though CASA itself does set any internal architecture requirements for the agents that interact through it (Kremer et al., 2004), the agents designed for the purposes of this research extend existing CASA agent classes. These new agents, by extension, possess the functionality necessary to exchange FIPA compliant messages in either the XML or KQML formats. This fulfils Requirement A3 (Agents must interpret messages containing FIPA performatives).

The role played by social commitments, policies, and the relationship between those two components is already established in CASA (Kremer and Flores, 2006). When an agent exchanges a message with another, the policies available to each agent translate message's performative into a set of social commitment operators (e.g., *add* or *delete*), which are then applied. Since the agents constructed here are extensions of existing CASA agents, they are already equipped to interpret policies and create and fulfil social commitments based on that interpretation. This satisfies Requirement A2 (Agents must be policy-aware and social).

In order to fulfil Requirement A1 (The agents must adopt conversations as their first-order communicative mechanism), modifications were made to CASA's base agent class to accommodate conversations in the manner prescribed and a `Conversation` class was implemented for the purpose of organizing and executing conversationlets, which are written in Common Lisp. The `Conversation` class is the software mechanism that affords conversations first-order status. The following subsections detail the modifications necessary to fulfil Requirement A1.

4.2.1 Base Agent Modifications

CASA's `TransientAgent` class provides all of the basic functionality expected of an agent in a CASA MAS. All of the agents that will be implemented for experimental purposes treat this as their foundational class. Though this basic CASA agent came with much of the required functionality already implemented (e.g., basic sociability, policy awareness, etc.), many modifications needed to be made to so that conversations became their preferred mode of communication.

The `TransientAgent.handleEvent` method is a pre-exisiting method that was modified to enable agents to contextualize their policies. `Conversation`s provide this context, so it was necessary to enable agents to determine if they are already engaged in an existing conversation and whether or not it is appropriate to instantiate a new `Conversation` in response to an `Event`[1]. By contextualizing policies within conversations, CASA agents

[1]Typically, of course, the types of `Events` that require the instantiation of a `Conversation` are sent or received `MessageEvents`, which are assigned a unique conversation identifier.

no longer rely exclusively on global policies.

Global policies are those that exist outside the bounds of a conversation. When an agent receives an `Event`, it always refers to its global policies. Some global policies are flagged *always applicable*. As such, the agent will always apply them. There are others that are only applicable to certain `Event`s. The agent will compare its collection of global policies against an `Event` to determine which of those need to be applied in addition to those flagged *always applicable*. Global policies are not contained within conversations, but often determine whether instantiating a new conversation is appropriate. When all of the relevant policies have been identified, they are executed and social commitments are formed or fulfilled as a result.

To enable CASA's `TransientAgent` class to contexualize policies within first-order conversations, the following instructions were added to the `handleEvent` method:

```
$policies \leftarrow \emptyset$
**if** $conversation\_id \in event$ **then**
  **if** $\exists$ $conversation$: $running\_conversations(conversation\_id)$ **then**
    $policies \leftarrow get\_relevant\_policies(event, conversation)$
  **else**
    $policies \leftarrow get\_relevant\_policies(event, global\_policies)$
  **end if**
  $policies \leftarrow policies \cup always\_apply \subseteq global\_policies$
**end if**
$apply(policies)$
```

A new conversation may be instantiated when the global policies collected in the algorithm listed above are applied. If a conversation-instantiating policy is executed, the resulting conversation will have the policies it contains matched to the incoming event and executed, if applicable.

Finally, a handful of methods were defined within the `TransientAgent` class to allow for the instantiation and management of `Conversations`. These are intended to be called by the various agent initialization scripts written in Common Lisp. The most important of these are:

`put-policy`: This is called to create the *global* policies mentioned previously. Global policies are often the ones that instantiate new conversations in response to a received `Event`.

`instantiate-conversation`: When a global policy has been identified as *relevant* to an event that has taken place, it may call this function to instantiate a new `Conversation`, which contain the policies necessary to the ensuing communicative interaction.

The design and implementation steps described in this section were taken so that the `TransientAgent.handleEvent` method will check every incoming and outgoing `MessageEvent`'s unique conversation identifier against an agent's collection of instantiated `Conversation` objects. This was done so that the policies collected in the `Conversation` with the matching identifier can be applied. This gives primacy to conversations in communicative interaction between CASA agents. As such, Requirement A1 (The agents must adopt conversations as their first-order communicative mechanism) is fulfilled.

4.3 Conversation Structure

The following steps are taken in order to fulfil the requirements outlined in Requirement C*. These include identifying and modularizing recurring message patterns as conversationlets (C1) and implementing a `Conversation` class to contain conversationlets (C2). The implementation of the `Conversation` class must accommodate CASA's treatment of policies and social commitments (C3). This section describes the design and implementation steps taken to fulfil each of these requirements.

4.3.1 Conversationlet Identification and Modularization

Existing CASA conversation protocols account for a number of conversational turns. For example, when one agent sends a *request* message to another, the receiving agent may respond in a number of ways. It may respond with a message containing a performative indicating that it *agree*s to perform the requested action. It may also *refuse* or indicate that it did not understand the contents of the message with a *not-understood* performative. If a certain amount of time elapses without any communication, CASA will issue a *timeout*[2]. Any of these performatives may be issued in response to a *request* message, as the receiving agent is not necessarily under any obligation to perform any *request* made of it.

The messages exchanged between a client and server agent engaged in a *request* conversation is illustrated in Figure 4.1. The conversation commences when the client sends a *request* message to the server. The server will respond in any of the three ways described in the previous paragraph, or CASA may issue a *timeout* if the server does not respond within an allotted amount of time. As illustrated, *not-understood*, *refuse*, and *timeout* result in the conversation's termination while *agree* prolongs it. If the server agrees to the client's *request*, the client is allowed the opportunity to *cancel*, the server may report a *failure*, or again, if the time allotted for the server to respond elapses, CASA may issue a *timeout*, whereupon the conversation will terminate. If the the server agent successfully performs the requested action, it issues a *propose/discharge*, which tells the client agent that the server believes it has completed its task. The familiar *not-understood* and *timeout* performatives result in the termination of the conversation,

[2]See Figure 2.2 for a complete overview of performatives available in CASA and their relationships to one another.

as does the client's *refuse*, which tells the server agent that the client does not agree that the requested action was performed successfully. If the client does agree with the server's report of success, then it sends an *agree/propose|discharge* message and the conversation terminates having fully run its course.

The messages exchanged between a client and server agent engaged in a *propose* conversation is illustrated in Figure 4.2. The conversation commences when the server sends a *propose* message to the client. The client will respond with a *refuse*, *not-understood*, or *agree* performative. Here again, if the client does not respond within the allotted amounted of time, CASA will issue a *timeout*. From there, the *propose* and *request* protocols are identical.

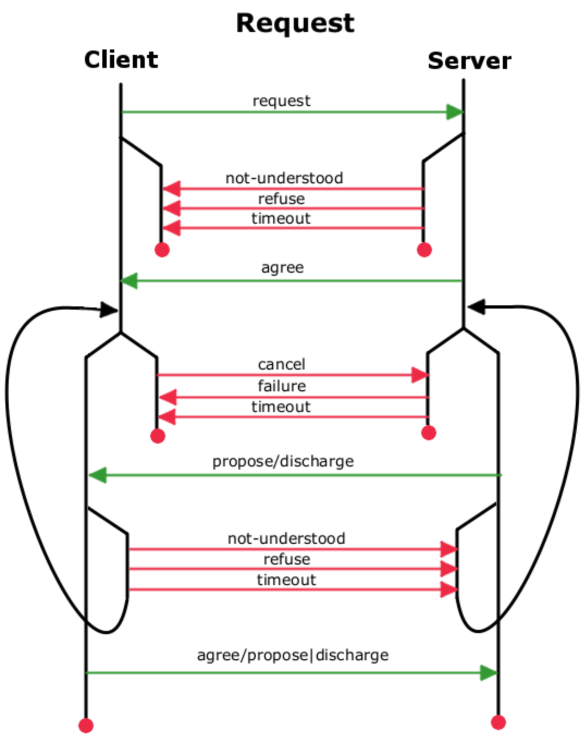


Figure 4.1: The CASA *request* protocol.

It has been observed that existing CASA conversation protocols share common sequences of performatives in accounting for conversational turns. This is especially true of conversational turns that indicate a communication failure and an agent's inability or unwillingness to perform an action, as when a *request* message has been received but cannot be fulfilled. These message sequences are often identical in both CASA's *request* and *propose* conversation protocols (see Figure 4.2 for an illustration of CASA's *propose* conversation). By packaging these common sequences as conversationlets, each conversation may be composed of common modules, which then may find application in the construction of new conversation protocols. Figure 4.3 illustrates the message sequences common to CASA's *propose* and *request* conversation protocols.

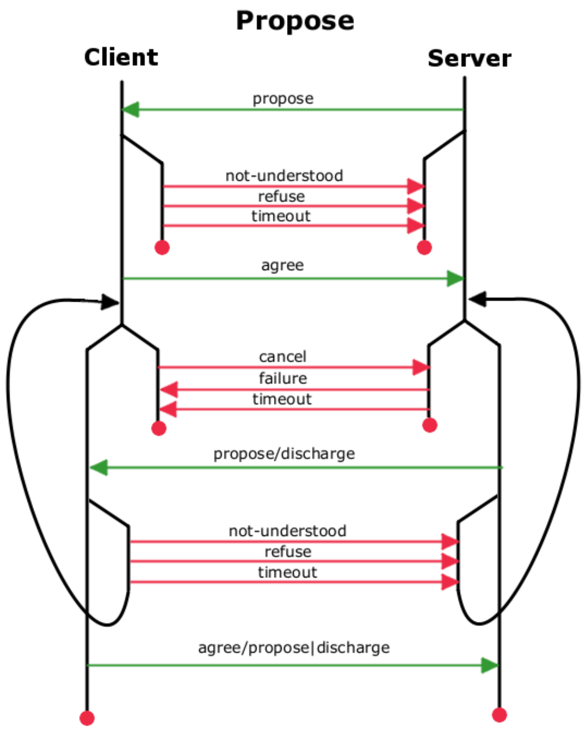


Figure 4.2: The CASA *propose* protocol.

The common message sequences in each of CASA's conversation protocols were identified and packaged as Common Lisp functions. The purpose of these functions is to

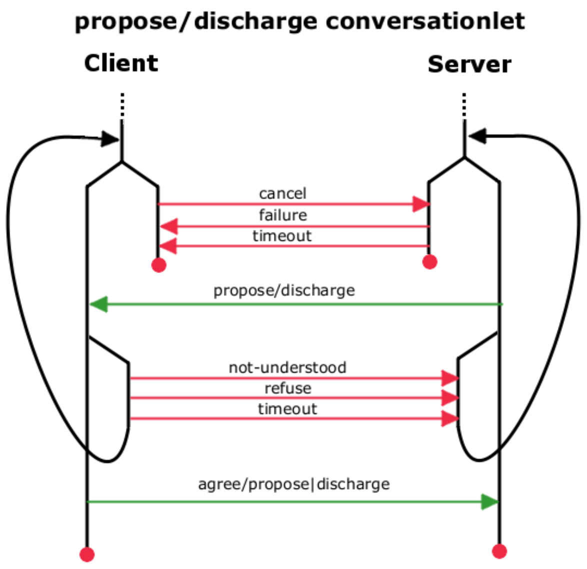


Figure 4.3: An example message sequence common to CASA's *propose* and *request* conversation protocols.

connect each performative message to a particular agent behaviour wherever necessary. This allows an agent to, for example, execute an internal method appropriate to the receipt of a *request* performative message. Further to the example given earlier, this method may determine if the agent is capable of performing the requested action, or may verify if the message is in a format that is able to understand. Having scrutinized the message in this way, the agent will then respond with an appropriate message, which will then be considered by the agent that initiated the interaction with internally defined methods of its own.

Given that message sequences, or conversationlets, were found to be identical between the respective CASA conversation protocols, it was deemed useful to incorporate their associated Lisp functions into conversation templates. These templates provide the basic structure with which internally defined agent methods are connected to the performative messages an agent may send or receive. This allows for the easy creation of conversations initiated with a *request* or *propose* performative message. As with the conversationlets themselves, conversation templates are implemented in Lisp.

4.3.2 Conversation Templates

Conversation templates are Lisp functions designed to couple agent behaviour[3] with message performatives. Policies are used to solidify this relationship. An agent, when initiating or participating in a conversation, will send or receive a message containing a performative. Relevant policies are collected (as described in Section 4.2.1) and applied. The agent acts in an appropriate manner, as dictated by the behaviours specified in the applied policies. A description of conversation templates and conversationlet functions follows.

CASA currently provides four conversation templates: `request`, `propose`, `subscribe`, and `query`. Their corresponding Lisp functions are defined in `defs.process.lisp` file, which is loaded by every agent on startup. There is also a `defs.agents.lisp` file, which defines global policies. There is currently one policy that CASA always attempts to apply. When a debtor agent is committed to send a message, this *always-apply* policy will fulfil that commitment when the message is sent. Likewise, if a creditor agent is expecting to receive a message from a debtor, it will fulfil the associated commitment when the message is received. Since this policy is required of all the conversation templates currently offered, it is only necessary to define it once. A brief description of CASA conversation templates follows:

`request:` A request is made of a server agent by a client.

`propose:` A proposal is sent to a client agent by a server.

`subscribe:` A client agents requests a server to send an `inform-ref` whenever the event

[3]Via internally defined methods

specified in the message takes place in the environment. The commitment formed by a `subscribe` conversation is persistent; it will not be fulfilled until either the client or server agent makes the initiative to have it removed.

`query`: A request for information made of a server agent by a client.

All CASA agents load these function definitions when they are first initialized. This allows the agent designer to specify which agent methods get called in response to a particular conversational turn. For example, if a client agent were to *request* that a server perform an *action*, the agent designer dictates what method the server agent will call in order to determine if it is willing and able to fulfil the request. Template functions are called in an agent's `*.init.lisp` files, where the wild card is the agent's fully qualified class name. Since CASA has incorporated an ABCL interpreter, the agent designer is free to include any well-formed Lisp code in an agent's conversation initialization file. Each conversation template has a client-side definition and a server-side definition. The example that follows shows the client conversation definition in `casa.TransientAgent.init.lisp` for a `request` to join a `CooperationDomain`:

```
(request-client "join_cd"
  `(jcall
    (jmethod (agent.get-class-name)
      "release_join_cd" "casa.MLMessage")
      agent (event.get-msg)
    )
  )
)
```

Make note of the parameters passed to the `request-client` function. The first, `join_cd`, is an action defined in `TransientAgent`'s corresponding ontology file. Agent ontologies are defined in the `*.ont.lisp` files, which follow the same naming rules as `*.init.lisp` files. As the name suggests, `join_cd` is the action an agent would take to join a `CooperationDomain`. The second parameter dictates what the requesting client agent will do if the CooperationDomain successfully fulfills its request. In this case, the release_join_cd method, as defined in the TransientAgent class will be called.

In the case of the request join_cd conversation, the server-side conversation definition looks similar to its client-side counterpart. The following definition is found in casa.CooperationDomain.init.lisp:

```
(request-server "join_cd"
  (jcall
    (jmethod (agent.get-class-name)
      "perform_join_cd" "casa.MLMessage")
      agent (event.get-msg)
    )
  )
)
```

Though the client and server-side join_cd conversations are similar in appearance, there is one significant difference (aside from the function names): the second parameter dictates the method called by the CooperationDomain when it receives a request from another agent to join. In this case, the CooperationDomain will attempt to allow the requesting agent to join.

Every conversation template has parameters that define what methods an agent must call whenever a particular message performative is sent or received. The join_cd conversation examples listed above are among the simplest cases. Should communication breakdown over the course of that conversation, or if the CooperationDomain fails to perform the requested action for some reason, the conversation templates define the default actions to be performed. If the agent designer determines that the default actions are not satisfactory, customized agent behaviour may be specified by naming the associated method calls via the corresponding parameters.

Conversationlet Functions

As has been declared from the start, conversations are comprised of conversationlets. The conversation templates described above are where this declaration is put into action. Since the regular conversational patterns upon which this research is grounded were first identified in CASA's `request` and `propose` conversations, these are the protocols that will be examined.

The form, content, and purposes of the `request` and `propose` conversations are similar enough that it was deemed unnecessary to define distinct Lisp conversation functions for each. When the agent designer calls upon either of the client or server-side `request` functions to create a `request` conversation, a corresponding `propose` conversation is also created. This is appropriate because it stands to reason that a server-agent should be allowed to offer the services it provides to a client without a `request` first being made, if it should decide to take that initiative. If the agent designer decides that a server should not be able to send a `propose` message, there is a boolean `nopropose` parameter that may be set in each conversation function.

The client and server-side Lisp functions that define the `request` and `propose` conversation are each comprised of Lisp-defined conversationlets. A call to the conversation functions takes the agent designer's parameters and passes them on to the conversationlet functions it contains. Recognizing that `request` and `propose` conversations are created as a result of the same conversation function calls, these are the conversationlet functions they contain:

`request-client` :

`ask-client:` takes the agent method calls specified by the designer and couples them with the appropriate client-side `request` conversation policies.

`offer-client:` takes the agent method calls specified by the designer and couples them with the appropriate client-side `propose` conversation policies.

`discharge-client`: takes the agent method calls specified by the designer and couples them with the policies that will fulfill all hanging social commitments on the client's side of the conversation.

`request-server` :

`ask-server`: takes the agent method calls specified by the designer and couples them with the appropriate server-side `request` conversation policies.

`offer-server`: takes the agent method calls specified by the designer and couples them with the appropriate server-side `propose` conversation policies.

`discharge-server`: takes the agent method calls specified by the designer and couples them with the policies that will fulfill all hanging social commitments on the server's side of the conversation.

Note that the `discharge` functions named above are applied to the resulting `request` and `propose` conversations in the same way. That is, while the policies in the first half of the `request` and `propose` conversations do not directly correspond (as indicated by the distinct `ask` and `offer` function definitions), the `discharge` policies for both are identical (see Figure 4.3). The policies contained in the conversationlets created by the `ask` and `offer` functions, on the other hand, are mutually exclusive. This means, for example, that any `propose`-type conversation instantiated from the templates described above will have no access to any of the policies contained in the corresponding `request`-type conversation created from the same template. This mutual exclusivity between conversations and the common incorporation of `discharge` conversationlets are, perhaps, the clearest illustration of how conversationlets can be reused in constructing the conversations suggested by FIPA performatives.

Conversation Bindings

In order to make the conversation templates applicable to the diverse range of uses described above, it was necessary to *bind* certain symbols to values upon conversation initialization. For example, the `request` functions create both `request` and `propose` conversations, but an agent's role in a given conversation is determined by which agent initiated the conversation and the performative contained in the inaugural message. If the inaugural message is a `request`, then we know the sending agent is the client and the receiving agent is the server. Conversely, if the message is a `propose` then the sending agent is the server and the receiving agent the client. Conversation bindings in the case of agent roles determine which agent fills a given role in the conversationlets instantiated by a conversation function call.

Tracking the overall state of conversations between conversationlets depends on binding state symbols. For example, consider the client-side `request` conversation as illustrated in Figure 4.4. It consists of two conversationlets: *ask* and *discharge*. The conversation transitions from state-to-state as the policies in the *ask* conversationlet are applied. The conversation may progress to a point where the *ask* conversationlet has no more applicable policies, leaving it in the `terminated-pending` state. Since the *discharge* conversationlet's `waiting-discharge` state is bound to *ask*'s `terminated-pending` state, the conversation now applies the policies contained in the *discharge* conversationlet. In this way, the state of the conversation transitions between conversationlet states, which would otherwise have no relation.

The state-binding relationship illustrated in Figure 4.4 was implemented with the partial Lisp code that follows this paragraph. Details specific to certain function calls were omitted in favour of illustrating the `:bind-state` parameter, which is immediately relevant to this section. However, make note of the call to the `conversation` function, which in turn calls the `ask`, `offer`, and `discharge` functions necessary to instantiate the

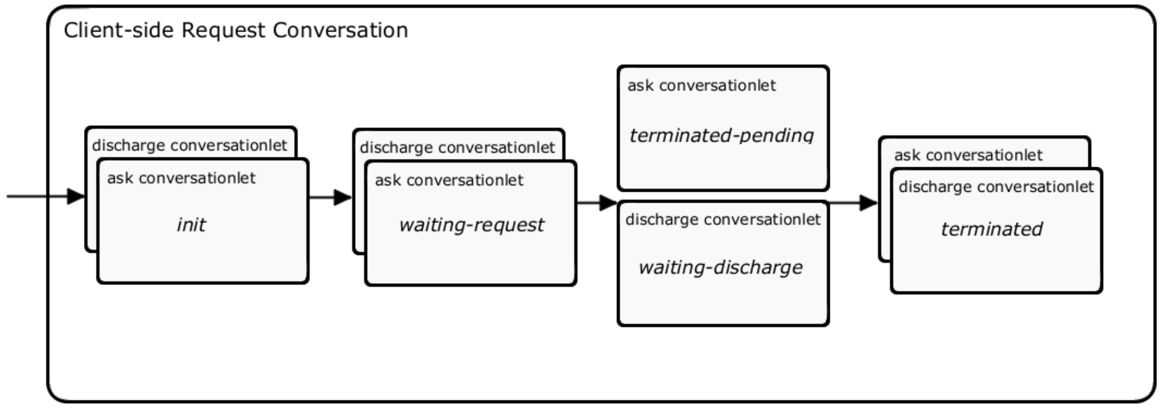


Figure 4.4: Bound conversation states in the client-side *request* protocol.

`request` and `propose` conversations, as described in Section 4.3.2.

```
(conversation name
  (list
    (ask-client ask-name the-act
      ...
      )
    (if nopropose
      ()
      (offer-client offer-name the-act
        ...
        )
    (discharge-client approver-name the-act
      ...
      )
  :bind-state
    (append `(
      ("init" ,ask-name "init")
      ("waiting-request" ,ask-name "waiting-request")
      ("terminated" ,ask-name "terminated")
      ("waiting-discharge" ,ask-name "terminated-pending")
    )
    (if nopropose
      ()
      `(
        ("init" ,offer-name "init")
        ("waiting-propose" ,offer-name "waiting-propose")
        ("terminated" ,offer-name "terminated")
        ("waiting-discharge" ,offer-name "terminated-pending")
        )
      )
    ‘(
       ("init" ,approver-name "blocked-init")
       ("waiting-request" ,approver-name "blocked-request")
       ("waiting-propose" ,approver-name "blocked-propopose")
       ("waiting-discharge" ,approver-name "init")
       ("waiting-propose-discharge-reply" ,approver-name "waiting-propose")
       ("terminated" ,approver-name "terminated")
       )
        ...
```

The preceding listing provides some of the Lisp code necessary to implement the state-binding required to construct a client-side `request` conversation from conversationlets, as illustrated in Figure 4.4. The conversationlets of note here are `ask-client` and `discharge-client`. They are labeled by the names passed through the `ask-name` and `approver-name` parameters, respectively. The `ask-client` conversationlet initiates a client-side `request` dialogue and `discharge-client` finalizes it. Here, the state symbols are bound from right to left. Take the `("waiting-discharge" ,ask-name "terminated-pending")` list as an example. When the `ask` conversationlet is in the `terminated-pending` state, the subsequent conversationlet needs to be in the `waiting--discharge` state. The subsequent conversationlet will, however, be in the `init` state, because it will just have been initialized. Here, the `approver` conversationlet binds its `init` state to the `waiting-discharge` state so that it can pick up execution where the `ask` conversationlet left off.

The design and implementation considerations described in this section were taken so that the recurring message sequences common to CASA's `propose` and `request` conversations could be identified and modularized. The modularization step was taken so that conversation protocols could be constructed from these common message sequences, or conversationlets. It was shown how conversationlets were implemented and incorporated into reusable conversation templates, so that an agent's behaviour can be dictated in response to the performative messages it sends and receives. Finally, it was shown how a conversationlets' state symbols are bound so that a conversation's overall state may transition from one conversationlet to another. All this was done in order to fulfil Requirement C1 (Identify and package recurring message patterns as conversationlets).

4.3.3 The Conversation Class

The `Conversation` class keeps a static-memory catalog of all the conversations known to a CASA MAS. Conversations are defined in CASA's `*.init.lisp` and an agent class's corresponding actions are defined in `*.ont.lisp`, as described in Section 4.3.2. The memory is organized in a tree structure, where nodes are conversations that may parent other conversations. Tree branches terminate with conversationlets. The `Conversation` class enables agents to query this tree, find the desired conversation, and instantiate individual instances, which are then used to engage other agents in conversation.

When an agent handles an event, as described in Section 4.2.1, it looks for applicable policies. It is within the global policy set that the agent may be called upon to instantiate a new conversation in order to properly address the event received. This is when a CASA agent queries the `Conversation` class to find the conversation appropriate to the situation. Once identified, the agent uses the returned template to instantiate a context-relevant `Conversation` object with all symbols properly bound (see Section 4.3.2).

Since CASA agents typically extend the `TransientAgent` class, it is inevitable that there would be overlap between a child agent and its ancestors. That is, child agents may require that the behaviour connected to certain messages exchanged in a conversation extend, or be different than that of their ancestors. In order to accommodate this eventuality, a conversational hierarchy was established.

Conversational Hierarchy

The `Conversation` class catalogues all of the conversations available to agents in a CASA MAS (See Section 4.3.3). It is not the case, however, that every agent is equipped to engage in every conversation available. Certain conversations may only be conducted with agents enabled or authorized to do so. Likewise, some agents, particularly those that inherit characteristics from parent agents, may required that the functionality contained within a conversation modify or override the functionality of its parent. Thus, there needs to be a method established through which conversational behaviour is attributed to the correct agents.

Though an agent may not be equipped to engage in a particular conversation, that is not to say that that agent is not be equipped to respond appropriately to a misdirected message. For example, supposing there was a *BankTeller* agent. It would be inappropriate for any given agent to check its bank balance through anything but a BankTeller. Should such a request be made of a *WeatherMonitor* agent, for instance, CASA does enable that agent to respond appropriately (i.e., by sending a message containing a `not-understood` performative). This functionality is built into the conversation templates, which in turn, relies upon an agent's ontology to define its scope of understanding and the actions it may perform. So while a `request bank-balance` message will not be understood by a WeatherAgent, it will recognize that a request was made, albeit one that it cannot fulfil.

The conversation hierarchy is closely related to the `*.init.lisp` and `*.ont.lisp` files described in Section 4.3.2. The former makes Lisp calls to create an agent's conversational templates, while the latter defines its ontology. The ontology files call the equivalent of a `super` method so that a child agent extends the ontology of its parent. As such, the child agent possesses all of its parent's conversational functionality as well.

The potential problem of an agent attempting to engage in conversations inappropriate to its design is solved by virtue of the order in which `*.init.lisp` and `*.ont.lisp` files are loaded and the relationships between agent classes: the files for primitive agent classes get loaded before any specialized child classes[4]. In this way, a WeatherMonitor agent has no knowledge of the conversations or actions necessary for a BankTeller agent to retrieve its bank balance, and vice versa. The WeatherMonitor could, however, `request` to know its bank balance from a BankTeller if they shared some parent ontology and conversation set. They could, for example, share a common `BankCustomer` ancestor.

The problem of distinguishing between the conversations shared by parent agents and child agents with specialized requirements remains, but is easily addressed. Take, for example, the `join_cd` conversation defined for `TransientAgents`. CASA defines a `ChatAgent`, which is a child of `TransientAgent`. The `ChatAgent` has a `join_cd` conversation with some functionality supplementary to its parent class. The `ChatAgent`, in fact, calls its parent's `super` methods when engaged in a `join_cd` conversation, but when the method returns, additional instructions are executed. If `casa.ChatAgent.init.lisp` were to simply redefine the `join_cd` conversation for its own purposes, that conversation would be overwritten for all agents, given CASA's current implementation of the `Conversation` class, as described in Section 4.3.3. This problem is avoided by simply not redefining the conversation in the `ChatAgent`'s initialization file. Rather, all conversation function calls are left to determine the class of the calling agent in the following manner:

```
(jmethod
  (agent.get-class-name) "release_join_cd" "casa.MLMessage")
  agent (event.get-msg)
)
```

Note the `agent.get-class-name` function call. So, although the conversation is defined in the `casa.TransientAgent.init.lisp` file, the `ChatAgent` is initialized with the

[4]E.g., a WeatherMonitor *is-a* WeatherAgent, so the WeatherMonitor is able to engage in all of the conversations loaded by its parent class.

conversations it needs to engage in its specialized `join_cd` conversation while `Transient-Agent`'s own conversations remain unmodified.

The `Conversation` class, as described in this section, is a repository that may be called upon by agents to instantiate individualized, context-relevant copies of the conversations it contains. Once instantiated, agents examine the policies contained within, determine which are applicable to the event received, and then apply those policies. Conversational functionality is attributed to the appropriate agents as described in Section 4.3.3. The steps described in this section fulfil Requirements C2 (Implement a `Conversation` class to contain conversationlets) and C3 (The `Conversation` class must accommodate the society's implementation of policies and social commitments).

4.4 Investigation

A full description of the investigation into the efficacy of the first-order conversational mechanism will be provided in Chapter 5 in order to meet Objective O5 (Analyze the System) and fulfil Requirements I*. This section provides a broad overview of the steps taken in that investigation. The overview is provided here in anticipation of the user interface requirements described in Section 4.5, which are shaped by these investigative requirements.

In fulfilment of Requirements I1 (Identify the conversations necessary to form a MAS) and I2 (Deploy primary MAS conversations), CASA's `execute` and `register_instance`, and `get_agents_registered` protocols have been implemented as first-order conversations using conversation templates. The first conversation is needed to execute Lisp instructions, including the instantiation of a society's agents. The second registers new agents with CASA's LAC so that they may seek out and establish contact with one another. The third allows agents to request the URL addresses of agents currently registered with the LAC. These three primary conversations are necessary to form a CASA society and have been deployed under the first-order conversational paradigm.

In fulfilment of Requirement I3 (Deploy secondary conversations), the conversations and agents necessary to simulate a simple financial transaction have been implemented. These conversations cannot take place outside of an established agent society, and have thus been deemed *secondary* to those necessary to societal formation. In order to demonstrate the modularity and ease-of-use of conversationlets, several auction agents and conversations were constructed. These simulate the English, Dutch, Sealed-Bid, and Vickrey-type auctions using the first-order conversational mode of communication.

In fulfilment of Requirement I4 (Verify agent behaviour and conversation termination), it is shown in Chapter 5 that agents function correctly in accordance with their design. That is, they successfully exchange products for currency as a result of carrying out one of the auction simulations. Further, it will be shown that all of the agents participating in the conversations necessary to conduct an auction simulation have no social commitments left unfulfilled once the simulation concludes.

In fulfilment of Requirement I5 (Deploy a secondary conversation in a non-CASA MAS), the CASA agents and conversations necessary to perform a simple financial transaction will be implemented in JADE[5]. This will provide the basis against which CASA's first-order conversational mechanism will be compared. Such a comparison is useful in revealing the strengths and weaknesses of each framework's approach to agent communication. These simple financial transaction agents are described in Chapter 5.

4.5 User Interface

The design and implementation decisions described in this section were made in order to fulfil the broad Requirement U* - User Interface, which encapsulates the specific

[5]An MAS comparable to CASA

requirements laid out in U1 (Agent interactions must be logged) and U2 (User interfaces need to reflect agent functionality). By fulfilling these requirements, it will then be possible to fulfil Requirement I4 (Verify agent behaviour and conversation termination).

In order to verify correct agent behaviour and conversation termination, several system events will need to be logged. The events of interest here include agent message exchanges, policy application, and the formation and fulfilment of social commitments. CASA already enables agent designers to set a number of flags to log a wide variety of system events, including those just mentioned. Individual log files are produced for every agent in a society, so the data necessary to verify agent behaviour is readily available. This feature of CASA fulfils Requirement U1 (Agent interactions must be logged).

CASA also provides its agents a foundational graphical user interface with tabs to monitor CD membership and social commitments. It also provides a tab with which users may enter command line instructions in the form of Common Lisp instructions. With the command line comes textual output, which displays the output resulting from command line arguments and the data collected in the agent log files. These graphical elements are insufficient for the purposes of this research by themselves, but does provide a foundation upon which to build graphical user interfaces customized to reflect specific agent functionality.

Graphical user interfaces were provided for both the `TransactionAgent` and `AuctionAgent` classes. The latter GUI extends much of the design and functionality of the former, with one exception: an *Auction* button is provided in place of a *Sell* button on the respective interfaces, as seen in Figure 4.5). This interface allows the user to determine what items an agent will buy or auction[6] by selecting the product in the corresponding list fields. The user may also modify the prices of listed items and an agent's *bank roll*, which simulates the currency an agent has at its disposal. The *Collectables* field depicts items

[6]Or *sell*, in the case of the `TransactionAgent` GUI

an agent has acquired, but will not put up for sale. Finally, the *Action* field outputs feedback so that the human observer may be informed of what events have transpired so far. Aside from the difference mentioned between the `TransactionAgent` and `AuctionAgent` GUIs, all agents built here employ this interface, thus fulfilling Requirement U2 (User interfaces need to reflect agent functionality).

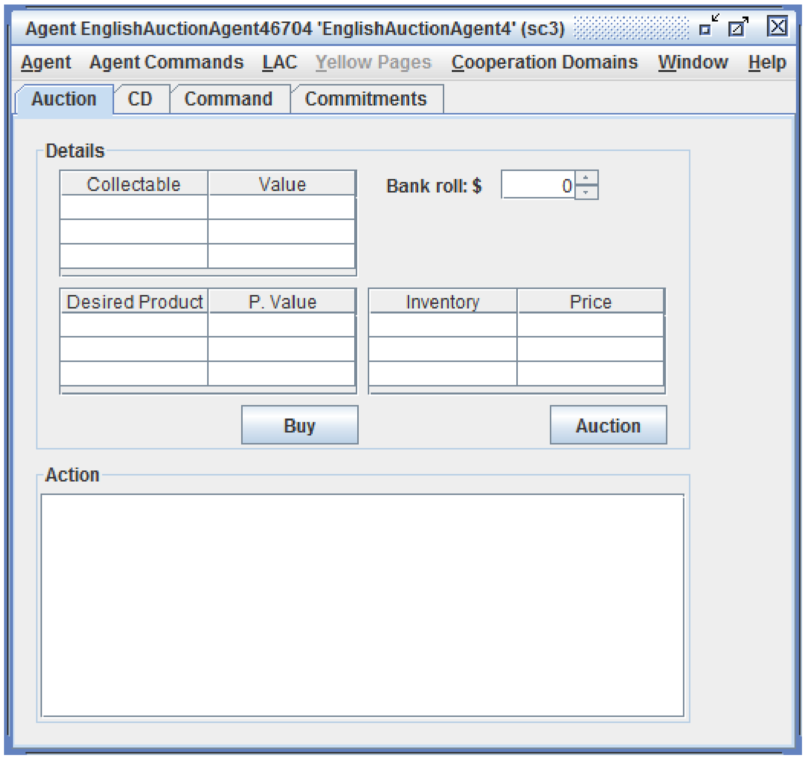


Figure 4.5: The `AuctionAgent` graphical user interface.

4.6 Summary

In order to help establish conversations as the first-order mode of agent communication, several requirements had to be fulfilled. This chapter was motivated by the Objectives O3 (Design the System) and O4 (Implement the System) declared in Chapter 1. The steps taken and described here demonstrate how the requirements identified in Chapter 3 were fulfilled.

The CASA MAS chosen for the purposes of this research provides the foundational agent infrastructure upon which conversation templates were deployed. These templates arose as the result of the desire to modularize recurring sequences of performative messages in reusable conversationlets. The conversationlets identified were used in the construction of three conversation templates: *request*, *propose*, and *subscribe*. Used in conjunction with CASA's newly introduced `Conversation` container class, the agent conversation designer needs only to specify which agent behaviour is associated with a particular conversational turn.

A broad overview of the investigatory steps taken in Chapter 5 was provided so that the user interface considerations outlined in this chapter could be made clear. Primary conversations were named alongside the secondary transaction and auction conversations chosen to demonstrate the effectiveness of the first-order conversational mechanism. As well, JADE was chosen as the multi-agent system to provide a basis of comparison to CASA. Toward this end, the simple `TransactionAgent` implemented in CASA will also be implemented in JADE so that the strengths and weaknesses of the respective MAS's approaches to agent communication can be identified.

Finally, user interface considerations were addressed to verify correct agent behaviour and to allow the human observer a way to intervene in the various agent societies. CASA already provides the means with which to log the system events of interest to this research. These events include: agent message exchanges, the application of polices, and the creation and fulfillment of social commitments. From there, graphical user interfaces were designed to reflect the functionality implemented in the `TransactionAgent` and `AuctionAgent` classes.

Chapter 5

Investigation

This chapter describes the steps taken to meet Objective O5 and will demonstrate the effectiveness of establishing agent conversations as the first-order mode of agent communicative interaction, as outlined in Requirement I*. In order to do this, Section 5.1 demonstrates that the protocols previously employed to form a MAS can be redeployed as first-order conversations, in the manner prescribed in Chapter 4 (as per Requirements I1 and I2). Likewise, Section 5.2 demonstrates that the conversations that depend on the establishment of a MAS can be similarly deployed (as per Requirement I3). Having implemented agents and conversations capable of engaging in the prescribed primary and secondary conversations, Section 5.3 demonstrates that agents behave appropriately, in accordance with the conversations in which they engage, and that all social commitments are fulfilled upon a conversation's termination (as per Requirement I4). Section 5.4 describes the JADE implementation of CASA's secondary transaction conversations and agents in fulfilment of Requirement I5. Section 5.5 describes a general purpose `Subscribe` extension made to CASA's `Conversation` class, which is designed for use with the FIPA semantic language (FIPA, 2002c). Finally, Section 5.6 summarizes the contents of this chapter.

5.1 Primary Conversations

Primary conversations are those necessary to form a functioning MAS. These conversations were described in Section 4.1. The purpose of CASA's formative, primary conversations is to establish a MAS's communication channels so that its member agents may engage one another. This is done with the `request execute`, `request register_instance`, and `request get_agents_registered` conversations.

Prior to the initialization of a CASA MAS, a temporary *bootstrap* agent called `casa` is instantiated for the purpose of executing the society's initialization file. Typically, its first task is to instantiate a `LAC` agent if one does not already exist. Then, allowing enough time for the `LAC` to initialize, the `casa` agent usually instantiates the society's other agents by sending `request execute` messages to the `LAC`[1]. The contents of these `request execute` messages may contain any properly formatted Lisp instruction accessible to the society's agents.

Once a society's `LAC` and other agents have been instantiated, the agents themselves send `request register_instance` messages to the `LAC`. By registering themselves with the `LAC`, agents make the URL addresses through which they may be contacted known to the society. If an agent should want to know the society's member agents, it can send a `request get_agents_registered` message to the `LAC`, which may then supply a list of URL addresses.

In summary, CASA's primary conversations consist of:

1. `request execute`
2. `request register_instance`
3. `request get_agents_registered`

These conversations are implemented using the conversation templates in the same manner described in Section 4.3.2. By identifying these formative societal protocols and redeploying them as first-order conversations, Requirements I1 (Identify the conversations necessary to form a MAS) and I2 (Deploy primary MAS conversations) are fulfilled.

[1]As stated, this is the event that *usually* takes place. The agent designer may, for example, decide that some different or supplementary interaction between the `casa` and `LAC` agents should take place.

5.2 Secondary Conversations

Secondary conversations are those that require that a MAS be formed before they can commence. These can be designed with any purpose in mind, but for the purposes of this research their end goal will be to simulate several different auction scenarios. This will be achieved by creating new conversations and an accompanying agent class designed to simulate simple financial transactions. Using these conversations and agent class, several new conversations and agent classes will extend the respective parent functionality in order to carry out a variety of auctions. This will demonstrate the extensibility of the first-order conversational mechanism and the reusability of its conversationlets in fulfillment of Requirement I3.

5.2.1 Transaction Agents

An agent's design is shaped by the first-order conversations in which it is to engage. Having identified the actions peculiar to a conversation, the conversational agent will need methods designed to perform those actions. Two conversation templates have been provided to connect these methods to the conversation's ontological actions (i.e., CASA's *request* and *propose* templates). Four actions were added to the `TransactionAgent`'s ontology:

`offer_to_buy`: When a client, or *customer* agent wants to purchase an item from a server, or *merchant* agent, it will send a send a *request* to the merchant to allow it to perform this action.

`offer_to_sell`: When a merchant agent wants to sell an item to a customer, it will send a send a *propose* to the customer to allow it to perform this action.

`inventory_list`: When a customer wants to know what items a merchant is selling, it will *request* its `inventory_list`.

`wants_list`: When a merchant wants to know what items a client is looking to purchase, it will *request* its `wants_list`.

The *items* being bought and sold by `TransactionAgent`s are represented by objects instantiated from the `Product` class. This class simply allows an agent to set and retrieve a `Product`'s details, such as its name and its value, as perceived by the agent. `Products` are contained in an `Inventory` object. A `TransactionAgent` implements two `Inventory` objects to track which `Product`s it has in its possession and which ones it wishes to acquire. When agents engage in any of the conversations associated with the actions listed above, they are exchanging messages that contain serialized `Products`.

`TransactionAgent`s are capable of taking on both the customer and merchant role, which means they are capable of engaging in any of the conversations required to perform the actions listed above. The following example illustrates how CASA's conversation templates are used to initialize the conversations necessary for a merchant to *propose* an `offer_to_sell` to a customer. The following creates the merchant's side of the `offer_to_sell` conversations:

```
(request-server "offer_to_sell" ;This is the conversation's name

;The method called when the server agrees to
;perform the requested action
 `(jcall
    (jmethod (agent.get-class-name)
      "perform_offer_to_sell" "casa.MLMessage")
      agent (event.get-msg)
    )

;The method called when the client agrees the server has
;satisfactorily completed the requested action
 :agree-discharge-action
 `(jcall
    (jmethod (agent.get-class-name)
      "conclude_offer_to_sell" "casa.MLMessage")
      agent (event.get-msg)
    )
 )
```

The following creates the customer's side of the `offer_to_sell` conversation:

```
(request-client "offer_to_sell" ;Conversation name

;The method called when the server sends a
;propose-discharge message. This allows the client to decide
;if the server successfully performed the requested action
 `(jcall
    (jmethod (agent.get-class-name)
      "release_offer_to_sell" "casa.MLMessage")
      agent (event.get-msg)
    )

;The method called by the client when the server proposes
;it perform an action. Here, the client decides if it wants the
;product the server is offering to sell
 :propose-decision
 `(jcall
    (jmethod (agent.get-class-name)
      "consider_offer_to_sell" "casa.MLMessage")
      agent (event.get-msg)
    )
 )
```

Here, when a customer[2] receives a message containing a *propose* for an `offer_to_sell`, it will call its internally-defined `consider_offer_to_sell` method, as passed through the `request-client` function's `propose-decision` parameter. If the product being offered for sale is one that it desires, the customer will *agree* to the merchant's proposal. The merchant will then call its `perform_offer_to_sell` method as its first step toward delivering the desired product to the customer. The merchant will then *propose* that the customer `discharge` the conversation, which is the next step taken and the one that actually delivers the desired product. The customer agent then calls its `release_offer_to_sell`

[2]I.e., a `TransactionAgent` that may purchase a product from another `TransactionAgent` acting as a merchant.

method to determine if the merchant did, in fact, deliver the desired product, at which point it may *refuse* to discharge the conversation because of some failure, or finalize the transaction with the merchant by paying[3] for the product. In either case, the merchant will be notified of the success or failure of the transaction and the details of the message will be processed by its `conclude_offer_to_sell` method. After that, the conversation terminates.

5.2.2 Auction Agents

Since an auction is conducted with the final goal of exchanging a product for currency in mind, it is reasonable to expect that the functionality of the `TransactionAgent` described above can be extended to construct an `AuctionAgent`. This extensibility is true of both the agents' class and of the actions and conversations for which they are designed. Like the `TransactionAgent`, which may engage in conversations designed for both customers and merchants, the `AuctionAgent` is equipped to engage in conversations designed for both bidders and auctioneers. Since the `AuctionAgent` extends the `TransactionAgent` class, it inherits all of its transaction-oriented functionality.

The `AuctionAgent` is an abstract class, upon which the four different types of auction agents will be built[4] (see Figure 5.1). It provides one abstract method, `startAuction`, that all subclasses must implement. As an abstract class, it is not intended to engage in auction conversations by itself. It does, however, implement some general functionality that the agents built upon it all must share. This functionality allows its child agents to engage in the conversations associated with the following actions:

`make_auction_cd`: This action is an immediate child of `execute`. It is made distinct from the `execute` action so that when the `make_auction_cd` conversation concludes,

[3]I.e., exchanging simulated currency.

[4]I.e., `EnglishAuctionAgent`, `DutchAuctionAgent`, `VickreyAuctionAgent`, and `SealedBidAuctionAgent`

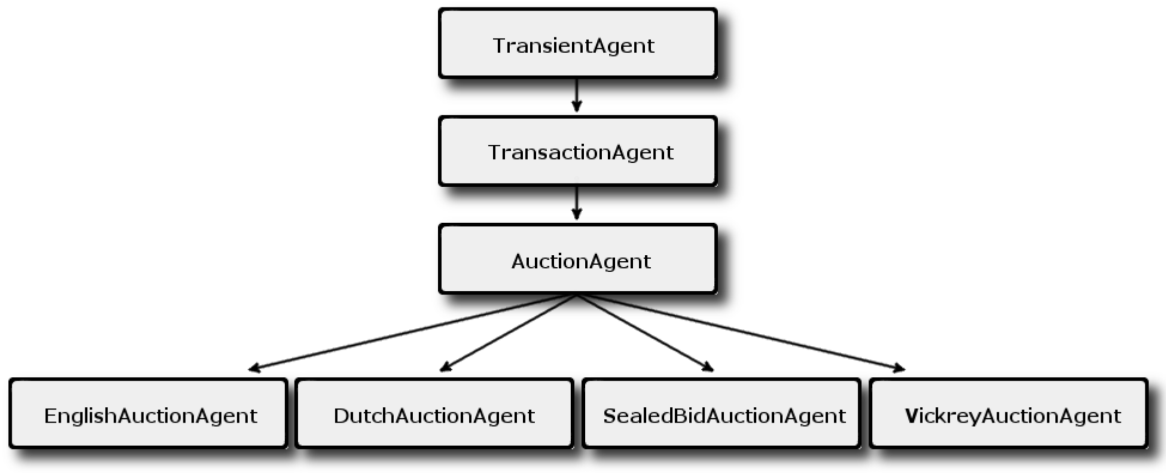


Figure 5.1: The `AuctionAgent` and its parent and child classes.

the associated `MessageEvent` is distinguishable from other `execute` conversations. There is more on auction CDs in Section 5.2.3.

`cry`**:** This action is an immediate child of `offer_to_sell`. When the auctioneer *cries* the bid, it is sending a *propose* to all interested agents that they would allow it to perform the `offer_to_sell` action. Naturally, many agents may agree to the auctioneer's proposal, but only the highest bidder will be chosen to engage in the `offer_to_sell` conversation to its proper conclusion.

`auction_starting`**:** This is an action sent via a *inform* message. It does not necessitate that either the sending or receiving agents engage in a first-order conversation. It simply conveys information (e.g., the auction is starting) that an agent may wish to consider. If an agent is interested in the item up for sale, it will join the CD identified in the message and wait for the auction to commence.

`auction_is_over`**:** This is an action sent via a *inform* message. As with `auction_starting`, it does not necessitate that either the sending or receiving agents engage in a first-order conversation. It simply informs the receiving agent that the auction in which they were participating is now over and that they should withdraw their membership from the auction CD.

Though the `AuctionAgent` is not equipped to engage in any particular auction without extending its functionality, it sets some general guidelines for how its child classes handle auction-related data. Just as `AuctionAgent`s extend `TransactionAgent`s, the items they auction, `Lot`s, extend the `TransactionAgent`'s `Product` class. `Lot` objects track data necessary to conduct an auction, such as the current bid and bidder. When an auctioneer agent *cries* for bids and bidders respond, serialized `Lot` objects are being passed within the messages they exchange.

The `AuctionAgent` class also sets guidelines for how its child classes are to initiate an auction. No matter the auction type, an `AuctionAgent` will always engage in a `request make_auction_cd` conversation with itself. Recall that the `make_auction_cd` conversation is a child of the `execute` conversation. Any agent that extends the `TransientAgent` class is equipped to `execute` the command necessary to instantiate a CD. The `AuctionAgent` simply sends the `request` to do so to itself. Once an auction CD has been instantiated, it engages in a `request get_agents_running` conversation with the `LAC`. When the `LAC` successfully replies with a list of known running agents, the `AuctionAgent` sends an `notify auction_starting` message containing the `Lot` up for sale[5]. Interested agents are given time to have `request join_cd` conversations with the auction CD, after which the auction begins. These steps are taken by all `AuctionAgent` child classes in order to initiate an auction.

Successful auction initiation depends entirely on the `AuctionAgent`'s ability to observe and handle `Events`. Each of the conversations mentioned in the previous paragraph must be executed in the order prescribed. That is, in order of execution:

1. `request make_auction_cd`

[5]The `Lot` also contains the URL of the auction CD

2. `request get_agents_running`

3. `inform auction_starting`

4. `request join_cd`

The `AuctionAgent` waits for each conversation to successfully conclude before engaging in the next. This is achieved with CASA's `MessageObserverEvent`, which executes instructions every time a given event occurs. For example, when the `LAC` has successfully provided the `AuctionAgent` a list of running agents (resulting from the `request get_agents_running` conversation), the associated `Event` triggers the instructions to send `inform auction_starting` messages to all the agents running. This could not happen unless the previous conversation had concluded successfully.

Once an auction has been initiated in accordance with steps previously outlined, each `AuctionAgent` subclass waits for the auctioneer to join the auction CD. `AuctionAgent` first sends the `inform auction_starting` to all of the MAS's running agents, and then sends the same message to itself. Children of `AuctionAgent` class wait for this event to successfully take place before bidding commences. The `AuctionAgent` class does not provide any of the functionality required to follow what is called the `auctioneerJoinedCDEvent`, but instead leaves its implementation to sub-classed agents through the abstract `startAuction` method.

5.2.3 Auction CDs

`CooperationDomain`s play an important role in the CASA MASs constructed for the purpose of simulating auctions. Their purpose is to identify the agents participating in the auction taking place. If an agent is interested in purchasing the product up for auction, it will join the designated CD. If it is not interested, or if the bidding goes beyond what it can afford, it will exit the CD. The auctioneer will only send messages to CD members.

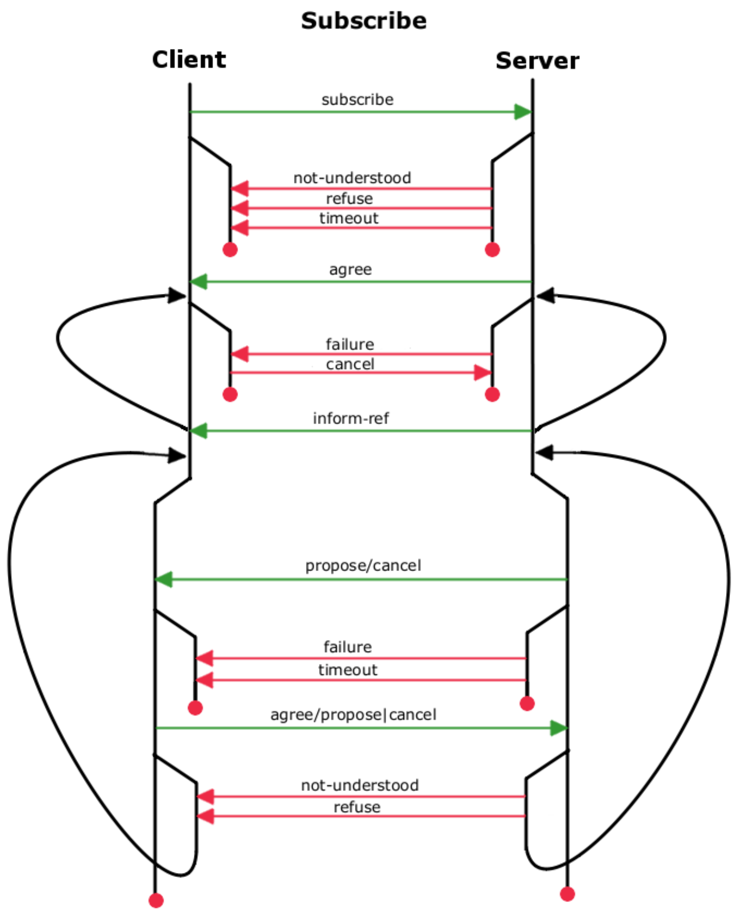


Figure 5.2: The CASA *subscribe* protocol.

The `CooperationDomain`s instantiated for the auctions are no different from the CDs described in Section 4.1. However, their inclusion as part of the auction MASs constructed here presented an opportunity to implement a new conversation template: *subscribe*. Part of the reason an agent will request membership in a CD is so that it may establish contact with other member agents. As such, the CD tracks membership and informs subscribing agents of any changes as they occur via an *inform-ref* message. A *subscribe* message sent from an agent to a CD creates a persistent social commitment that obligates the CD to send an *inform-ref* message containing its membership roster every time an agent joins or leaves. As with CASA's *propose* and *request* conversations, a *subscribe* and all the *inform-ref* `MessageEvent`s subsequent to its execution take place within the context of a first-order conversation.

The performative messages exchanged between a client and server in a *subscribe* conversation is illustrated in Figure 5.2. The conversation begins when a client sends a `subscribe` message to a server, which asks the server to inform the client every time the `Event` specified in the message's content field is triggered. As with the protocols illustrated in Figures 4.1 and 4.2, the server may respond in such a way that the conversation will terminate immediately, or it may send an `agree` message committing it to send an `inform-ref` message to the client every time the `Event` specified occurs. The commitment formed in a `subscribe` conversation is fulfilled when the client sends a message to `cancel`, the server sends a `propose/cancel`, or if the server sends a `failure`.

The following listing demonstrates the manner in which a server-side *subscribe* conversation is implemented:

```
(subscription-provider "membership_change"
 "event_CDMembershipChange"
 `("event_joinCD" "event_withdrawCD")
 `(jcall
    (jmethod (agent.get-class-name)
      "evaluate_membership_change" "casa.MLMessage")
      agent (event.get-msg)
    )
 `(jcall
    (jmethod (agent.get-class-name)
      "perform_get_members" "casa.MLMessage")
      agent NIL
    )
 )
```

Here, the required parameters in order of occurrence are:

1. The act

2. The event
3. The watched event
4. The action to perform when the subscription is received
5. The action to perform when the watched event occurs (send an *inform-ref* message containing the information for which the client subscribed).

The following listing demonstrates the implementation of a client-side *subscribe* conversation.

```
(subscription-request "membership_change"
 `()
 :inform-action
 `(jcall
    (jmethod (agent.get-class-name)
      "release_get_members" "casa.MLMessage")
      agent (event.get-msg)
    )
 :terminate-action
 `(
    (sc.cancel
      :Debtor server
      :Creditor client
      :Performative inform-ref
      :Act (act membership_change)
      )
    (conversation.set-state "terminated")
    )
 )
```

Here, the act is the only required parameter. The keyed parameters dictate what the agent will do with the information contained in the *inform-ref* messages it will receive as a result of subscribing, and what it should do when it cancels its subscription.

5.2.4 English Auction

The `EnglishAuctionAgent` extends the functionality established in the `AuctionAgent` class. The most notable of these extensions are found in its implementation of the `startAuction` method, the extension of the `perform_offer_to_sell` method, and the addition of the `i_hear` action to its ontology. These extensions are described in this section.

As described in Section 5.2.2, the `auctioneerJoinedCDEvent` must take place before the auctioneer can call for bids. The instructions this event triggers are tailored to the type of auction being implemented. As such, a child of the `AuctionAgent` class declares the instructions triggered by this event in its own way via the abstract `Auction-Agent.startAuction` method.

When the agent acting as the English auctioneer triggers the `auctioneerJoinedCD-Event`, the `startAuction` method sends the first round of `propose cry` messages containing the `Lot` currently on the auction block to all of the CD's member agents. This instantiates an `offer_to_sell` conversation in every receiving bidder. If a bidder is willing and able to pay the `Lot`'s current asking price, it replies to the auctioneer's `propose cry` message with a message containing an `agree` performative. If a bidder is not able to pay or not interested in the `Lot` being offered, it `refuse`s the auctioneer's `propose` message and leaves the CD. The incoming `agree` messages are processed in the order in which they are received by the auctioneer's `perform_offer_to_sell` method.

The `perform_offer_to_sell` method performs five functions:

1. Determine if an incoming bid is higher than the previous high bid, if one exists.
2. Send a `failure discharge|perform|cry` message to the bidder with the previous high bid, if one exists. This fulfils the social commitments created by the `agree` message sent by the bidder in response to the auctioneer's initiating `propose cry`

message.

3. Send an `inform i_hear` message notifying all bidders of the current high bid.

4. Send a new round of `propose cry` messages to call for new bids.

5. Finalize the financial transaction when the highest bid has been received and no new bids are forthcoming.

The `i_hear` action was added to the `EnglishAuctionAgent`'s ontology to provide a way for the auctioneer to `inform` the bidding agents of the bid currently being considered. The `inform i_hear` message is sent to all bidders, including the successful one, so that they may each consider whether to continue participating in the auction. The successful bidder is informed so that it knows not to submit a bid when the next `propose cry` message is received. If a higher bid is received as a result of the auctioneer's `cry`, the subsequent `inform i_hear` message tells the previous successful bidder that its bid is no longer being considered and that it should submit a new bid on the auctioneer's next `cry` if it is able to do so.

The English auction is carried forward by rounds of `propose cry` and `inform i_hear` messages. The price of the `Lot` is increased with each new round of bidding. Whenever the auctioneer receives an `agree propose|cry` message, which means the bidder is willing to pay at least the auctioneer's current asking price, the bid contained therein, if sufficient, is recorded in the `Lot` object and all bidders are apprised of the bid currently being considered via an `inform i_hear` message. Meanwhile, the `agree propose|cry` message containing the current successful bid is saved by the auctioneer and the action required by the resulting social commitment is *dropped*[6]. The saved `agree propose|cry` message is

[6]It is important to note that when an action is *dropped* its associated social commitment remains unfulfilled. In this case, the `perform_offer_to_sell` method simply returns a value that instructs the agent to refrain from performing the action necessary to fulfil the commitment. The intention is for this action to be performed at a later time.

reconsidered at a later time to determine if it is still the highest bid and if any new bids have been received. This is accomplished via a `RecurringTimeEvent`, which executes instructions at intervals set by the agent designer.

The `EnglishAuctionAgent`'s `RecurringTimeEvent` simply passes the saved `agree propose|cry` message containing the current high bid to the `perform_offer_to_sell` method at regular intervals. When the `Lot`'s price is sufficiently high, bidders will cease submitting bids and drop out of the auction. The high bid that remains is passed to the method three times[7]. This allows any remaining agents a chance to submit a new bid, at which point the current bid would be discarded and the auction would continue. If the successful bid passes three times, however, the auctioneer will then continue on in the `offer_to_sell` transaction initiated by `cry` conversation. The transaction continues as it would with the `TransactionAgent`'s `offer_to_sell` conversation. Once the transaction has been finalized, the auction ends, the auction participants leave the auction CD, and the auction CD closes.

5.2.5 Dutch Auction

The `DutchAuctionAgent` extends the functionality established in the `AuctionAgent` class in two ways: in its implementation of the abstract `AuctionAgent.startAuction` method, which monitors a new auction event called the `RecurringCryEvent`.

As with all `AuctionAgent` child classes, the `DutchAuctionAgent` auctioneer is required to watch for the `auctioneerJoinedCDEvent` before it may call for bids. Here, the `startAuction` method call triggered by this event set the `Lot`'s asking price much higher than its actual value, as perceived by the auctioneer agent. The `startAuction` method then starts the `RecurringCryEvent`, which triggers at timed intervals set by the agent designers.

[7]As with, *going once... going twice... gone!*

Once the `Lot`'s initial price has been established and the `RecurringCryEvent` started, the auctioneer sends `propose cry` messages containing the `Lot` to all of the bidders participating in the auction. The auction ends as soon as the auctioneer receives `agree propose cry` message from one of the bidders. If no bids are received the `RecurringCry-Event` triggers instructions to lower the `Lot`'s asking price slightly and send another round of `propose cry` messages. This continues until an `agree propose cry` message is received by a bidder or until the `Lot`'s price drops below that of auctioneer's reserve price.

If the auctioneer receives an `agree propose cry` message, it first sends `inform auction_is_over` messages to the auction CD's member agents. Then the `offer_to_sell` transaction initiated by its extending `cry` conversation proceeds between the auctioneer and the successful bidder. If no `agree propose cry` message is received and the `Lot`'s price drops below the auctioneer's reserve price, the auctioneer sends `inform auction_is_over` messages and the auction ends. In both cases the receiving agents withdraw their membership from the CD and the CD closes.

5.2.6 Sealed Bid Auction

The `SealedBidAuctionAgent` is among the simplest of agents implemented here. Much of its required functionality is already established in the abstract `AuctionAgent` class. It did not require the addition of any new actions to its ontology, as the majority of extensions and modifications were made to its `handleEvent` method and in its implementation of the abstract `Auctiongent.startAuction` method. Similarly, its implementation of the `consider_offer_to_sell` and `perform_offer_to_sell` methods first implemented in the `AbstractTransactionAgent` class are minor. Their re-implementation focus on the expanded functionality required to handle the data encapsulated within `Lot` objects, which extend the functionality of the `TransactionAgent`'s `Product` class. Aside from these minor modifications, which will be outlined in this section, the implementation of the `SealedBidAuctionAgent` required relatively little customization.

As with the `EnglishAuctionAgent`, the `SealedBidAuctionAgent` requeues `MessageEvent`s while it waits for *agree* messages to arrive in response to the `propose cry` conversations it initiates with bidding agents. When an *agree* message is received, the details of the bid it contains in its `content` field are stored for later consideration and the `MessageEvent` is requeued. A `MessageEvent` that has been seen previously will be ignored on subsequent passed through the `Event` queue. When the time allotted for accepting bids has passed, the agent with the highest bid is identified and the `MessageEvent`s that were previously ignored are then processed normally by CASA's `TransientAgent.handleEvent` super method. In applying the policies contained within the first-order conversations already established for auction purposes, the transaction commences with the winning bidder and all lesser bids are refused.

As with the other `AuctionAgent` subclasses, the implementation of the abstract `startAuction` method sends out the auctioneer's `propose cry` messages to all of the auction CD's member agents. The `SealedBidAuctionAgent` then waits a predetermined amount of time for the receiving agents to submit their bids via *agree* messages. When the time elapses, the winning bidder is identified and all of the auction CD's member agents are sent `inform auction_is_over` messages so that they know to withdraw their membership. The `TransientAgent.handleEvent` then processes *agree* messages as described in the previous paragraph.

The `SealedBidAuctionAgent`'s `consider_offer_to_sell` and `perform_offer_to_sell` methods override those defined in the `AbstractTransactionAgent` class, though the differences between each respective method are minor. Unlike the buyer `TransactionAgent`, which will only considers if it is willing and able to pay the seller's asking price, the `SealedBidAuctionAgent` bidder is made aware of the `Lot`'s reserve price and will submit a bid according to what it believes the `Lot` is worth and what it can afford. Similar modifications were necessary to enable the `SealedBidAuctionAgent` auctioneer to sell an item for a price different than that recorded in its `Inventory`. The overridden methods are only different insofar as how they accommodate `Lot` objects as opposed to the `Product` objects handled by the `TransactionAgent` ancestor class.

5.2.7 Vickrey Auction

The `VickreyAuctionAgent` is an immediate descendant of the `SealedBidAuctionAgent`. In order to simulate a Vickrey auction, it only needed to implement the abstract `AuctionAgent.startAuction` method and override the `perform_offer_to_sell` method. As with the changes made to `SealedBidAuctionAgent` and its parent class, `VickreyAuctionAgent`'s modifications are few.

`VickreyAuctionAgent`'s implementation of the `AuctionAgent.startAuction` method is only different from that of the `SealedBidAuctionAgent` in one way. While both agent classes designate the bidder with the highest bid as the winner, the `VickreyAuctionAgent` auctioneer sells the lot to the winning bidder for the price bid by the runner up agent. That is, if *Agent A* bids \$50 for a `Lot` and *Agent B* bids \$45, the auctioneer will sell the lot to *Agent A* for \$45. Each agent class's implementation of the `AuctionAgent.startAuction` method is the same except for final payment required of the winning bidder.

The `VickreyAuctionAgent` auctioneer's overridden `perform_offer_to_sell` method differs from that of its parent only in how it treats the winning bidder's *agree* message sent in reply to the auctioneer's initial `propose cry` message. Whereas the `SealedBidAuctionAgent` auctioneer would simply accept the price set by the winning bidder, the `VickreyAuctionAgent` auctioneer updates the `content` field of the winning bidder's message, which contains the serialized `Lot` object, to reflect the price set by the second-place bidder. As with previous `AuctionAgents`, the transaction between the winning bidder and the auctioneer proceeds while all other bids are refused.

This section described secondary conversations needed to simulate the English, Dutch, Sealed Bid, and Vickrey auction types. Specialized agent classes were designed to engage in each corresponding conversations. These steps were taken in order to fulfil Requirement I3 (Deploy secondary conversations).

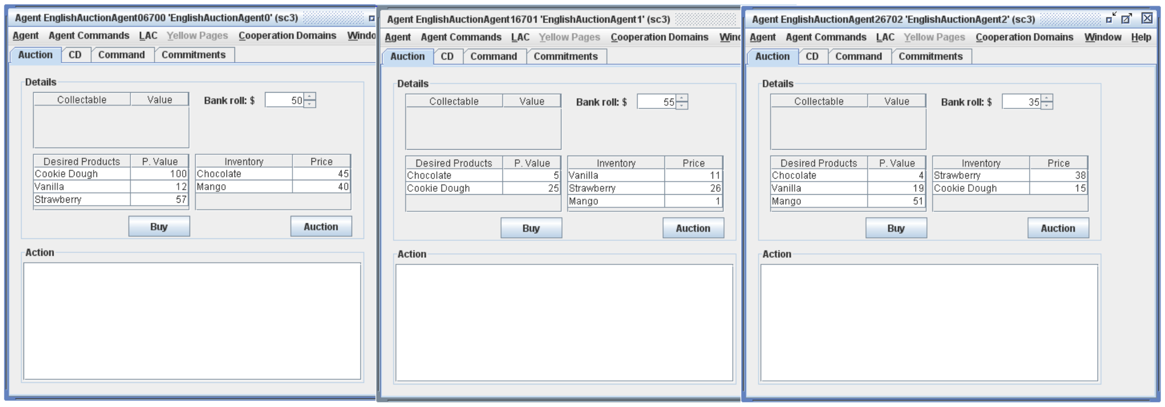
5.3 Verification

This section demonstrates that the first-order conversational mechanism and the conversations created from the templates implemented in Chapter 4 function according to the purposes for which they are designed. It will also be shown that the agents that engage in the conversations implemented in this chapter are left with no unfulfilled social commitments when the conversations conclude. These steps are taken in order to fulfil Requirement I4 (Verify agent behaviour and conversation termination).

The conversations and agents necessary to simulate an English Auction will be examined here. These components were selected for investigative purposes because the English Auction is the most complex of all the auctions implemented. And, as with all of the agents and conversations implemented in this chapter, they extend the functionality of their parent classes and conversations, so a good portion of the secondary conversations described here will be represented. The `EnglishAuctionAgents`' functionality will be verified first. After that, it will be shown that the agents participating in the English Auction simulation have no unfulfilled social commitments.

5.3.1 Functionality

The simplest way to verify that the `EnglishAuctionAgents` function in accordance with their design is to run an English auction and examine the output produced by the GUI shown in Figure 4.5. For this verification, three `EnglishAuctionAgents` were instantiated and initialized with unique inventories, lists of desired products, and available currency. For the purposes of this demonstration, `EnglishAuctionAgent2` will auction the *cookie dough* in its inventory. As shown in Figure 5.3, *cookie dough* is a product both `EnglishAuctionAgent0` and `EnglishAuctionAgent1` desire and can afford.


Agent EnglishAuctionAgent06700 'EnglishAuctionAgent0' (sc3)

Bank roll: $ 50

Collectable	Value

Desired Products	P. Value
Cookie Dough	100
Vanilla	12
Strawberry	57

Inventory	Price
Chocolate	45
Mango	40

Agent EnglishAuctionAgent16701 'EnglishAuctionAgent1' (sc3)

Bank roll: $ 55

Collectable	Value

Desired Products	P. Value
Chocolate	5
Cookie Dough	25

Inventory	Price
Vanilla	11
Strawberry	26
Mango	1

Agent EnglishAuctionAgent26702 'EnglishAuctionAgent2' (sc3)

Bank roll: $ 35

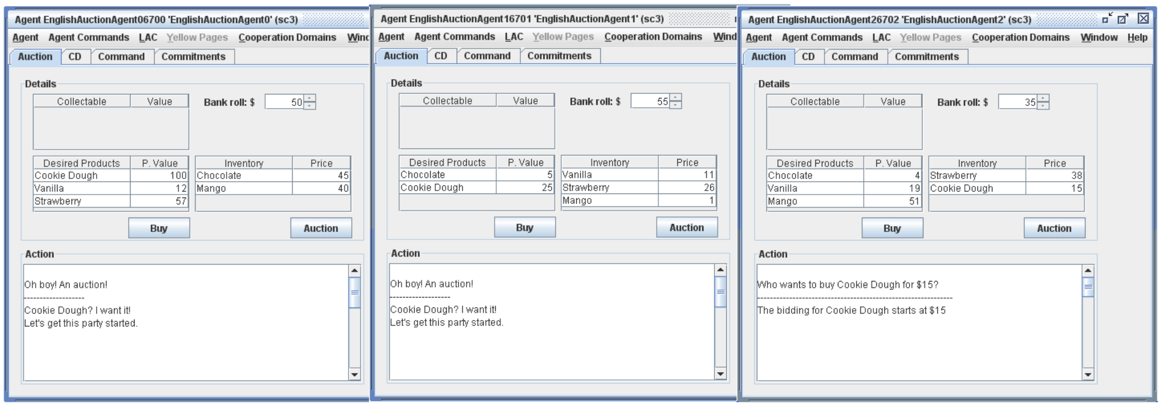
Collectable	Value

Desired Products	P. Value
Chocolate	4
Vanilla	19
Mango	51

Inventory	Price
Strawberry	38
Cookie Dough	15

Figure 5.3: The `EnglishAuctionAgents` prior to the start of the auction.

Figure 5.4 shows the `EnglishAuctionAgents` after the human observer has selected *cookie dough* from `EnglishAuctionAgent2`'s inventory and pressed the Auction button. The auctioneer has joined the auction CD and sent the first round of `propose cry` messages to the CD's member agents. The starting bid is \$15 for *cookie dough*.


Agent EnglishAuctionAgent06700 'EnglishAuctionAgent0' (sc3)

Bank roll: $ 50

Desired Products	P. Value
Cookie Dough	100
Vanilla	12
Strawberry	57

Inventory	Price
Chocolate	45
Mango	40

```
Oh boy! An auction!
------------------
Cookie Dough? I want it!
Let's get this party started.
```

Agent EnglishAuctionAgent16701 'EnglishAuctionAgent1' (sc3)

Bank roll: $ 55

Desired Products	P. Value
Chocolate	5
Cookie Dough	25

Inventory	Price
Vanilla	11
Strawberry	26
Mango	1

```
Oh boy! An auction!
------------------
Cookie Dough? I want it!
Let's get this party started.
```

Agent EnglishAuctionAgent26702 'EnglishAuctionAgent2' (sc3)

Bank roll: $ 35

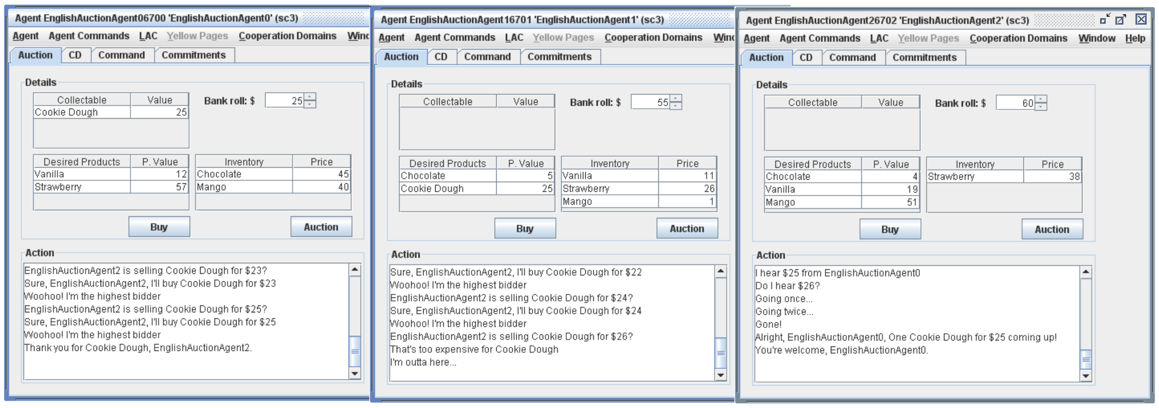
Desired Products	P. Value
Chocolate	4
Vanilla	19
Mango	51

Inventory	Price
Strawberry	38
Cookie Dough	15

```
Who wants to buy Cookie Dough for $15?
------------------------------------------------------------
The bidding for Cookie Dough starts at $15
```

Figure 5.4: The `EnglishAuctionAgents` at the start of the auction.

Finally, Figure 5.5 shows the agents after the auction has finished. **`EnglishAuction-Agent2`** now has \$25 more in its bank roll and no longer has *cookie dough* in its inventory. **`EnglishAuctionAgent0`**, which submitted the highest bid, now has \$25 less in its bank roll and lists *cookie dough* as one of its collectables. Collectively, the figures presented in this section demonstrate that the **`EnglishAuctionAgent`**s function in accordance with their design.



Agent EnglishAuctionAgent06700 'EnglishAuctionAgent0' (sc3)

Bank roll: $ 25

Collectable	Value
Cookie Dough	25

Desired Products	P. Value
Vanilla	12
Strawberry	57

Inventory	Price
Chocolate	45
Mango	40

```
EnglishAuctionAgent2 is selling Cookie Dough for $23?
Sure, EnglishAuctionAgent2, I'll buy Cookie Dough for $23
Woohoo! I'm the highest bidder
EnglishAuctionAgent2 is selling Cookie Dough for $25?
Sure, EnglishAuctionAgent2, I'll buy Cookie Dough for $25
Woohoo! I'm the highest bidder
Thank you for Cookie Dough, EnglishAuctionAgent2.
```

Agent EnglishAuctionAgent16701 'EnglishAuctionAgent1' (sc3)

Bank roll: $ 55

Collectable	Value

Desired Products	P. Value
Chocolate	5
Cookie Dough	25

Inventory	Price
Vanilla	11
Strawberry	26
Mango	1

```
Sure, EnglishAuctionAgent2, I'll buy Cookie Dough for $22
Woohoo! I'm the highest bidder
EnglishAuctionAgent2 is selling Cookie Dough for $24?
Sure, EnglishAuctionAgent2, I'll buy Cookie Dough for $24
Woohoo! I'm the highest bidder
EnglishAuctionAgent2 is selling Cookie Dough for $26?
That's too expensive for Cookie Dough
I'm outta here...
```

Agent EnglishAuctionAgent26702 'EnglishAuctionAgent2' (sc3)

Bank roll: $ 60

Collectable	Value

Desired Products	P. Value
Chocolate	4
Vanilla	19
Mango	51

Inventory	Price
Strawberry	38

```
I hear $25 from EnglishAuctionAgent0
Do I hear $26?
Going once...
Going twice...
Gone!
Alright, EnglishAuctionAgent0, One Cookie Dough for $25 coming up!
You're welcome, EnglishAuctionAgent0.
```

Figure 5.5: The `EnglishAuctionAgent`s after auction is complete.

5.3.2 Conversation Termination

Further to the example provided in Section 5.3.1, this section provides an accounting of the conversations that take place in order to form a MAS and to conduct the English auction described. The conversations will be categorized as *primary* and *secondary* and presented in the order in which they are executed. Once the various conversations have been identified, it will be demonstrated that every social commitment formed over their execution is fulfilled.

CASA tracks every social commitment formed over the course of every agent conversation. What follows is an actual accounting of these social commitments. It has been verified in practice that no outstanding social commitments remain at the end of all auctions and communicative interactions. This is true of every participating agent.

The *primary* conversations necessary to establish a CASA MAS are as follows:

`request register_instance`: sent by the LAC to itself so as to include it in its own catalogue of running agents.

`request execute`: sent by the `casa` bootstrap agent to the LAC to instantiate the society's agents. These conversations do not necessarily need to take place at start up, but are identified here because it is assumed that a MAS will host multiple agents and, as such, the conversations will take place at some point.

`request register_instance`: sent by the agents in the MAS to the LAC so that they are included in the LAC's catalogue of running agents.

The *secondary* conversations necessary to initiate an English auction are as follows:

`request execute`: sent by the `casa` bootstrap agent to the society's various auction agents so as to initialize their internal data. In the case of an auction agent, this data includes its inventory, desires, and bank roll. Although the `request execute` conversation is a *primary* conversation, it is listed here because it initializes internal agent data that serves a secondary purpose.

`request make_auction_cd`: sent by the English auctioneer to itself to create a CD to track interest in the auction. Only bidders interested in the `Lot` on the block join this CD.

`request get_agents_running`: sent by the English auctioneer to the LAC to retrieve a list of all the agents currently running in the society.

`inform auction_starting`: sent by the English auctioneer to all agents running in the LAC to inform them of the `Lot` currently on the block. If a bidder wants to purchase the `Lot` it will initiate a `request join_cd` conversation with the auction CD.

`request join_cd`: sent by interested English bidders and the auctioneer to the auction CD to request membership.

`subscribe membership_change`: sent by English bidders and the auctioneer to the auction CD. Whenever an agent joins or leaves the CD, the CD will send an `inform-ref membership_change` message notifying current members of the event.

`request get_members`: sent by English bidders and the auctioneer to the auction CD to initialize CD member lists.

The *secondary* conversations that take place over the course of the English auction are as follows:

`propose cry`: sent by the English auctioneer to the bidders. This is an extended `propose offer_to_sell` conversation, which invites interested bidders to purchase the `Lot` on the block.

`inform i_hear`: sent by the English auctioneer to the bidders. This informs the bidders of the bid currently being considered.

`inform withdraw_cd`: sent by English bidders and the auctioneer to the auction CD once the auction is over. This is preceded by a `cancel subscribe|membership_change`, which cancels the subscription made previously. Once all member agents have withdrawn, the CD closes.

Of the conversations listed, it is easiest to verify the fulfillment of all social commitments formed over the course of the simple `request`-type conversations. The `subscribe` conversation is also fairly easy to verify. It is more challenging to demonstrate the fulfillment of all social commitments formed over the repeating cycles of `propose cry` conversations specific to the English auction. The difficulty arises because of the need to *drop* actions while incoming bids are being considered (see Section 5.2.4). So, while all the social commitments formed over the course of a `propose cry` conversation are fulfilled, they are not necessarily fulfilled before a new `propose cry` is initiated. An accounting of each conversation's creation and fulfilment of social commitments follows.

`request` Conversation Verification

All of the conversations listed in this section are created using templates (see Section 4.3.2). As such, any `request`-type conversations created with the `request` template will treat social commitments the same way. Table 5.1 provides an accounting of all of the social commitments formed and fulfilled over the course of a `request register_instance` conversation between an `AuctionAgent` and the LAC.

`request register_instance`

`AuctionAgent` SCs	Message	`LAC` SCs
C: `reply request|...`	`request register_instance` ⟹	D: `reply request|...`
~~C: `reply request|...`~~ C: `propose discharge|...`	⟸ `agree request|register_instance`	~~D: `reply request|...`~~ D: `perform register_instance` D: `propose discharge|...`
~~C: `propose discharge|...`~~ D: `reply propose|...`	⟸ `propose discharge|perform|register_instance`	~~D: `propose discharge|...`~~ C: `reply propose|...` D: `perform register_instance`
~~D: `reply propose|...`~~	`agree propose|...|...|register_instance` ⟹	~~C: `reply propose|...`~~ ~~D: `perform register_instance`~~

Table 5.1: The formation and fulfillment of social commitments during a `request register_instance` conversation between an `AuctionAgent` and the LAC.

Table 5.1 consists of three columns, which account for the social commitments created and fulfilled by the LAC and `AuctionAgent`, and the messages passed between them. In the *SC* columns, a *C* prefix denotes that this agent is a *creditor* in the formation of that commitment. The *debtor* agent has a matching commitment prefixed with a *D*. When either agent sends or receives a message, as denoted by the directional arrows, a social commitment will be formed or fulfilled as a result. A fulfilled commitment is *struck through*, while an unfulfilled commitment has no such marking[8]. As depicted in

[8]These conventions hold for all of the social commitment accounting tables found in this section.

Table 5.1, every commitment formed has a matching commitment that has been *struck through*. No commitments have been left unfulfilled. Thus, having demonstrated that `request register_instance` conversation leaves no commitments unfulfilled, it can be assumed that all conversations created from the `request` template behave similarly.

`subscribe` Conversation Verification

Conversations constructed from the `subscribe` template leave a *persistent* social commitment. This commitment remains until the subscriber tells the subscription provider to `cancel` the subscription. Table 5.2 provides an accounting of all the social commitments formed and fulfilled over the course of a `subscribe membership_change` conversation between an `AuctionAgent` and an auction CD.

`subscribe membership_change`

`AuctionAgent` SCs	Message	Auction `CD` SCs
C: `reply subscribe|...`	`subscribe membership_change` ⟹	D: `reply subscribe|...`
~~C: `reply subscribe|...`~~ C: `inform-ref membership_change`	⟸ `agree subscribe|membership_change`	~~D: `reply subscribe|...`~~ D: `inform-ref membership_change`
	...	
	⟸ `inform-ref membership_change`	
	...	
~~C: `inform-ref membership_change`~~	`cancel subscribe|membership_change` ⟹	~~D: `inform-ref membership_change`~~

Table 5.2: The formation and fulfillment of social commitments during a `subscribe membership_change` conversation between an `AuctionAgent` and an auction CD.

The `subscribe membership_change` conversation, as depicted in Table 5.2, creates a commitment for the auction CD to inform subscribed `AuctionAgents` every time an agent joins or leaves, which is the basic functionality expected of a CASA CD. The CD membership list is delivered via an `inform-ref membership_change` message. This commitment remains until one side `cancel`s the subscription. In this case, the `AuctionAgent` fulfilled the CD's commitment by sending the `cancel subscribe|membership_change` message. Conversations created from the `subscribe` template leave no unfulfilled social commitments at their conclusion.

`propose cry` Conversation Verification

`propose`-type conversations are created using the same template as the `request`-type conversations (see Section 4.3.2). The `propose cry` conversation, as implemented for use by the `EnglishAuctionAgent` class, is no exception. It is, however, unique in that certain actions are *dropped* to allow the auctioneer to consider all incoming bids (see Section 5.2.4). This makes accounting for the social commitments formed over the course of repeated bidding cycles more of a challenge, given the number of `propose cry` conversations involved and the unpredictable order in which commitments are fulfilled. As will be shown, a commitment from a previous conversation may remain temporarily unfulfilled even though a new conversation has started. Given the context provided by first-order conversations, however, `EnglishAuctionAgent`s still function according to their design and all commitments do get fulfilled.

The tables presented in this section account for the social commitments formed between a bidder and an auctioneer participating in a short English auction. In these examples there were two bidders participating, but only the winning bidder is included in the accounting. It took six cycles to determine the highest bid, which required six `propose cry` conversations to be conducted between the auctioneer and each bidder.

Table 5.3 presents an accounting of the social commitments formed between the bidder and auctioneer in the first cycle of an English auction. The commitments are numbered in the order they are formed so as to provide references between subsequent tables. Here, the auctioneer starts a `propose cry` conversation with both of the interested bidders. The bidder presented in the table replies with an `agree propose|cry` message indicating that it wishes to purchase the `Lot`. This round of bidding ends with an `inform i_hear` message from the auctioneer, which tells the participating bidders which bid is currently being considered. Three social commitments pertaining to the completion of the financial transaction remain unfulfilled.

propose cry bidding cycle #1

English Bidder SCs	Message	English Auctioneer SCs
2 D: reply propose|cry	⟸ propose cry	1 C: reply propose|cry
~~2 D: reply propose|cry~~ 3 C: propose discharge|perform|cry	agree propose|cry ⟹	~~1 C: reply propose|cry~~ 4 D: perform cry 5 D: propose discharge|perform|cry
	⟸ inform i_hear	

Table 5.3: The formation and fulfillment of social commitments between a bidder and auctioneer in the first round of bidding in an English auction.

Table 5.4 presents an accounting of the social commitments formed between the bidder and auctioneer in the second cycle of an English auction. Here, the bidder agrees to the auctioneers second call for bids even though there are still commitments remaining from the previous round of bidding. This means that its bid was not accepted the first time and was informed of this via the `inform i_hear` message. All three unresolved social commitments are fulfilled by the auctioneer's `failure` message and three new ones are formed by the current `propose cry` conversation. In this case the commitment to `perform cry` is dropped because the bidder submitted the best bid. The auctioneer will save the best bid until a better bid arrives.

propose cry bidding cycle #2

English Bidder SCs	Message	English Auctioneer SCs
7 D: reply propose|cry	⟸ propose cry	6 C: reply propose|cry
~~7 D: reply propose|cry~~ 8 C: propose discharge|perform|cry	agree propose|cry ⟹	~~6 C: reply propose|cry~~ DROP → ~~9 D: perform cry~~ 10 D: propose discharge|perform|cry
~~3 C: propose discharge|perform|cry~~	⟸ failure discharge|perform|cry	~~4 D: perform cry~~ ~~5 D: propose discharge|perform|cry~~
	⟸ inform i_hear	

Table 5.4: The formation and fulfillment of social commitments between a bidder and auctioneer in the second round of bidding in an English auction.

Table 5.5 presents an accounting of the social commitments formed between the bidder and auctioneer in the third cycle of an English auction. Here the bidder sends a `refuse` message to the auctioneer's `propose cry` message, because its bid was the one accepted in the previous round. Since the auctioneer started `propose cry` conversations with both interested bidders, it received an acceptable bid from this bidder's competitor. As a result, the social commitments formed in the previous round of bidding are fulfilled by the auctioneer's `failure discharge|perform|cry`. At this point, there are no unfulfilled social commitments between the auctioneer and this bidder.

`propose cry` bidding cycle #3

English Bidder SCs	Message	English Auctioneer SCs
12 D: `reply propose|cry`	$\Longleftarrow$ `propose cry`	11 C: `reply propose|cry`
~~12 D: `reply propose|cry`~~	`refuse propose|cry` $\Longrightarrow$	~~11 C: `reply propose|cry`~~
~~8 C: `propose discharge|perform|cry`~~	$\Longleftarrow$ `failure discharge|perform|cry`	~~10 D: `propose discharge|perform|cry`~~
	$\Longleftarrow$ `inform i_hear`	

Table 5.5: The formation and fulfillment of social commitments between a bidder and auctioneer in the third round of bidding in an English auction.

The social commitments formed in the fourth round of bidding in the English auction are identical to those catalogued in Table 5.3. Similarly, the commitments formed in the fifth round of bidding are identical to those catalogued in Table 5.4. If the auction were extended[9], the patterns of creation and fulfillment would be repeated.

Table 5.6 presents an accounting of the social commitments formed between the bidder and auctioneer in the final cycle of an English auction. Two commitments remain from the previous round: the creditor and debtor commitments for the auctioneer to send a `propose discharge|perform|cry` message. The commitment to `perform cry` was *dropped* as it was in Table 5.4. The English auction arrives at this point when the auctioneer receives no new bids. It attempts to invite bids three times[10] before it finalizes the transaction with the winning bidder. The transaction is finalized when the auctioneer performs the previously dropped action and sends a `propose discharge|perform|cry` message to fulfil the associated commitment. The bidder agrees that the transaction was successful, so the conversation terminates and all social commitments are fulfilled.

[9]By attributing a greater perceived value for the `Lot` on the block or increasing each agent's bank roll, for example.

[10]As in, *Going once... going twice... sold!*

propose cry bidding cycle #6

English Bidder SCs	Message	English Auctioneer SCs
24 D: reply propose|cry	$\Longleftarrow$ propose cry	23 C: reply propose|cry
~~24 D: reply propose|cry~~	refuse propose|cry $\Longrightarrow$	~~23 C: reply propose|cry~~
26 D: reply propose|cry	$\Longleftarrow$ propose cry	25 C: reply propose|cry
~~26 D: reply propose|cry~~	refuse propose|cry $\Longrightarrow$	~~25 C: reply propose|cry~~
28 D: reply propose|cry	$\Longleftarrow$ propose cry	27 C: reply propose|cry
~~28 D: reply propose|cry~~	refuse propose|cry $\Longrightarrow$	~~27 C: reply propose|cry~~
~~22 C: propose discharge|perform|cry~~ 30 C: reply propose|...|cry	$\Leftarrow$ propose discharge|perform|cry	~~22 D: propose discharge|perform|cry~~ 29 C: reply propose|...|cry
~~30 C: reply propose|...|cry~~	agree propose|...|cry $\Longrightarrow$	~~29 C: reply propose|...|cry~~

Table 5.6: The formation and fulfillment of social commitments between a bidder and auctioneer in the sixth (final) round of bidding in an English auction.

This section demonstrated that the first-order conversational mechanism and the conversations created from the templates implemented in Chapter 4 function according to the purposes for which they are designed. It was also shown that the agents that engage in the conversations implemented in this chapter are left with no unfulfilled social commitments when the conversations conclude. These steps fulfil Requirement I4 (Verify agent behaviour and conversation termination).

5.4 CASA vs. JADE

JADE is a multi-agent platform comparable to CASA (JADE). It fulfills most of the broad requirements outlined in Requirement S*, but falls short in that it does not apply policies in the creation and fulfilment of social commitments. Though not useful in meeting the overall aim of this research, JADE is useful in providing a basis of comparison. This section describes the steps taken to implement a CASA-style `TransactionAgent` in JADE without the first-order conversational mode of interaction. This is done in order to fulfil Requirement I5 (Deploy a secondary conversation in a non-CASA MAS). The details of each implementation will provide the means through which the strengths and weaknesses of each system will be identified.

5.4.1 JADE TransactionAgent Design

Like CASA's `TransientAgent`, JADE provides a base `Agent` class. This class was extended to create the new JADE `TransactionAgent`. All of the functionality of CASA's `TransactionAgent` was replicated in JADE's version of the class, but the implementation was carried out while respecting JADE's limitations and features. JADE's limitations will become evident by way of comparison with CASA. Its most significant features, however, include the following:

1. The Directory Facilitator (DF) agent, which allows agents to advertise the services they provide.
2. The Call For Proposals (CFP) performative with accompanying protocol.

The JADE `TransactionAgent` incorporates the features listed above. Neither a DF-type agent nor a *CFP* conversation template are currently available in CASA. These are desirable features, and as such, were incorporated into the design of the JADE `TransactionAgent`. The overall functionality of the `TransactionAgent`s as implemented in both the CASA and JADE frameworks remain the same, but the transactions themselves are initiated in a different manner.

The differences between the CASA and JADE implementations are evident immediately upon the society's initialization when JADE `TransactionAgent`s register the services they perform with the DF agent. When a `TransactionAgent` receives instructions from the human observer to either buy or sell a product, it initiates the transaction by first querying the DF for an address list of agents advertising the relevant service. The addresses are inserted into the `recipient` field of a message containing a *CFP* performative alongside the `Product` of interest, which is inserted into the `content` field.

The initiating agent then sends the *CFP* performative message. Without the benefit of a DF-type agent, CASA `TransactionAgent`s simply send their offers to buy or sell to all of the agents registered with the LAC, regardless of whether they are equipped to participate in a transaction.

After the initiating `TransactionAgent` sends the *CFP* performative messages, the CASA and JADE implementations behave in similar ways. In both cases, interested recipients respond with messages containing the `Product`'s best price, be it a *high* price in the case of seller looking for a buyer, or a *low* price in the case of buyer looking for a seller. The initiating agent chooses the best price from among the offers received and the currency and product are exchanged.

The differences between the CASA and JADE implementations of the `TransactionAgent` described so far have little bearing on the aim of this research. It is necessary to describe them, however, in order to demonstrate that the JADE agents constructed here make best use of the framework's features. These features lend themselves to an agent design paradigm distinct from CASA's. The design paradigms are of primary interest here. Namely, CASA's new emphasis on first-order conversations as the preferred mode of agent communication versus JADE's method of implementing *complex* agent conversations.

5.4.2 JADE Conversations

By default, JADE conversations are motivated by the receiving of messages. CASA conversations, in comparison, are motivated by message exchanges. That is, CASA agents process all incoming and outgoing messages via the `TransientAgent.handleEvent` method. If the CASA agent designer connects an agent's method to the sending of a message in a conversation's template[11], that agent will execute the method specified whenever that

[11]Beyond the formation of a social commitment, which is dictated by the conversation template

message is sent. When a JADE agent sends a message, on the other hand, no processing beyond the physical sending of the message is required. During a conversation, a JADE agent's default behaviour is to only act when a message is received.

JADE's `Behaviour` class determines what actions an agent performs upon receipt of a message. `Behaviours` may be added to an agent's `Scheduler` any time during its execution. An agent typically pulls a message off of its `MessageQueue` inside the next `Behaviour` scheduled for execution. Queued messages may be filtered with a `MessageTemplate`, which can specify any number of parameters required to make a match, including the performative and ontology contained in that message. The first message that matches the template is returned to the calling `Behaviour` class and is removed from the agent's `MessageQueue`. In this way, an agent's `Behaviour` is motivated by messages selected off of its incoming `MessageQueue`.

A JADE agent's actions are directed by any number of `Behaviours` over the course of a conversation. It did, however, prove quite cumbersome to implement more than three such `Behaviours` in the case of the JADE `TransactionAgent`. These three `Behaviours` are as follows:

`TransactionCFPServer:` a continuously running `CyclicBehaviour` that waits for incoming messages containing *CFP* performatives. The ontological action contained therein may be either an `offer_to_buy` or `offer_to_sell`.

`PurchaseOrdersServer:` a continuously running `CyclicBehaviour` that waits for incoming messages containing an *agree* or *request* performative. This behaviour completes the transaction initiated by the `TransactionCFPServer` behaviour.

`TransactionPerformer:` this `Behaviour` is added to the `TransactionAgent`'s scheduler when the human observer prompts the agent to buy or sell a `Product`. It receives incoming offers and identifies the agent with the best price, which is then passed on to the `PurchaseOrdersServer` behaviour to complete the transaction.

The `Behaviours` described above are the result of the simplest and quickest implementation of the `TransactionAgent` in JADE. Despite the simplicity and the relatively small number of associated `Behaviours`, the implementation quickly became unwieldy. Much of the difficulty revolved around the `TransactionAgent`'s ability to initiate both the `propose offer_to_sell` and `request offer_to_buy` conversations (see Section 5.2.1). As described above, the functionality respective to both the `offer_to_buy` and `offer_to_sell` actions are contained in the same `CyclicBehaviour` classes, which need to run continuously in order to detect incoming messages. As well, the `TransactionCFPServer`, which by definition, should only call for *proposals*[12], was easily made to call for *requests*[13]. A discussion of these issues follows.

The problems raised by JADE's implementation of the `TransactionAgent` are not shared by the CASA implementation. While it may be the case that JADE's `CyclicBehaviours` could be organized to allow for a separation of the functionality required of `offer_to_buy` or `offer_to_sell` actions, this would have taken significantly more effort and time. And by modularizing functionality in that way, the complexity of the implementation increases, as `Behaviour` scheduling needs to be considered, and increasingly restrictive `MessageTemplates` need to be constructed for the various distinct `Behaviours` to identify and receive relevant incoming messages. The ease with which the *CFP* protocol was violated is also a matter of concern, as there appears to be no mechanism to enforce compliance. In fact, as the JADE framework is currently implemented, there is nothing to prevent the agent designer from substituting one performative in a message with any other. As long as an agent's `Behaviours` match the incoming message with the appropriate `MessageTemplate`, a performative can be made to take on any meaning.

[12]As in `propose offer_to_sell`
[13]As in `request offer_to_buy`

These problems are not known to CASA implementations for three simple reasons:

1. CASA's first-order conversational mechanism eliminates the need for multiple server-type `Behaviours` to run continuously in order to detect incoming messages.
2. CASA's first-order conversations provide context to incoming messages so `Message-Template`s need not be applied ad hoc.
3. CASA's conversation templates enforce adherence to the protocols they define.

Although the JADE framework has desirable features and makes simple MASs easy to deploy, it possesses no mechanism with which to combine the individual messages exchanged by its agents into complete, cohesive conversations. The `MessageTemplate`s used to identify messages relevant to a particular `Behaviour` require increasing specificity as the number of `Behaviour`s increases in a complex JADE conversation. Given that there is a finite number of `MessageTemplate` parameters with which to sort messages, there is a limit on the number of `Behaviour`s a JADE agent can schedule. This, in turn, implies a limit on the complexity of a JADE conversation. These limitations were revealed by implementing the CASA-style `TransactionAgent` in JADE, which fulfils Requirement I5 (Deploy a secondary conversation in a non-CASA MAS).

5.5 FIPA-SL and the General Purpose `Subscribe` Classes

CASA now provides a general purpose `subscribe` conversation, which makes use of the FIPA-SL[14]. This `subscribe` conversation has both server-side and client-side class definitions: `SubscribeServerConversation` and `SubscribeClientConversation`, respectively. Both of these classes extend CASA's `Conversation` class. The methods contained therein are connected to agent message exchanges using the `subscribe` template in the

[14]FIPA-SL may also be employed by agents designed for JADE (Bellifemine et al., 2007).

same manner described in Section 4.3.2. This section describes the `Subscribe` classes and their usage.

The use of `SubscribeClientConversation` and `SubscribeServerConversation` is best understood in terms of their relationship to the `subscribe` template. CASA templates establish a connection between the performative messages exchanged between agents and the internal considerations those agents make in response. The following code listing illustrates the connections made by a client agent that has subscribed to be notified of an event. This listing is found in the `casa.TransientAgent.init.lisp` file:

```
(subscription-request "-"
  `()
  :inform-action
  ;This establishes which function will be called when the client agent
  ;receives the data to which it subscribed
  `(jcall
    (jmethod
      "casa.conversation2.SubscribeClientConversation" "update_subscribe"
      "casa.MLMessage")
    conversation (event.get-msg)
    )
  ;Fulfil the pertinent social commitment when the subscription ends
  :terminate-action
    `(
      (sc.cancel
        :Debtor server
        :Creditor client
        :Performative inform-ref
        :Act (act membership_change)
        )
       (conversation.set-state "terminated")
     )
  :class "casa.conversation2.SubscribeClientConversation"
  :language "FIPA-SL"
  )
```

The server side of the general purpose `subscribe` conversation employs its template in a similar way:

```
(subscription-provider "-" "event" NIL
  '(jcall
    (jmethod "casa.conversation2.SubscribeServerConversation"
      "evaluate_subscribe"
      "casa.MLMessage")
     conversation (event.get-msg)
     )
    NIL
    :class "casa.conversation2.SubscribeServerConversation"
    :language "FIPA-SL"
  )
```

Obviously, it may be necessary to customize the considerations agents will make in accordance with the type of event to which they subscribe. These customizations occur internally within the agents' class definitions. To do this, client agents instantiate instances of the `SubscribeClientConversation` class. The class definition requires a parameter consisting of the the FIPA-SL expression, which expresses the event to which the agent wishes to subscribe. As part of the class's instantiation, a message is sent to the server agent[15], which binds the new object's unique methods to that particular subscribe conversation. This is possible because all conversations have a unique identifier attributed to them. In this way, the general purpose `subscribe` conversation may be customized in accordance with the type of event to which the agent subscribed.

The considerations agents make when engaged in a general purpose `subscribe` conversation are defined by overriding the `SubscribeClientConversation` class's `update` method. This occurs when a new object is instantiated, as illustrated by the following code listing:

```
SubscribeClientConversation convBump = new SubscribeClientConversation(
  "--subscription-request",
  this, server,
  "(all ?x (BumpsAndWheelDrops ?x))", null)
  {
    @Override
    public void update(URLDescriptor agentB, Term term) {
```

[15]The server agent's URL is also passed as a parameter to the `SubscribeClientConversation` class.

```
      if (term==null || demoOn())
        return;
      String intString = term.toString();
        int val = Integer.parseInt(intString);
        onBumpsAndWheelDrops(val);
      }
  };
```

When the `SubscribeClientConversation` object is instantiated, as shown above, the agent will execute the code defined in the overridden `update` method whenever it is notified that the subscribed event took place. The corresponding `SubscribeServer-Conversation` has no method that needs to be overridden. When a subscription request is received, the server agent simply determines if it is an observer of the event of which the client agent wants to be apprised. This is determined by executing the `evaluate_subscribe` method, which then informs the client agent of whether it is able to add it as a subscriber.

5.6 Summary

This chapter was motivated by Objective O5 (Analyze the System) and fulfils all of Requirements I*, which outline the expectations set for this investigation. It describes how the conversation templates implemented in Chapter 4 were used to create primary and secondary conversations. Primary conversations are those that are fundamental to the formation of a CASA MAS. The fundamental preexisting CASA protocols were redeployed using conversation templates. Using the same templates, several secondary conversations, which are dependent upon the establishment of an MAS, were created to simulate a financial transaction and several different auction types.

The secondary conversations implemented in this chapter required accompanying agent classes. These agent classes extended CASA's existing `TransientAgent` class, which was modified to give primacy to the conversational mode of communication whenever a `MessageEvent` is received in Chapter 4. The first child class, the `TransactionAgent`, was created to establish the behaviour necessary for agents to exchange *Product*s for currency. Upon this class, an abstract `AuctionAgent` was created to establish the agent behaviour appropriate to all of the auction-types of interest here. From there, behaviours specific to the various auctions were implemented in their associated `AuctionAgent` child classes.

Having implemented the `TransactionAgent` and various `AuctionAgent` classes, agent behaviour and conversation termination were verified. It was shown that the agents engaging in an English Auction simulation were left with no unfulfilled social commitments once the simulation had ended. The simulation was traced from initialization to conclusion so that both CASA's primary conversations and a good representation of secondary conversations were demonstrated. Complete log file listings for all of the agents involved are available in the Appendices.

Finally, CASA's `TransactionAgent` was reimplemented in JADE, which is a multi-agent framework that does not employ a first-order conversational mechanism. Though it was shown that JADE has some desirable features and is reasonably easy to use when implementing simple MASs, problems were revealed in its treatment of agent communication. Namely, the complexity of JADE agents' conversations is limited by the number of unique `MessageTemplate`s that can be applied in order to direct messages to the appropriate agent `Behaviour` class. And perhaps more problematic is that performatives contained within JADE agent messages can be made to take different or extended meanings. These problems are not shared by CASA, because the first-order conversational mechanism provides context to the messages passed between agents, so ad hoc message filtering is unnecessary. As well, CASA conversation templates respect performative semantics. Agent designers who utilize CASA conversation templates are denied the opportunity to deviate from the semantics established therein.

Chapter 6

Conclusion and Future Work

The work presented in this thesis contributes to the establishment of conversations as a first-order communicative mechanism in multi-agent systems. Agents built upon the CASA MAS were modified so that the messages exchanged between them are now processed within the context of a conversation. To achieve this, existing CASA communication protocols were examined and recurring sequences of performative messages were identified and modularized into reusable components called conversationlets. The conversationlets were used in the construction of conversation templates, which allow agent designers to easily establish the connection between agent behaviour in response to messages sent and received. When communication between agents is conducted under the auspices of a first-order conversation, complete, coherent agent dialogues are simple to construct.

The investigative component of this research, as described in Chapter 5, proceeded under the assumption that all of the agents involved were *well behaved*. It was beyond this research's scope to investigate the impact of *anti-social* agents; those that do not adhere to a society's policies, whether accidental or by design. Similarly, the agent interactions investigated were not subjected to the stresses of unstable communication infrastructure or measured against any security standard. As such, none of the messages exchanged between agents were ever lost or interfered with.

Two existing CASA communication protocols were modularized into conversationlets and used in the construction of conversation templates. The relationship between the existing *request* and *propose* protocols are close enough that only one template was deemed necessary to implement both conversations for the respective agent roles. A new *subscribe* conversation template was constructed using many of the same modular components. All of the conversation templates described here prescribe different agent behaviour depending on the agent's role in the conversation. As such, templates are provided for agents in both the *server* and *client* roles using the same conversationlets wherever appropriate.

Social semantics' chief component, the social commitment, was used to connect conversationlets into coherent conversations. Most communicative acts either generate or fulfill a social commitment. This, in turn, is determined by the policies defined within the conversation's conversationlets. The creation and fulfilment of social commitments motivates agents' dialogues by dictating which performative message will fulfil a given commitment. When a conversationlet has run its course and can no longer fulfil a particular commitment, another conversationlet, which is able to do so, is called upon. A conversation constructed from conversationlets terminates once all social commitments have been fulfilled.

Having modified CASA to support its agents to engage in first-order conversations, its existing communication protocols were redeployed and new conversations were created. The conversations necessary to form a MAS were designated *primary.* The new *secondary* conversations could not take place unless an MAS had already been established by agents that had engaged in the primary conversations. The secondary conversations deployed here were designed to simulate a financial transaction and several different auction types[1]. The auction conversations extended the functionality established by the transaction conversations. Similarly, CASA's base agent class was extended to create a specialized transaction agent class, which was, in turn, extended to create the specialized auction agent classes. Each specialized agent was equipped to engage in its designated first-order conversation.

[1]English, Dutch, Sealed Bid, and Vickrey

All of the design and implementation steps taken over the course of this research and described in this chapter culminate in a MAS which is unconstrained by the limitations inherent in other MASs. Specifically, the newly modified CASA allows for an ease of design and implementation not available in the comparable JADE MAS[2]. The remainder of this section describes CASA's improvements.

JADE relies on `MessageTemplate`s to filter and direct messages passed between agents. While simple agents may define loose criteria by which to identify and direct relevant messages, agents capable of complex interactions require increasingly refined criteria, which requires an increasing number of `MessageTemplate`s. As a result, designing agents capable of complex interactions in JADE quickly becomes unwieldy.

CASA, in contrast with JADE, does not burden the agent designer with message filters requiring ever-increasing complexity. Though CASA does filter incoming and outgoing messages, the first and most important point of consideration is a message's unique conversation ID. This is used by an agent to determine if an appropriate conversation has already been instantiated, which in turn provides the context by which the CASA agent interprets the message. If no such conversation exists, the CASA agent refers to its policies to see if it is necessary to instantiate one. CASA's use of the first-order conversational mechanism renders the complexities of JADE's `MessageTemplate`s and `Behaviours` unnecessary. As a result, it is relatively easy to design agents capable of engaging in complex communicative interactions.

CASA also makes significant improvements over JADE in how it directs agent behaviours. These improvements are directly related to the well-defined protocols outlined in CASA's conversation templates and the modular, reusable conversationlets of which they are comprised. In CASA, agent behaviour is tied directly to the conversational turns defined in every conversation protocol. JADE, by contrast, encompasses agent behaviour

[2]JADE's limitations were identified in Section 5.4.

in its `Behaviour` class, which is left to determine how far a particular interaction has progressed and what behaviour is appropriate. With complex agents, this leads to long `Behaviour` classes organized as finite state machines. This makes such classes difficult, if not impossible to extend, as was the experience with the JADE `TransactionAgent` described in Section 5.4.2. CASA is not limited in this respect, as it was demonstrated that its agents are easily extensible in Chapter 5.

Finally, unlike JADE, CASA agents employ an in-built fault tolerance for communication failures. This improvement is the result of CASA's conversation templates, which define default failure-prompted conversational turns. This is also directly related to the assertion made here that conversations terminate when no unfulfilled social commitments remain. If some communication failure has occurred, unfulfilled social commitments will be fulfilled as a result of the agent's response to the failure, which is predefined in the associated conversation's template. Here, the CASA agent designer may simply rely on the default behaviour, or specify a custom agent response. JADE offers no such functionality. In the event that a JADE interaction fails, a `Behaviour`'s finite state machine could potentially remain unterminated, unless the agent designer took the appropriate precautions to prevent this from happening (e.g., specifying a timeout condition).

This research has established the viability of conversations as the primary mode of agent communication. It identified recurring sequences of performative messages in existing communication protocols and modularized them as reusable conversationlets. Relationships were then established between these conversationlets using policy-guided social commitments in order to construct complete, coherent agent dialogs. Used in this way, first-order conversations provide context to the messages agents exchange and guide their subsequent actions.

6.1 Future Work

This research raises speculative questions about the human observer's role in agent conversations and practical questions concerning communication stability and security. Whether practical or speculative, these avenues of discussion are potentially worth investigating. An overview of issues raised and observations made follows.

6.1.1 Communication Stability and Security

As stated in the previous section, the investigation described in Chapter 5 made assumptions about the reliability of the communication infrastructure and the sociability of the agents that formed the various societies. Here, no messages were lost due to unstable connectivity or any other factors. Nor was there any interference from anti-social agents that may otherwise disrupt agent interaction within the society. This research did not directly address any of these questions concerning communication stability or security.

Although communication stability was not directly addressed, CASA's new conversation templates are equipped to handle a variety of the uncertainties involved with network communication. Take, for example, the CASA *request* protocol illustrated in Figure 4.1. This protocol, and the conversation template embodied by it, allow both the server and client agent to send `not-understood` and `timeout` performative messages where appropriate. This accounts for errors potentially resulting from mangled data and limited network connectivity, respectively. In the investigative situations where these problems did arise, the mechanisms put in place did work, but they have not been subjected to a formal evaluation, nor is it certain that every failure-causing eventuality has been accounted for.

The success of investigative component of this research depended on the sociability of the agents involved. A social agent is one that adheres to the society's policies. The issue of sociability overlaps many of the issues that give rise to security concerns. At this point, it is entirely uncertain as to how first-order conversations might accommodate, and potentially mitigate, problems caused by anti-social agents, whether they are malfunctioning in some capacity or malicious by design. It may be the case that first-order conversations aggravate these problems. Either way, considerations concerning first-order conversations and anti-social agents is a necessary area of investigation.

6.1.2 Internal Conversations and Human-Computer Interaction

In pursuing the task of equipping CASA agents to process all communicative messages within a conversational context, it was found necessary to enable agents to conduct these conversations *internally*. An internalized conversation is one in which a solitary agent engages in a dialogue with itself. As such, the solitary agent takes on the roles of both the client and server agent. Though there was little difficulty in implementing such functionality, the fact that it was necessary gives rise to questions concerning human-agent interaction and may encourage an expansion of the conversational paradigm established in this thesis.

The pre-existing CASA MAS produced one scenario where it was expedient for an agent to take on both the server and client roles. On initialization, the Local Area Coordinator agent is required to engage in a `register_instance` conversation. This is a primary conversation required of all agents in a CASA MAS. Member agents engage in a `register_instance` conversation with the LAC in order to make their presence known to the wider society. The LAC, as a MAS member agent, is not excepted from this requirement. As a client, the LAC sends itself a `request register_instance` message, which immediately requires it to process the `MessageEvent` in the server role. The conversation proceeds and terminates as it would if distinct agents were exchanging `register_instance` messages.

The opportunity to utilize internalized secondary conversations arose as a result of design considerations made in the implementation of the abstract `AuctionAgent` class and its associated GUI class. Specifically, it was deemed appropriate for an auctioneer to conduct an internalized `make_auction_cd` conversation whenever the human observer pressed the *Auction* button on the agent's GUI[3]. This conversation is the first in a series of conversations conducted in order to simulate an auction from start to finish. This is a matter of some interest. Since the auction is conducted at the behest of a human observer, the human effectively takes on the client role in the conversation. The human could, if thought appropriate, be provided the means to manually engage in conversation with the auctioneer through some GUI created for that purpose. Here, the internalized secondary conversation could be made interchangeable with policy-driven social commitments formed in a human observer equipped with the means to engage the society.

The preceding observation gives rise to the question of how humans are to relate to agent societies. Artificial agents and humans could easily be made to engage one another under the auspices of first-order conversations. Or, human interests could be mediated by an avatar agent present in the MAS, which would then take direction from the human it represents. However, the manner in which the human engages the avatar could just as easily be conducted as first-order conversations. This gives rise to the question of whether there is any difference between a human-agent conversation and an agent unaware of the external factors motivating the internalized conversations conducted in its own *head*. In exploring this question, it may be shown that communicative interaction between humans and agents should also be conducted within the context of first-order conversations.

[3]It was also found expedient to have the auctioneer send itself an `inform auction_starting` message when it is time to call for bids, though `inform` messages do not necessarily need to be exchanged within a conversation.

Glossary

ACL a standardized mechanism which, minimally, defines the syntax of messages passed between agents. ACLs may also define semantics. The ACL produced by the Foundation for Intelligent Physical Agents (FIPA) is of primary interest here(FIPA, 1997).

act an umbrella term denoting the actions an agent is capable of performing in its environment. Acts may be communicative (as per Austin and Searle's *speech acts* (Austin, 1975; Searle, 1969)) or *physical* in nature (as when an agent behaves in such a way as to change the state of its *environment*).

agent a software entity, the primary characteristics of which are the abilities to perceive and act upon its environment autonomously, and to communicate with other agents. There is no universally accepted definition of agency (Wooldridge and Jennings, 1994).

CD a specialized CASA agent whose primary purpose is to act as a communication proxy. CASA agents may request membership in a Cooperation Domain in order to communicate with other member agents.

consideration the internal contemplative action undertaken by an agent upon receipt of a message or having perceived an environmental state potentially requiring some action.

conversation a sequence of communicative acts motivated by social commitments formed in accordance with a given set of policies. Individual speech acts are not often issued in isolation, so it is assumed that any given act may be linked with one that came previously and that it may result in future speech acts (Ferber, 1999). A conversation only exist so long as social commitments formed during its execution remain unfulfilled.

conversationlet these help dictate the appropriate sequence of communicative acts within the context of a container conversation. Conversationlets, under the dictates of policies, form social commitments that will be fulfilled by performatives in other conversationlets.

environment the operating conditions with which agents must contend, which is understood from the perspective of those occupying agents. For example, an environment's characteristics may be determined, in part, by an agent's ability to *know* or *control* its surroundings, and the degree to which the acts performed therein produce predictable outcomes and depend upon historical environmental states (Huhns and Singh, 1998).

FIPA a standards organization for agents and multi-agent systems that defines the FIPA Communicative Act Library Specification (FIPA, 2002b).

global policy a policy beyond the context of a conversation. When an event is received, an agent will always refer to its global policies to see if any apply plural.

LAC the first agent initialized in a CASA society. Its primary roles are to instantiate and register agents..

MAS a system comprised of an environment and two or more communicative software agents capable of perceiving and acting upon that environment (Ferber, 1999)..

mentalistic semantics the semantic mechanism that demands agents reveal their beliefs, desires, and intentions (BDI) to other agents (Singh, 2000).

message the formalized communicative packet passed between agents engaged in conversation. The messages discussed hereafter contain performatives.

ontology names the actions an agent is capable of performing and those actions' relationship to the other actions named therein (Kremer and Flores, 2006).

open MAS a multi-agent system capable of accommodating foreign agents. That is, a system equipped to receive external messages and to deliver them to the agent addressed therein.

performative a communicative action. Performatives are rooted philosophically in Austin and Searle's *speech acts* (Austin, 1975; Searle, 1969) and are defined by an ACL for use in a MAS.

policy policies determine what social commitments are formed and fulfilled upon the issuance of a message.

role an agent engaged in a conversationlet plays one of two roles: *client* or *server*. Roles are not necessarily constant over the course of a conversation. Agent roles typically reverse, especially at a conversation's conclusion..

social commitment a construct whereby a debtor agent is obliged to perform some action for a creditor agent. The formation and fulfilment of social commitments is dictated by the agent society's policies(Singh, 2000).

social semantics the mechanism that depends on social commitments to convey meaning in an ACL (Singh, 2000).

speech act as per Austin and Searle. A communicative act undertaken by an agent. Speech acts and performatives are treated as synonymous (Austin, 1975; Searle, 1969).

Bibliography

Alagar, V. and Zheng, M. (2005). A software architecture for multi-agent systems. In Hao, Y., Liu, J., Wang, Y., Cheung, Y.-m., Yin, H., Jiao, L., Ma, J., and Jiao, Y.-C., editors, *Computational Intelligence and Security*, volume 3801 of *Lecture Notes in Computer Science*, pages 303–312. Springer Berlin / Heidelberg. 10.1007/11596448_44.

Amgoud, L. and de Saint-Cyr, F. D. (2008). A new semantics for acl based on commitments and penalties. *Int. J. Intell. Syst.*, 23(3):286–312.

Austin, J. L. (1975). *How to do things with Words: the William James Lectures delivered at Harvard University in 1955*. Clarendon Press, Oxford, 2 edition.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE*. Wiley.

Booch, G. (1995). *Object solutions: managing the object-oriented project*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible markup language (xml) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126.

Chaib-Draa, B., Labrie, M.-A., Bergeron, M., and Pasquier, P. (2006). Diagal: An agent communication language based on dialogue games and sustained by social commitments. *Autonomous Agents and Multi-Agent Systems*, 13(1):61–95.

Dignum, F., Dignum, V., Thangarajah, J., Padgham, L., and Winikoff, M. (2007). Open agent systems??? In *Agent-Oriented Software Engineering*, pages 73–87.

Ferber, J. (1999). *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Professional.

Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). KQML as an agent communication language. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 456–463, New York, NY, USA. ACM.

Finin, T., Fritzson, R., Mckay, D., Shapiro, S., and Wiederhold, G. (April 1992). An overview of kqml: A knowledge query and manipulation language. Department of Computer Science, University of Maryland, Baltimore County, Technical report.

FIPA (1997). Fipa97 specification part 2: Agent communication language. Technical report, Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00018/OC00018.pdf.

FIPA (2002a). Fipa acl message structure specification. FIPA Agent Communication Language Specifications SC00061G, The Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00061.

FIPA (2002b). Fipa communicative act library specification. FIPA Agent Communication Language Specifications SC00037J, The Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00037.

FIPA (2002c). Fipa sl content language specification. FIPA Agent Communication Language Specifications SC00008I, The Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00008/.

Flores, R. A. and Kremer, R. C. (2002). To commit or not to commit: Modelling agent conversations for action. *Computational Intelligence*, 18:2003.

Flores, R. A., Pasquier, P., and Chaib-Draa, B. (2007). Conversational semantics sustained by commitments. *Autonomous Agents and Multi-Agent Systems*, 14(2):165–186.

Fornara, N., Viganò, F., Verdicchio, M., and Colombetti, M. (2008). Artificial institutions: a model of institutional reality for open multiagent systems. *Artificial Intelligence and Law*, 16(1):89–105.

Gelfond, M. and Lifschitz, V. (1993). Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322.

Greaves, M., Holmback, H., and Bradshaw, J. (2000). What is a conversation policy? In *Issues in Agent Communication*, pages 118–131, London, UK. Springer-Verlag.

Grosz, B. J. (1974). The structure of task-oriented dialogs. In *Proceedings of the IEEE Symposium on Speech Recognition: Contributed Papers*, pages 250–253, Pittsburgh, PA†.

Grosz, B. J. and Sidner, C. L. (1986). Attention, intentions, and the structure of discourse. *Comput. Linguist.*, 12(3):175–204.

Guerin, F. and Vasconcelos, W. W. (2007). Component-based standardisation of agent communication. In *DALT*, pages 227–244.

Huhns, M. H. and Singh, M. P. (1998). Agents and multiagent systems: Themes, approaches, and challenges. In Huhns, M. H. and Singh, M. P., editors, *Readings in agents*, pages 1–23, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

JADE. Jade - java agent development framework. `http://jade.tilab.com/`.

Karacapilidis, N. I. and Moraitis, P. (2004). Inter-agent dialogues in electronic marketplaces. *Computational Intelligence*, 20(1):1–17.

Kibble, R. (2006). Speech acts, commitment and multi-agent communication. *Comput. Math. Organ. Theory*, 12(2-3):127–145.

Kinny, D. (2001). Reliable agent communication: A pragmatic perspective. *New Generation Computing*, 19:139–156. 10.1007/BF03037251.

Klemperer, P. (1999). Auction theory: A guide to the literature. Microeconomics, EconWPA.

Kremer, R. and Flores, R. (2005). Using a performative subsumption lattice to support commitment-based conversations. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 114–121, New York, NY, USA. ACM.

Kremer, R., Flores, R., and La Fournie, C. (2004). A performative type hierarchy and other interesting considerations in the design of the casa agent architecture. In Dignum, F., editor, *Advances in Agent Communication*, volume 2922 of *Lecture Notes in Computer Science*, pages 1956–1956. Springer Berlin / Heidelberg. 10.1007/978-3-540-24608-4_4.

Kremer, R. C. and Flores, R. A. (2006). Flexible conversations using social commitments and a performatives hierarchy. In Dignum, F., van Eijk, R. M., and Flores, R. A., editors, *AC*, volume 3859 of *Lecture Notes in Computer Science*, pages 93–108. Springer.

Maudet, N. and Chaib-Draa, B. (2002). Commitment-based and dialogue-game-based protocols: new trends in agent communication languages. *Knowl. Eng. Rev.*, 17(2):157–179.

Mcburney, P. and Parsons, S. (2001). Agent Ludens: Games for Agent Dialogues. In *Proceedings of the 2001 AAAI Spring Symposium on Game Theoretic and Decision Theoretic Agents*. Stanford.

McBurney, P. and Parsons, S. (2002). Games that agents play: A formal framework for dialogues between autonomous agents. *J. of Logic, Lang. and Inf.*, 11(3):315–334.

McBurney, P. and Parsons, S. (2003). The posit spaces protocol for multi-agent negotiation. In *Workshop on Agent Communication Languages*, pages 364–382.

Parsons, S., McBurney, P., and Wooldridge, M. (2003). The mechanics of some formal inter-agent dialogues. In *Workshop on Agent Communication Languages*, pages 329–348.

Parsons, S., Wooldridge, M., and Amgoud, L. (2002). An analysis of formal inter-agent dialogues. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 394–401, New York, NY, USA. ACM.

Pasquier, P. and Chaib-draa, B. (2005). Agent communication pragmatics: the cognitive coherence approach. *Cognitive Systems Research*, 6(4):364 – 395.

Pitt, J. and Mamdani, A. (1999). Some remarks on the semantics of fipa's agent communication language. *Autonomous Agents and Multi-Agent Systems*, 2(4):333–356.

Reed, C. (1998). Dialogue frames in agent communication. *Multi-Agent Systems, International Conference on*, 0:246.

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition.

Searle, J. (1975). *A taxonomy of illocutionary acts*, pages 334–369. University of Minnesota Press, Minneapolis.

Searle, J. R. (1969). *Speech Acts: an Essay in the Philosophy of Language*. Cambridge University Press, Cambridge.

Searle, J. R. (2005). What is an institution? *Journal of Institutional Economics*, 1(1):1–22.

Singh, M. (2000). A social semantics for agent communication languages. In Dignum, F. and Greaves, M., editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Computer Science*, pages 31–45. Springer.

Singh, M. P. (1998). Agent communication languages: Rethinking the principles. *Computer*, 31(12):40–47.

Stergiou, C. and Arys, G. (2003). A policy based framework for software agents. In *IEA/AIE'2003: Proceedings of the 16th international conference on Developments in applied artificial intelligence*, pages 426–436. Springer Springer Verlag Inc.

Verdicchio, M. and Colombetti, M. (2009). Communication languages for multiagent systems. *Computational Intelligence*, 25(2):136–159.

Vickrey, W. (1961). Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37.

Walton, D. and Krabbe, E. (1995). *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. State University of New York Press, New York.

Wooldridge, M. (1999). Intelligent agents. In Weiss, G., editor, *Multiagent Systems*. The MIT Press.

Wooldridge, M. and Jennings, N. R. (1994). Intelligent agents: Theory and practice. *Knowledge Engineering Review*.

Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.